

# Embedded Software for SoC

Edited by

Ahmed Amine Jerraya, Sungjoo Yoo,  
Diederik Verkest and Norbert Wehn

Kluwer Academic Publishers



# EMBEDDED SOFTWARE FOR SoC

*This page intentionally left blank*

# Embedded Software for SoC

Edited by

**Ahmed Amine Jerraya**

*TIMA Laboratory, France*

**Sungjoo Yoo**

*TIMA Laboratory, France*

**Diederik Verkest**

*IMEC, Belgium*

and

**Norbert Wehn**

*University of Kaiserslautern, Germany*

**KLUWER ACADEMIC PUBLISHERS**

NEW YORK, BOSTON, DORDRECHT, LONDON, MOSCOW

eBook ISBN: 0-306-48709-8  
Print ISBN: 1-4020-7528-6

©2004 Springer Science + Business Media, Inc.

Print ©2003 Kluwer Academic Publishers  
Dordrecht

All rights reserved

No part of this eBook may be reproduced or transmitted in any form or by any means, electronic, mechanical, recording, or otherwise, without written consent from the Publisher

Created in the United States of America

Visit Springer's eBookstore at:  
and the Springer Global Website Online at:

<http://www.ebooks.kluweronline.com>  
<http://www.springeronline.com>

# DEDICATION

*This book is dedicated to all  
designers working in  
hardware hell.*

*This page intentionally left blank*

# TABLE OF CONTENTS

Dedication	v
Contents	vii
Preface	xiii
Introduction	xv
PART I: EMBEDDED OPERATING SYSTEMS FOR SOC	
Chapter 1	1
APPLICATION MAPPING TO A HARDWARE PLATFORM THROUGH ATOMATED CODE GENERATION TARGETING A RTOS <i>Monica Besana and Michele Borgatti</i>	3
Chapter 2	
FORMAL METHODS FOR INTEGRATION OF AUTOMOTIVE SOFTWARE <i>Marek Jersak, Kai Richter, Razvan Racu, Jan Staschulat, Rolf     Ernst, Jörn-Christian Braam and Fabian Wolf</i>	11
Chapter 3	
LIGHTWEIGHT IMPLEMENTATION OF THE POSIX THREADS API FOR AN ON-CHIP MIPS MULTIPROCESSOR WITH VCI INTERCONNECT <i>Frédéric Pétrot, Pascal Gomez and Denis Hommais</i>	25
Chapter 4	
DETECTING SOFT ERRORS BY A PURELY SOFTWARE APPROACH: METHOD, TOOLS AND EXPERIMENTAL RESULTS <i>B. Nicolescu and R. Velazco</i>	39
PART II: OPERATING SYSTEM ABSTRACTION AND TARGETING	
Chapter 5	53
RTOS MODELLING FOR SYSTEM LEVEL DESIGN <i>Andreas Gerstlauer, Haobo Yu and Daniel D. Gajski</i>	55
Chapter 6	
MODELING AND INTEGRATION OF PERIPHERAL DEVICES IN EMBEDDED SYSTEMS <i>Shaojie Wang, Sharad Malik and Reinaldo A. Bergamaschi</i>	69

Chapter 7		
	SYSTEMATIC EMBEDDED SOFTWARE GENERATION FROM SYSTEMIC	
	<i>F. Herrera, H. Posadas, P. Sánchez and E. Villar</i>	83
	PART III:	
	EMBEDDED SOFTWARE DESIGN AND IMPLEMENTATION	95
Chapter 8		
	EXPLORING SW PERFORMANCE USING SOC TRANSACTION-LEVEL	
	MODELING	
	<i>Imed Moussa, Thierry Grellier and Giang Nguyen</i>	97
Chapter 9		
	A FLEXIBLE OBJECT-ORIENTED SOFTWARE ARCHITECTURE FOR SMART	
	WIRELESS COMMUNICATION DEVICES	
	<i>Marco Götz</i>	111
Chapter 10		
	SCHEDULING AND TIMING ANALYSIS OF HW/SW ON-CHIP	
	COMMUNICATION IN MP SOC DESIGN	
	<i>Youngchul Cho, Ganghee Lee, Kiyoun Choi, Sungjoo Yoo and</i>	
	<i>Nacer-Eddine Zergainoh</i>	125
Chapter 11		
	EVALUATION OF APPLYING SPECC TO THE INTEGRATED DESIGN	
	METHOD OF DEVICE DRIVER AND DEVICE	
	<i>Shinya Honda and Hiroaki Takada</i>	137
Chapter 12		
	INTERACTIVE RAY TRACING ON RECONFIGURABLE SIMD MORPHOSYS	
	<i>H. Du, M. Sanchez-Elez, N. Tabrizi, N. Bagherzadeh,</i>	
	<i>M. L. Anido and M. Fernandez</i>	151
Chapter 13		
	PORTING A NETWORK CRYPTOGRAPHIC SERVICE TO THE RMC2000	
	<i>Stephen Jan, Paolo de Dios, and Stephen A. Edwards</i>	165
	PART IV:	
	EMBEDDED OPERATING SYSTEMS FOR SOC	177
Chapter 14		
	INTRODUCTION TO HARDWARE ABSTRACTION LAYERS FOR SOC	
	<i>Sungjoo Yoo and Ahmed A. Jerraya</i>	179
Chapter 15		
	HARDWARE/SOFTWARE PARTITIONING OF OPERATING SYSTEMS	
	<i>Vincent J. Mooney III</i>	187

Chapter 16

EMBEDDED SW IN DIGITAL AM-FM CHIPSET

*M. Sarlotte, B. Candaele, J. Quevremont and D. Merel* 207

PART V:

SOFTWARE OPTIMISATION FOR EMBEDDED SYSTEMS 213

Chapter 17

CONTROL FLOW DRIVEN SPLITTING OF LOOP NESTS AT THE SOURCE CODE LEVEL

*Heiko Falk, Peter Marwedel and Francky Catthoor* 215

Chapter 18

DATA SPACE ORIENTED SCHEDULING

*M. Kandemir, G. Chen, W. Zhang and I. Kolcu* 231

Chapter 19

COMPILER-DIRECTED ILP EXTRACTION FOR CLUSTERED VLIW/EPIC MACHINES

Satish Pillai and Margarida F. Jacome 245

Chapter 20

STATE SPACE COMPRESSION IN HISTORY DRIVEN QUASI-STATIC SCHEDULING

*Antonio G. Lomeña, Marisa López-Vallejo, Yosinori Watanabe and Alex Kondratyev* 261

Chapter 21

SIMULATION TRACE VERIFICATION FOR QUANTITATIVE CONSTRAINTS

*Xi Chen, Harry Hsieh, Felice Balarin and Yosinori Watanabe* 275

PART VI:

ENERGY AWARE SOFTWARE TECHNIQUES 287

Chapter 22

EFFICIENT POWER/PERFORMANCE ANALYSIS OF EMBEDDED AND GENERAL PURPOSE SOFTWARE APPLICATIONS

*Venkata Syam P. Rapaka and Diana Marculescu* 289

Chapter 23

DYNAMIC PARALLELIZATION OF ARRAY BASED ON-CHIP MULTI-PROCESSOR APPLICATIONS

*M. Kandemir W. Zhang and M. Karakoy* 305

Chapter 24

SDRAM-ENERGY-AWARE MEMORY ALLOCATION FOR DYNAMIC MULTI-MEDIA APPLICATIONS ON MULTI-PROCESSOR PLATFORMS

*P. Marchal, J. I. Gomez, D. Bruni, L. Benini, L. Piñuel, F. Catthoor and H. Corporaal* 319

	PART VII:	
	SAFE AUTOMOTIVE SOFTWARE DEVELOPMENT	331
Chapter 25	SAFE AUTOMOTIVE SOFTWARE DEVELOPMENT	
	<i>Ken Tindell, Hermann Kopetz, Fabian Wolf and Rolf Ernst</i>	333
	PART VIII:	
	EMBEDDED SYSTEM ARCHITECTURE	343
Chapter 26	EXPLORING HIGH BANDWIDTH PIPELINED CACHE ARCHITECTURE FOR SCALED TECHNOLOGY	
	<i>Amit Agarwal, Kaushik Roy and T. N. Vijaykumar</i>	345
Chapter 27	ENHANCING SPEEDUP IN NETWORK PROCESSING APPLICATIONS BY EXPLOITING INSTRUCTION REUSE WITH FLOW AGGREGATION	
	<i>G. Surendra, Subhasis Banerjee and S. K. Nandy</i>	359
Chapter 28	ON-CHIP STOCHASTIC COMMUNICATION	
	<i>Tudor Dumitraş and Radu Mărculescu</i>	373
Chapter 29	HARDWARE/SOFTWARE TECHNIQUES FOR IMPROVING CACHE PERFORMANCE IN EMBEDDED SYSTEMS	
	<i>Gokhan Memik, Mahmut T. Kandemir, Alok Choudhary and Ismail Kadayif</i>	387
Chapter 30	RAPID CONFIGURATION & INSTRUCTION SELECTION FOR AN ASIP: A CASE STUDY	
	<i>Newton Cheung, Jörg Henkel and Sri Parameswaran</i>	403
	PART IX	
	TRANSFORMATIONS FOR REAL-TIME SOFTWARE	419
Chapter 31	GENERALIZED DATA TRANSFORMATIONS	
	<i>V. Delaluz, I. Kadayif, M. Kandemir and U. Sezer</i>	421
Chapter 32	SOFTWARE STREAMING VIA BLOCK STREAMING	
	<i>Pramote Kuacharoen, Vincent J. Mooney III and Vijay K. Madiseti</i>	435

<i>Table of Contents</i>	xi
Chapter 33	
ADAPTIVE CHECKPOINTING WITH DYNAMIC VOLTAGE SCALING IN EMBEDDED REAL-TIME SYSTEMS	
<i>Ying Zhang and Krishnendu Chakrabarty</i>	449
	PART X:
	LOW POWER SOFTWARE
	465
Chapter 34	
SOFTWARE ARCHITECTURAL TRANSFORMATIONS	
<i>Tat K. Tan, Anand Raghunathan and Niraj K. Jha</i>	467
Chapter 35	
DYNAMIC FUNCTIONAL UNIT ASSIGNMENT FOR LOW POWER	
<i>Steve Haga, Natsha Reeves, Rajeev Barua and Diana         Marculescu</i>	485
Chapter 36	
ENERGY-AWARE PARAMETER PASSING	
<i>M. Kandemir, I. Kolcu and W. Zhang</i>	499
Chapter 37	
LOW ENERGY ASSOCIATIVE DATA CACHES FOR EMBEDDED SYSTEMS	
<i>Dan Nicolaescu, Alex Veidenbaum and Alex Nicolau</i>	513
Index	527

*This page intentionally left blank*

## PREFACE

The evolution of electronic systems is pushing traditional silicon designers into areas that require new domains of expertise. In addition to the design of complex hardware, System-on-Chip (SoC) design requires software development, operating systems and new system architectures. Future SoC designs will resemble a miniature on-chip distributed computing system combining many types of microprocessors, re-configurable fabrics, application-specific hardware and memories, all communicating via an on-chip inter-connection network. Designing good SoCs will require insight into these new types of architectures, the embedded software, and the interaction between the embedded software, the SoC architecture, and the applications for which the SoC is designed.

This book collects contributions from the Embedded Software Forum of the Design, Automation and Test in Europe Conference (DATE 03) that took place in March 2003 in Munich, Germany. The success of the Embedded Software Forum at DATE reflects the increasing importance of embedded software in the design of a System-on-Chip.

*Embedded Software for SoC* covers all software related aspects of SoC design

- Embedded and application-domain specific operating systems, interplay between application, operating system, and architecture.
- System architecture for future SoC, application-specific architectures based on embedded processors and requiring sophisticated hardware/software interfaces.
- Compilers and interplay between compilers and architectures.
- Embedded software for applications in the domains of automotive, avionics, multimedia, telecom, networking, . . .

This book is a must-read for *SoC designers* that want to broaden their horizons to include the ever-growing embedded software content of their next SoC design. In addition the book will provide *embedded software designers* invaluable insights into the constraints imposed by the use of embedded software in a SoC context.

Diederik Verkest  
*IMEC*  
*Leuven, Belgium*

Norbert Wehn  
*University of Kaiserslautern*  
*Germany*

*This page intentionally left blank*

# INTRODUCTION

Embedded software is becoming more and more important in system-on-chip (SoC) design. According to the ITRS 2001, “embedded software design has emerged as the most critical challenge to SoC” and “Software now routinely accounts for 80% of embedded systems development cost” [1]. This will continue in the future. Thus, the current design productivity gap between chip fabrication and design capacity will widen even more due to the increasing ‘embedded SoC SW implementation gap’. To overcome the gap, SoC designers should know and master embedded software design for SoC. The purpose of this book is to enable current SoC designers and researchers to understand up-to-date issues and design techniques on embedded software for SoC.

One of characteristics of embedded software is that it is heavily dependent on the underlying hardware. The reason of the dependency is that embedded software needs to be designed in an application-specific way. To reduce the system design cost, e.g. code size, energy consumption, etc., embedded software needs to be optimized exploiting the characteristics of underlying hardware.

Embedded software design is not a novel topic. Then, why do people consider that embedded software design is more and more important for SoC these days? A simple, maybe not yet complete, answer is that we are more and more dealing with platform-based design for SoC [2].

Platform-based SoC design means to design SoC with relatively fixed architectures. This is important to reduce design cycle and cost. In terms of reduction in design cycle, platform-based SoC design aims to reuse existing and proven SoC architectures to design new SoCs. By doing that, SoC designers can save architecture construction time that includes the design cycle of IP (intellectual property core) selection, IP validation, IP assembly, and architecture validation/evaluation.

In platform-based SoC design, architecture design is to configure, statically or dynamically in system runtime, the existing platforms according to new SoC designs [3]. Since the architecture design space is relatively limited and fixed, most of the design steps are software design. For instance, when SoC designers need to implement a functionality that is not implemented by hardware blocks in the platform, they need to implement it in software. As the SoC functionality becomes more complex, software will implement more and more functionality compared to the relatively fixed hardware. Thus, many design optimization tasks will become embedded software optimization ones.

To understand embedded software design for SoC, we need to know current issues in embedded software design. We want to classify the issues into two parts: software reuse for SoC integration and architecture-specific software optimization. Architecture-specific software optimization has been studied for decades. On the other side, software reuse for SoC integration is an important new issue. To help readers to understand better the specific contribution of this book, we want to address this issue more in detail in this introduction.

### SW REUSE FOR SOC INTEGRATION

Due to the increased complexity of embedded software design, the design cycle of embedded software is becoming the bottleneck to reduce time-to-market. To shorten the design cycle, embedded software needs to be reused over several SoC designs. However, the hardware dependency of embedded software makes software reuse very difficult.

A general solution to resolve this software reuse problem is to have a multi-layer architecture for embedded software. Figure 1 illustrates such an architecture. In the figure, a SoC consists of sub-systems connected with each other via a communication network. Within each sub-system, embedded

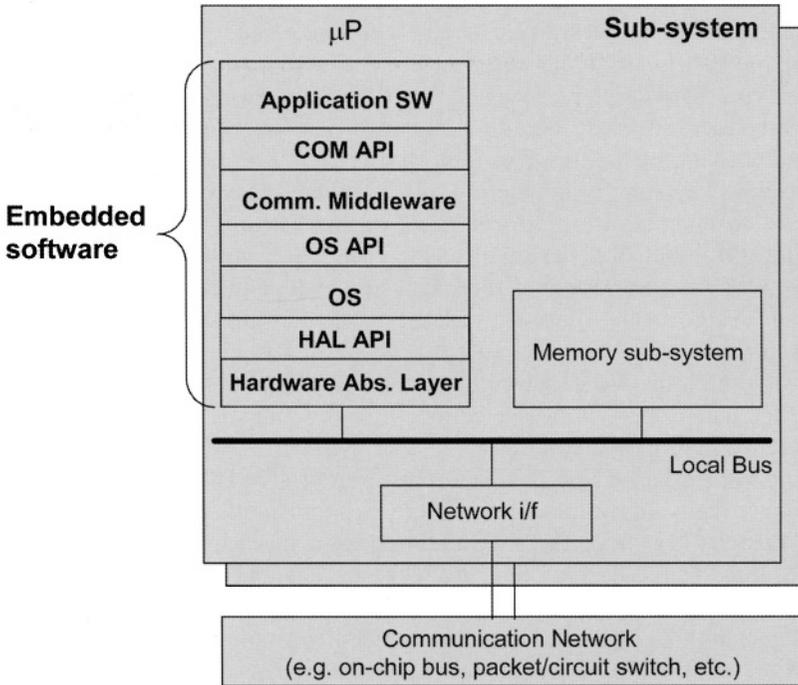


Figure 1. Multi-layer software architecture in SoC.

software consists of several layers: application software, communication middleware (e.g. message passing interface [4]), operating system (OS), and hardware abstraction layer (HAL)). In the architecture, each layer uses an abstraction of the underlying ones. For instance, the OS layer is seen by upper layers (communication middleware and application layers) as an abstraction of the underlying architecture, in the form of OS API (application programming interface), while hiding the details of OS and HAL implementation and those of the hardware architecture.

Embedded software reuse can be done at each layer. For instance, we can reuse an RTOS as a software component. We can also think about finer granularity of software component, e.g. task scheduler, interrupt service routine, memory management routine, inter-process communication routine, etc. [5].

By reusing software components as well as hardware components, SoC design becomes an integration of reused software and hardware components. When SoC designers do SoC integration with a platform and a multi-layer software architecture, the first question can be ‘what is the API that gives an abstraction of my platform?’ We call the API that abstracts a platform ‘platform API’. Considering the multi-layer software architecture, the platform API can be Communication API, OS API, or HAL API. When we limit the platform only to the hardware architecture, the platform API can be an API at transaction level model (TLM) [6]. We think that a general answer to this question may not exist. The platform API may depend on designer’s platforms. However, what is sure is that the platform API needs to be defined (by designers, by standardization institutions like Virtual Socket Interface Alliance, or by anyone) to enable platform-based SoC design by reusing software components.

In SoC design with multi-layer software architecture, another important problem is the validation and evaluation of reused software on the platform. Main issues are related to software validation without the final platform and, on the other hand, to assess the performance of the reused software on the platform. Figure 2 shows this problem more in detail. As shown in the figure,

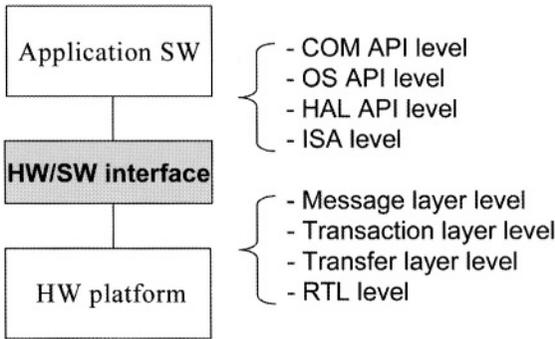


Figure 2. Validation and evaluation of reused embedded software and hardware platform.

software can be reused at one of several abstraction levels, Communication API, OS API, HAL API, or ISA (instruction set architecture) level, each of which corresponds to software layer. The platform can also be defined with its API. In the figure, we assume a hardware platform which can be reused at one of the abstraction levels, message, transaction, transfer layer, or RTL [6]. When SoC designers integrate both reused software and hardware platform at a certain abstraction level for each, the problem is how to validate and evaluate such integration. As more software components and hardware platforms are reused, this problem will become more important.

The problem is to model the interface between reused software and hardware components called ‘hardware/software interface’ as shown in Figure 2. Current solutions to model the HW/SW interface will be bus functional model, BCA (bus cycle accurate) shell, etc. However, they do not consider the different abstraction levels of software. We think that there has been little research work covering both the abstraction levels of software and hardware in this problem.

## GUIDE TO THIS BOOK

The book is organised into 10 parts corresponding to sessions presented at the Embedded Systems Forum at DATE’03. Both software reuse for SoC and application specific software optimisations are covered.

The topic of **Software reuse for SoC** integration is explained in three parts “Embedded Operating System for SoC”, “Embedded Software Design and Implementation”, “Operating System Abstraction and Targeting”. The key issues addressed are:

- The **layered software architecture and its design** in chapters 3 and 9.
- The **OS layer design** in chapters 1, 2, 3, and 7.
- The **HAL layer** in chapter 1.
- The **problem of modelling the HW/SW interface** in chapters 5 and 8.
- **Automatic generation of software layers**, in chapters 6 and 11.
- **SoC integration** in chapters 10, 12 and 13.

**Architecture-specific software optimization problems** are mainly addressed in five parts, “Software Optimization for Embedded Systems”, “Embedded System Architecture”, “Transformations for Real-Time Software”, “Energy Aware Software Techniques”, and “Low Power Software”. The key issues addressed are:

- **Sub-system-specific techniques** in chapters 18, 19, 26, 29, 30 and 31.
- **Communication-aware techniques** in chapters 23, 24, 27 and 28.
- **Architecture independent solutions** which perform code transformation to enhance performance or to reduce design cost without considering specific target architectures are presented in chapters 17, 20, 21 and 33.

- **Energy-aware techniques** in chapters 22, 23, 24, 34, 35, 36 and 37.
- **Reliable embedded software design techniques** in chapters 4, 25 and 32.

## REFERENCES

1. *International Technology Roadmap for Semiconductors*, available at <http://public.itrs.net/>
2. Alberto Sangiovanni-Vincentelli and Grant Martin. "Platform-Based Design and Software Design Methodology for Embedded Systems." *IEEE Design & Test of Computers*, November/December 2001.
3. Henry Chang, Larry Cooke, Merrill Hunt, Grant Martin, Andrew McNelly, and Lee Todd. *Surviving the SOC Revolution, A Guide to Platform-Based Design*. Kluwer Academic Publishers, 1999.
4. *The Message Passing Interface Standard*, available at <http://www-unix.mcs.anl.gov/mpi/>
5. Anthony Massa. *Embedded Software Development with eCos*. Prentice Hall, November 2002.
6. *White Paper for SoC Communication Modeling*, available at [http://www.synopsys.com/products/cocentric\\_studio/communication\\_wp10.pdf](http://www.synopsys.com/products/cocentric_studio/communication_wp10.pdf)

Sungjoo Yoo  
TIMA  
Grenoble, France

Ahmed Amine Jerraya  
TIMA  
Grenoble, France

*This page intentionally left blank*

**PART I:**

**EMBEDDED OPERATING SYSTEMS FOR SOC**

*This page intentionally left blank*

# Chapter 1

## APPLICATION MAPPING TO A HARDWARE PLATFORM THROUGH AUTOMATED CODE GENERATION TARGETING A RTOS

### *A Design Case Study*

Monica Besana and Michele Borgatti

*STMicronics, Central R&D – Agrate Brianza (MI), Italy*

**Abstract.** Consistency, accuracy and efficiency are key aspects for practical usability of a system design flow featuring automatic code generation. Consistency is the property of maintaining the same behavior at different levels of abstraction through synthesis and refinement, leading to functionally correct implementation. Accuracy is the property of having a good estimation of system performances while evaluating a high-level representation of the system. Efficiency is the property of introducing low overheads and preserving performances at the implementation level.

RTOS is a key element of the link to implementation flow. In this paper we capture relevant high-level RTOS parameters that allow consistency, accuracy and efficiency to be verified in a top-down approach. Results from performance estimation are compared against measurements on the actual implementation. Experimental results on automatically generated code show design flow consistency, an accuracy error less than 1% and an overhead of about 11.8% in term of speed.

**Key words:** design methodology, modeling, system analysis and design, operating systems

## 1. INTRODUCTION

Nowadays, embedded systems are continuously increasing their hardware and software complexity moving to single-chip solutions. At the same time, market needs of System-on-Chip (SoC) designs are rapidly growing with strict time-to-market constraints. As a result of these new emerging trends, semiconductor industries are adopting hardware/software co-design flows [1, 2], where the target system is represented at a high-level of abstraction as a set of hardware and software reusable macro-blocks.

In this scenario, where also applications complexity is scaling up, real-time operating systems (RTOS) are playing an increasingly important role. In fact, by simplifying control code required to coordinate processes, RTOSs provide a very useful abstraction interface between applications with hard real-time requirements and the target system architecture. As a consequence, availability

---

This work is partially supported by the Medea+ A502 MESA European Project.

of RTOS models is becoming strategic inside hardware/software co-design environments.

This work, based on Cadence Virtual Component Co-design (VCC) environment [3], shows a design flow to automatically generate and evaluate software – including a RTOS layer – for a target architecture. Starting from executable specifications, an untimed model of an existing SoC is defined and validated by functional simulations. At the same time an architectural model of the target system is defined providing a platform for the next design phase, where system functionalities are associated with a hardware or software architecture element. During this mapping phase, each high-level communication between functions has to be refined choosing the correct protocol from a set of predefined communication patterns. The necessary glue for connecting together hardware and software blocks is generated by the interface synthesis process.

At the end of mapping, software estimations have been performed before starting to directly simulate and validate generated code to a board level prototype including our target chip.

Experimental results show a link to implementation consistency with an overhead of about 11.8% in term of code execution time. Performance estimations compared against actual measured performances of the target system show an accuracy error less than 1%.

## 2. SPEECH RECOGNITION SYSTEM DESCRIPTION

A single-chip, processor-based system with embedded built-in speech recognition capabilities has been used as target in this project. The functional block diagram of the speech recognition system is shown in Figure 1-1. It is basically composed by two hardware/software macro-blocks.

The first one, simply called front-end (FE), implements the speech acquisition chain. Digital samples, acquired from an external microphone, are processed (Preproc) frame by frame to provide a sub-sampled and filtered speech data to EPD and ACF blocks. While ACF computes the auto-correlation function, EPD performs an end-point detection algorithm to obtain silence-speech discrimination.

ACF concatenation with the linear predictive cepstrum block (LPC) translates each incoming word (i.e. a sequence of speech samples) into a variable-length sequence of cepstrum feature vectors [4]. Those vectors are then compressed (Compress) and transformed (Format) in a suitable memory structure to be finally stored in RAM (WordRam).

The other hardware/software macro-block, called back-end (BE), is the SoC recognition engine where the acquired word (WordRAM) is classified comparing it with a previously stored database of different words (Flash Memory).

This engine, based on a single-word pattern-matching algorithm, is built by two nested loops (DTW Outloop and DTW Innerloop) that compute L1 or

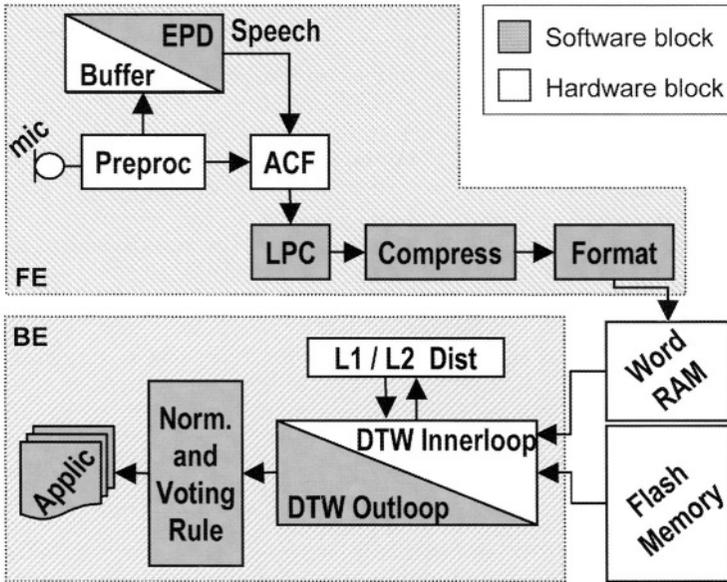


Figure 1-1. System data flow.

L2 distance between frames of all the reference words and the unknown one. Obtained results are then normalized (Norm-and-Voting-Rule) and the best distance is supplied to the application according to a chosen voting rule.

The ARM7TDMI processor-based chip architecture is shown in Figure 1-2. The whole system was built around an AMBA bus architecture, where a bus bridge connects High speed (AHB) and peripherals (APB) buses. Main targets on the AHB system bus are:

- a 2Mbit embedded flash memory (e-Flash), which stores both programs and word templates database;
- the main processor embedded static RAM (RAM);
- a static RAM buffer (WORDRAM) to store intermediate data during the recognition phase.

The configurable hardwired logic that implements speech recognition functionalities (Feature Extractor and Recognition Engine) is directly connected to the APB bus.

### 3. DESIGN FLOW DESCRIPTION

In this project a top-down design flow has been adopted to automatically generate code for a target architecture. Figure 1-3 illustrates the chosen approach.

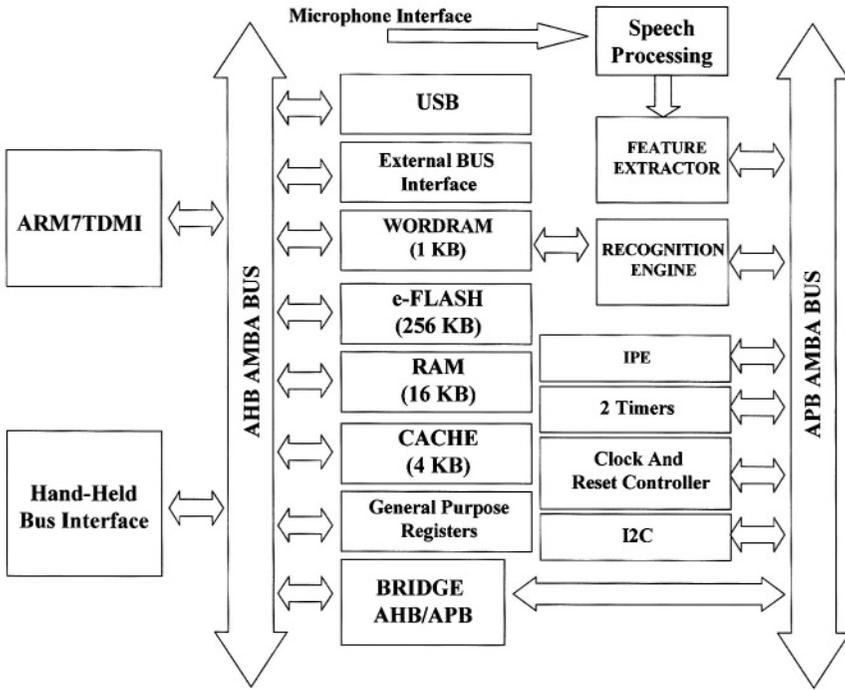


Figure 1-2. System architecture.

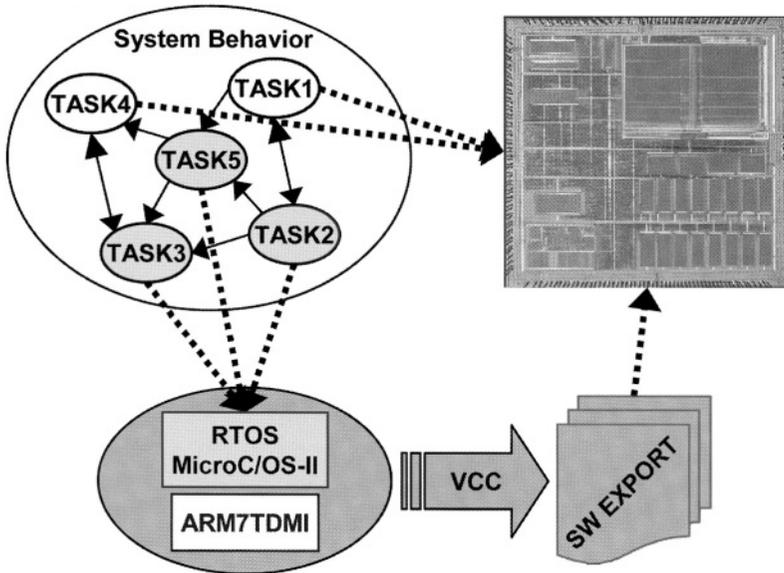


Figure 1-3. Project design flow.

Starting from a system behavior description, hardware and software tasks have been mapped to the target speech recognition platform and to MicroC/OS-II (a well-known open-source and royalties-free pre-emptive real-time kernel [5]) respectively.

Then mapping and automatic code generation phases allow to finally simulate and validate the exported software directly on a target board.

In the next sections a detailed description of the design flow is presented.

### 3.1. Modeling and mapping phases

At first, starting from available executable specifications, a behavioral description of the whole speech recognition system has been carried out. In this step of the project FE and BE macro-blocks (Figure 1-1) have been split in 21 tasks, each one representing a basic system functionality at untimed level, and the obtained model has been refined and validated by functional simulations. Behavioral memories has been included in the final model to implement speech recognition data flow storage and retrieval.

At the same time, a high-level architectural model of the ARM7-based platform presented above (Figure 1-2) has been described. Figure 1-4 shows the result of this phase where the ARM7TDMI core is connected to a MicroC/OS-II model that specifies tasks scheduling policy and delays associated with tasks switching. This RTOS block is also connected to a single task scheduler (Task), that allows to transform a tasks sequence in a single task, reducing software execution time.

When both descriptions are completed, the mapping phase has been started. During this step of the design flow, each task has been mapped to a hardware or software implementation (Figure 1-5), matching all speech recognition

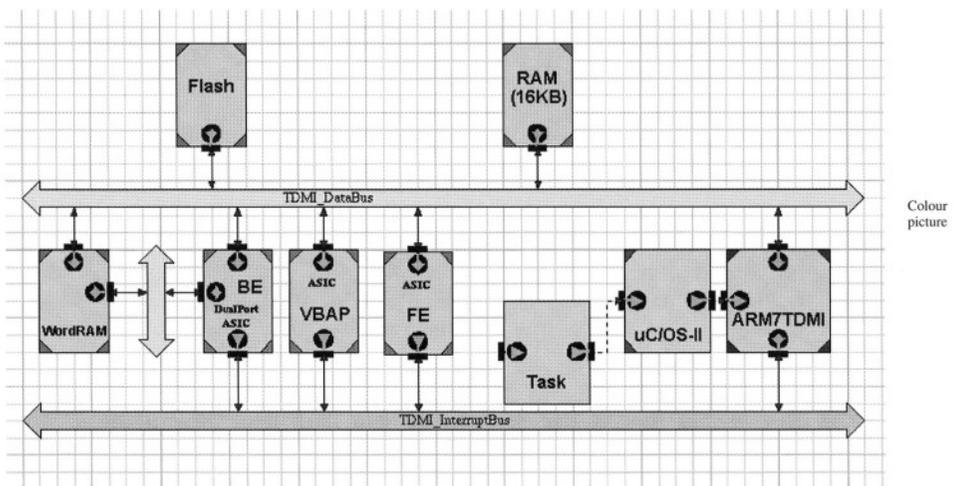
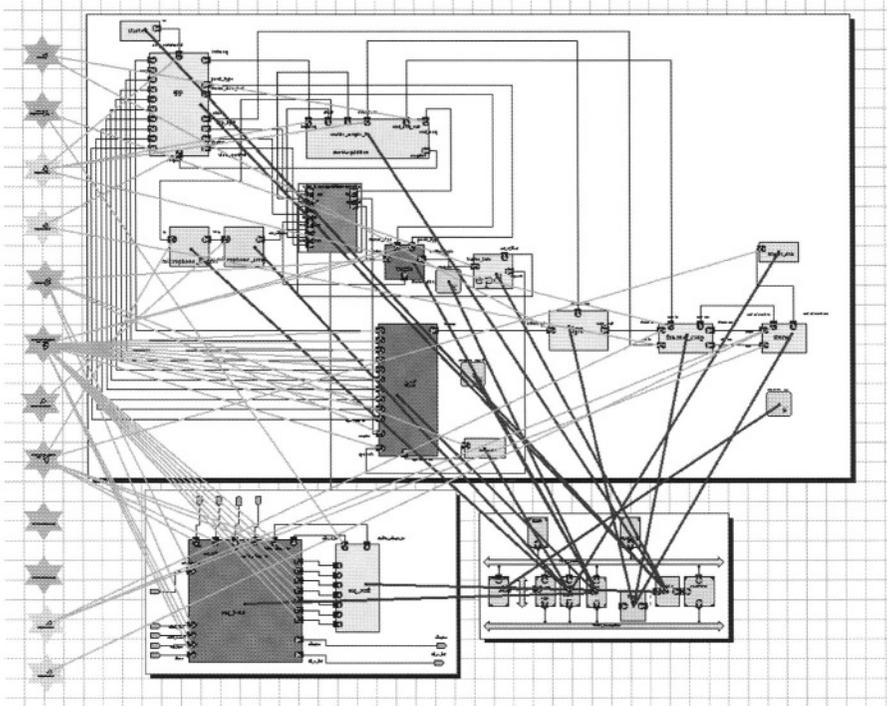


Figure 1-4. Architecture model.



Colour picture

Figure 1-5. FE blocks mapping.

platform requirements in order to obtain code that can be directly executed on target system. To reach this goal the appropriate communication protocol between modeled blocks has had to be selected from available communication patterns. Unavailable communication patterns have been implemented to fit the requirements of the existing hardware platform.

### 3.2. Software performance estimation

At the end of mapping phase, performance estimations have been carried out to verify whether the obtained system model meets our system requirements. In particular most strict constraints are in term of software execution time.

These simulations have been performed setting clock frequency to 16 MHz and using the high-level MicroC/OS-II parameter values obtained via RTL-ISS simulation (Table 1-1) that describe RTOS context switching and interrupt latency overheads. In this scenario the ARM7TDMI CPU architectural element has been modeled with a processor basis file tuned on automotive applications code [6].

Performance results show that all front-end blocks, which are system blocks with the hard-real time constraints, require 6.71 ms to complete their execu-

Table 1-1. RTOS parameters.

	Cycles	16MHz
Start_overhead (delay to start a reaction)	~220	0.014 ms
Finish_overhead (delay to finish a reaction)	~250	0.016 ms
Suspend_overhead (delay to suspend a reaction)	~520	0.033 ms
Resume_overhead (delay to resume a preempted reaction)	~230	0.014 ms

tion. This time does not include RTOS timer overhead that has been estimated via RTL-ISS simulations in 1000 cycles (0.0633 ms at 16 MHz).

Setting MicroC/OS-H timer to a frequency of one tick each ms, all front-end blocks present an overall execution time of 7.153 ms. Since a frame of speech (the basic unit of work for the speech recognition platform) is 8 ms long, performance simulations show that generated code, including the RTOS layer, fits hard real time requirements of the target speech recognition system.

### 3.3. Code generation and measured results

Besides evaluating system performances, VCC environment allows to automatically generate code from system blocks mapped software. This code, however, does not include low-level platform dependent software. Therefore, to execute it directly on the target chip, we have had to port MicroC/OS-II to the target platform and then this porting has been compiled and linked with software generated when the mapping phase has been completed. Resulting image has been directly executed on a board prototype including our speech recognition chip in order to prove design flow consistency.

The execution of all FE blocks, including an operating system tick each 1 ms, results in an execution time of 7.2 ms on the target board (core set to 16 MHz). This result shows that obtained software performance estimation presents an accuracy error less than 1 % compared to on SoC execution time.

To evaluate design flow efficiency we use a previously developed C code that, without comprising a RTOS layer, takes 6.44 ms to process a frame of speech at 16 MHz. Comparing this value to the obtained one of 7.2 ms, we get an overall link to implementation overhead, including MicroC/OS-II execution time, of 11.8%.

#### 4. CONCLUSIONS

In this paper we have showed that the process of capturing system functionalities at high-level of abstraction for automatic code generation is consistent. In particular high-level system descriptions have the same behavior of the execution of code automatically generated from the same high-level descriptions.

This link to implementation is a key productivity improvement as it allows implementation code to be derived directly by the models used for system level exploration and performance evaluation. In particular an accuracy error less than 1% and maximum execution speed reduction of about 11.8% has been reported. We recognize this overhead to be acceptable for the implementation code of our system.

Starting from these results, the presented design flow can be adopted to develop and evaluate software on high-level model architecture, before target chip will be available from foundry.

At present this methodology is in use to compare software performances of different RTOSs on our speech recognition platform. This to evaluate which one could best fit different speech application target constraints.

#### ACKNOWLEDGEMENTS

The authors thank M. Selmi, L. Calli, F. Lertora, G. Mastrorocco and A. Ferrari for their helpful support on system modeling. A special thank to P.L. Rolandi for his support and encouragement.

#### REFERENCES

1. G. De Micheli and R.K. Gupta. "Hardware/Software Co-Design." *Proceedings of the IEEE*, Vol. 85, pp. 349-365, March 1997.
2. W. Wolf. *Computers as Components – Principles of Embedded Computing System Design*. Morgan Kaufmann, 2001.
3. S. J. Krolikoski, F. Schirrmeister, B. Salefski, J. Rowson, and G. Martin. "Methodology and Technology for Virtual Component Driven Hardware/Software Co-Design on the System-Level." *Proceedings of the IEEE International Symposium on Circuits and Systems*, Vol. 6, 1999.
4. J. W. Picone. "Signal Modeling Techniques in Speech Recognition." *Proceedings of the IEEE*, Vol. 81, pp. 1215-1247, September 1993.
5. J. J. Labrosse. "MicroC/OS-II: The Real-Time Kernel." *R&D Books Lawrence KS*, 1999.
6. M. Baleani, A. Ferrari, A. Sangiovanni-Vincentelli, C. Turchetti. "HW/SW Codesign of an Engine Management System." *Proceedings of Design, Automation and Test in Europe Conference*, March 2000.

# Chapter 2

## FORMAL METHODS FOR INTEGRATION OF AUTOMOTIVE SOFTWARE

Marek Jersak<sup>1</sup>, Kai Richter<sup>1</sup>, Razvan Racu<sup>1</sup>, Jan Staschulat<sup>1</sup>, Rolf Ernst<sup>1</sup>, Jörn-Christian Braam<sup>2</sup> and Fabian Wolf<sup>2</sup>

<sup>1</sup>*Institute for Computer and Communication Network Engineering, Technical University of Braunschweig, D-38106 Braunschweig, Germany;* <sup>2</sup>*Aggregateelektronik-Versuch (Power Train Electronics), Volkswagen AG, D-38436 Wolfsburg, Germany*

**Abstract.** Novel functionality, configurability and higher efficiency in automotive systems require sophisticated embedded software, as well as distributed software development between manufacturers and control unit suppliers. One crucial requirement is that the integrated software must meet performance requirements in a certifiable way. However, at least for engine control units, there is today no well-defined software integration process that satisfies all key requirements of automotive manufacturers. We propose a methodology for safe integration of automotive software functions where required performance information is exchanged while each partner's IP is protected. We claim that in principle performance requirements and constraints (timing, memory consumption) for each software component and for the complete ECU can be formally validated, and believe that ultimately such formal analysis will be required for legal certification of an ECU.

**Key words:** automotive software, software integration, software performance validation, electronic control unit certification

### 1. INTRODUCTION

Embedded software plays a key role in increased efficiency of today's automotive system functions, in the ability to compose and configure those functions, and in the development of novel services integrating different automotive subsystems. Automotive software runs on electronic control units (ECUs) which are specialized programmable platforms with a real-time operating system (RTOS) and domain-specific basic software, e.g. for engine control. Different software components are supplied by different vendors and have to be integrated. This raises the need for an efficient, secure and certifiable software integration process, in particular for safety-critical functions.

The functional software design including validation can be largely mastered through a well-defined process including sophisticated test strategies [6]. However, safe integration of software functions on the automotive platform requires validation of the integrated system's performance. Here, non-functional system properties, in particular timing and memory consumption are

the dominant issues. At least for engine control units, there is today no established integration process for software from multiple vendors that satisfies all key requirements of automotive OEMs (original equipment manufacturers).

In this chapter, we propose a flow of information between automotive OEM, different ECU vendors and RTOS vendors for certifiable software integration. The proposed flow allows to exchange key performance information between the individual automotive partners while at the same time protecting each partner's intellectual property (IP). Particular emphasis is placed on formal performance analysis. We believe that ultimately formal performance analysis will be required for legal certification of ECUs. In principle, analysis techniques and all required information are available today at all levels of software, including individual tasks, the RTOS, single ECUs and networked ECUs. We will demonstrate how these individual techniques can be combined to obtain tight analysis results.

## 2. CURRENT PRACTICE IN SOFTWARE INTEGRATION

The software of a sophisticated programmable automotive ECU, e.g. for power train control, is usually composed of three layers. The lowest one, the system layer consists of the RTOS, typically based on the OSEK [8] automotive RTOS standard, and basic I/O. The system layer is usually provided by an RTOS vendor. The next upward level is the so-called 'basic software' which is added by the ECU vendor. It consists of standard functions that are specific to the role of the ECU. Generally speaking, with properly calibrated parameters, an ECU with RTOS and basic software is a working control unit for its specific automotive role.

On the highest layer there are sophisticated control functions where the automotive OEM uses its vehicle-specific know-how to extend and thus improve the basic software, and to add new features. The automotive OEM also designs distributed vehicle functions, e.g. adaptive cruise-control, which span several ECUs. Sophisticated control and vehicle functions present an opportunity for automotive product differentiation, while ECUs, RTOS and basic functions differentiate the suppliers. Consequently, from the automotive OEM's perspective, a software integration flow is preferable where the vehicle function does not have to be exposed to the supplier, and where the OEM itself can perform integration for rapid design-space exploration or even for a production ECU.

Independent of who performs software integration, one crucial requirement is that the integrated software must meet performance requirements in a certifiable way. Here, a key problem that remains largely unsolved is the reliable validation of performance bounds for each software component, the whole ECU, or even a network of ECUs. Simulation-based techniques for performance validation are increasingly unreliable with growing application and architecture complexity. Therefore, formal analysis techniques which consider

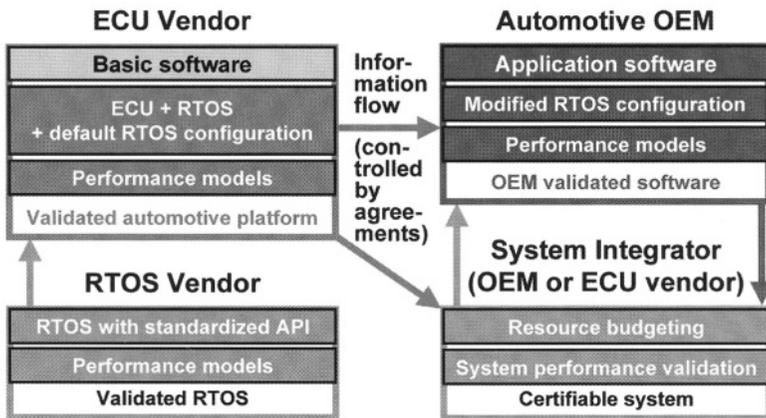
conservative min/max behavioral intervals are becoming more and more attractive as an alternative or supplement to simulation. However, a suitable validation methodology based on these techniques is currently not in place.

### 3. PROPOSED SOLUTION

We are interested in a software integration flow for automotive ECUs where sophisticated control and vehicle functions can be integrated as black-box (object-code) components. The automotive OEM should be able to perform the integration itself for rapid prototyping, design space exploration and performance validation. The final integration can still be left to the ECU supplier, based on validated performance figures that the automotive OEM provides. The details of the integration and certification flow have to be determined between the automotive partners and are beyond the scope of this paper.

We focus instead on the key methodological issues that have to be solved. On the one hand, the methodology must allow integration of software functions without exposing IP. On the other hand, and more interestingly, we expect that ultimately performance requirements and constraints (timing, memory consumption) for each software component and the complete ECU will have to be formally validated, in order to certify the ECU. This will require a paradigm shift in the way software components including functions provided by the OEM, the basic software functions from the ECU vendor and the RTOS are designed.

A possible integration and certification flow which highlights these issues is shown in Figure 2-1. It requires well defined methods for RTOS configuration, adherence to software interfaces, performance models for all entities involved, and a performance analysis of the complete system. Partners



Colour picture

Figure 2-1. Proposed integration and certification flow.

exchange properly characterized black-box components. The required characterization is described in corresponding agreements. This is detailed in the following sections.

## 4. SOFTWARE INTEGRATION

In this section, we address the roles and functional issues proposed in Figure 2-1 for a safe software integration flow, in particular RTOS configuration, communication conventions and memory budgeting. The functional software structure introduced in this section also helps to better understand performance issues which are discussed in Section 5.

### 4.1. RTOS configuration

In an engine ECU, most tasks are either executed periodically, or run synchronously with engine RPM. RTOS configuration (Figure 2-1) includes setting the number of available priorities, timer periods for periodic tasks etc. Configuration of basic RTOS properties is performed by the ECU provider.

In OSEK, which is an RTOS standard widely used in the automotive industry [8], the configuration can be performed in the ‘OSEK implementation language’ (OIL [12]). Tools then build C or object files that capture the RTOS configuration and insert calls to the individual functions in appropriate places. With the proper tool chain, integration can also be performed by the automotive OEM for rapid prototyping and IP protection.

In our experiments we used ERCOSEK [2], an extension of OSEK. In ERCOSEK code is structured into tasks which are further substructured into processes. Each task is assigned a priority and scheduled by the RTOS. Processes inside each task are executed sequentially. Tasks can either be activated periodically with fixed periods using a timetable mechanism, or dynamically using an alarm mechanism.

We configured ERCOSEK using the tool ESCAPE [3]. ESCAPE reads a configuration file that is based on OIL and translates it into ANSI-C code. The individual software components and RTOS functions called from this code can be pre-compiled, black-box components.

In the automotive domain, user functions are often specified with a block-diagram-based tool, typically Simulink or Ascet/SD. C-code is then obtained from the block diagram using the tool’s code generator or an external one. In our case, user functions were specified in Ascet/SD and C-code was generated using the built-in code generator.

### 4.2. Communication conventions and memory budgeting

Black-box components with standard software interfaces are needed to satisfy IP protection. At the same time, validation, as well as modularity and flexi-

bility requirements have to be met. Furthermore, interfaces have to be specific enough that any integrator can combine software components into a complete ECU function.

IP protection and modularity are goals that can be combined if read accesses are hidden and write accesses are open. An open write access generally does not uncover IP. For example, the fact that a function in an engine ECU influences the amount of fuel injected gives away little information about the function's internals. However, the variables read by the function can yield valuable insight into the sophistication of the function.

From an integration perspective, hidden write accesses make integration very difficult since it is unclear when a value is potentially changed, and thus how functions should be ordered. Hidden read accesses pose no problem from this perspective.

The ECU vendor, in his role as the main integrator, provides a list of all pre-defined communication variables to the SW component providers. Some of these may be globally available, some may be exclusive to a subset of SW component providers. The software integrator also budgets and assigns memory available to each SW component provider, separated into memory for code, local data, private communication variables and external I/O variables.

For each software component, its provider specifies the memory actually used, and actual write accesses performed to shared variables. If the ECU exhibits integration problems, then each SW component's adherence to its specification can be checked on the assembly-code level using a debugger. While this is tedious, it allows a certification authority to determine which component is at fault. An alternative may be to use hardware-based memory protection, if it is supported. Reasonable levels of granularity for memory access tables (e.g. vendor, function), and the overhead incurred at each level, still have to be investigated. An analysis of access violation at compile or link-time, on the other hand, seems overly complex, and can be easily tricked, e.g. with hard-to-analyze pointer operations.

Another interesting issue is the trade-off between performance and flexibility as a result of basic software granularity. Communication between SW components is only possible at component boundaries (see communication mechanisms described in Section 4.1). While a fine basic software granularity allows the OEM to augment, replace or introduce new functions at very precise locations, overhead is incurred at every component boundary. On the other hand, coarse basic software may have to be modified more frequently by the ECU vendor to expose interfaces that the OEM requires.

## **5. TIMING ANALYSIS**

The second, more complex set of integration issues deals with software component and ECU performance, in particular timing. Simulation-based

techniques for timing validation are increasingly unreliable with growing application and architecture complexity. Therefore, formal timing analysis techniques which consider conservative min/max behavioral intervals are becoming more and more attractive as an alternative or supplement to simulation. We expect that, ultimately, certification will only be possible using a combination of agreed-upon test patterns and formal techniques. This can be augmented by run-time techniques such as deadline enforcement to deal with unexpected situations (not considered here).

A major challenge when applying formal analysis methodologies is to calculate tight performance bounds. Overestimation leads to poor utilization of the system and thus requires more expensive target processors, which is unacceptable for high-volume products in the automotive industry.

Apart from conservative performance numbers, timing analysis also yields better system understanding, e.g. through visualization of worst case scenarios. It is then possible to modify specific system parameters to assess their impact on system performance. It is also possible to determine the available headroom above the calculated worst case, to estimate how much additional functionality could be integrated without violating timing constraints.

In the following we demonstrate that formal analysis is consistently applicable for single processes, RTOS overhead, and single ECUs, and give an outlook on networked ECUs, thus opening the door to formal timing analysis for the certification of automotive software.

### 5.1. Single process analysis

Formal single process timing analysis determines the worst and best case execution time (WCET, BCET) of one activation of a single process assuming an exclusive resource. It consists of (a) path analysis to find all possible paths through the process, and (b) architecture modeling to determine the minimum and maximum execution times for these paths. The challenge is to make both path analysis and architecture modeling tight.

Recent analysis approaches, e.g. [9], first determine execution time intervals for each basic block. Using an integer linear programming (ILP) solver, they then find the shortest and the longest path through the process based on basic block execution counts and time, leading to an execution time interval for the whole process. The designer has to bound data-dependent loops and exclude infeasible paths to tighten the process-level execution time intervals.

Pipelines and caches have to be considered for complex architectures to obtain reliable analysis bounds. Pipeline effects on execution time can be captured using a cycle-accurate processor core model or a suitable measurement setup. Prediction of cache effects is more complicated. It first requires the determination of worst and best case numbers for cache hits and misses, before cache influence on execution time can be calculated.

The basic-block based timing analysis suffers from the over-conservative assumption of an unknown cache state at the beginning of each basic block.

Therefore, in [9] a modified control-flow graph is proposed capturing potential cache conflicts between instructions in different basic blocks.

Often the actual set of possibly conflicting instructions can be substantially reduced due to input-data-independent control structures. Given the known (few) possible sequences of basic blocks – the so called process segments – through a process, cache tracing or data-flow techniques can be applied to larger code sequences, producing tighter results. Execution time intervals for the complete process are then determined using the known technique from [9] for the remaining data dependent control structures between process segments instead of basic blocks. The improvement in analysis tightness has been shown with the tool SYMTA/P [18].

To obtain execution time intervals for engine control functions in our experiments, we used SYMTA/P as follows: Each segment boundary was instrumented with a trigger point [18, 19], in this case an inline-assembly store-bit instruction changing an I/O pin value. The target platform was a TriCore running at 40 MHz with 1 k direct-mapped instruction cache. Using appropriate stimuli, we executed each segment and recorded the store-bit instruction with a logic state analyzer (LSA). With this approach, we were able to obtain clock-cycle-accurate measurements for each segment. These numbers, together with path information, were then fed into an ILP solver, to obtain minimum and maximum execution times for the example code.

To be able to separate the pure CPU time from the cache miss influences, we used the following setup: We loaded the code into the scratchpad RAM (SPR), an SRAM memory running at processor speed, and measured the execution time. The SPR responds to instruction fetches as fast as a cache does in case of a cache hit. Thus, we obtained measurements for an ‘always hit’ scenario for each analyzed segment. An alternative would be to use cycle-accurate core and cache simulators and pre-load the cache appropriately. However, such simulators were not available to us, and the SPR proved a convenient workaround.

Next, we used a (non cycle-accurate) instruction set simulator to generate the corresponding memory access trace for each segment. This trace was then fed into the DINERO [5] cache simulator to determine the worst and best case ‘hit/miss scenarios’ that would result if the code was executed from external memory with real caching enabled.

We performed experiments for different simple engine control processes. It should be noted that the code did not contain loops. This is because the control-loop is realized through periodic process-scheduling and not inside individual processes. Therefore, the first access to a cache line always resulted in a cache miss. Also, due to the I-cache and memory architectures and cache-miss latencies, loading a cache line from memory is not faster than reading the same memory addresses directly with cache turned off. Consequently, for our particular code, the cache does not improve performance. Table 2-1 presents the results. The first column shows the measured value for the process execution in the SPR (‘always hit’). The next column shows

Table 2-1. Worst case single process analysis and measurements.

Process	Measured WCET in scratchpad	Calculated max # of cache misses	Calculated WCET w/ cache	Measured WCET w/ cache	Measured WCET w/o cache
a	15.3 $\mu$ s	79	47.4 $\mu$ s	46.0 $\mu$ s	46.0 $\mu$ s
b	15.3 $\mu$ s	79	47.4 $\mu$ s	46.7 $\mu$ s	46.7 $\mu$ s
c	20.3 $\mu$ s	104	62.4 $\mu$ s	61.0 $\mu$ s	61.0 $\mu$ s

the worst case number of cache misses. The third column contains the worst case execution times from external memory with cache calculated using the SYMPTA/P approach. The measurements from external memory – with and without cache – are given in the last two columns.

## 5.2. RTOS analysis

Apart from influencing the timing of individual tasks through scheduling, the RTOS itself consumes processor time. Typical RTOS primitive functions are described e.g. in [1]. The most important are: task or context switching including start, preemption, resumption and termination of tasks; and general OS overhead, including periodic timer interrupts and some house-keeping functions. For formal timing analysis to be accurate, the influence of RTOS primitives needs to be considered in a conservative way.

On the one hand, execution time intervals for each RTOS primitive need to be considered, and their dependency on the number of tasks scheduled by the RTOS. The second interesting question concerns patterns in the execution of RTOS primitives, in order to derive the worst and best case RTOS overhead for task response times.

Ideally, this information would be provided by the RTOS vendor, who has detailed knowledge about the internal behavior of the RTOS, allowing it to perform appropriate analyses that cover all corner cases. However, it is virtually impossible to provide numbers for all combinations of targets, compilers, libraries, etc. Alternatively, the RTOS vendor could provide test patterns that the integrator can run on its own target and in its own development environment to obtain the required worst and best case values. Some OS vendors have taken a step in that direction, e.g. [11].

In our case, we did not have sufficient information available. We therefore had to come up with our own tests to measure the influence of ERCOSEK primitives. This is not ideal, since it is tedious work and does not guarantee corner-case coverage. We performed our measurements by first instrumenting accessible ERCOSEK primitives, and then using the LSA-based approach described in Section 5.1. Fortunately, ESCAPE (Section 4.1) generates the ERCOSEK configuration functions in C which then call the corresponding ERCOSEK functions (object code library). The C functions provide hooks for instrumentation.

We inserted code that generates unique port signals before and after accessible ERCOSEK function calls. We measured:

- tt**: time table interrupt, executed whenever the time table needs to be evaluated to start a new task.
- ph start/stop**: the preemption handler is started to hand the CPU to a higher priority task, and stops after returning the CPU to the lower priority task.
- X act**: activates task X. Executed whenever a task is ready for execution.
- X term**: terminate task X is executed after task X has finished.
- X1**: task X is actually executing.

A snapshot of our measurements is shown in Figure 2-2, which displays the time spent in each of the instrumented RTOS functions, as well as execution patterns. As can be seen, time is also spent in RTOS functions which we could not clearly identify since they are hidden inside the RTOS object-code libraries and not visible in the source-code. To included this overhead, we measured the time between **tt** and **X act** (and called this time **Activate Task pre**), the time between **X act** and **X1** (**Task pre**), the time between **X1** and **X term** (**Terminate Task pre**), and the time between **X term** and **ph stop** (**Terminate Task post**). The measurement results are shown in Table 2-2.

Our measurements indicate that for a given ERCOSEK configuration and a given task set, the execution time of some ERCOSEK primitives in the SPR varies little, while there is a larger variation for others. This supports our claim that an RTOS vendor needs to provide methods to appropriately characterize the timing of each RTOS primitive, since the user cannot rely on self-made benchmarks. Secondly, the patterns in the execution of RTOS primitives are surprisingly complex (Figure 2-2) and thus also need to be properly characterized by the RTOS vendor. In the next section we will show

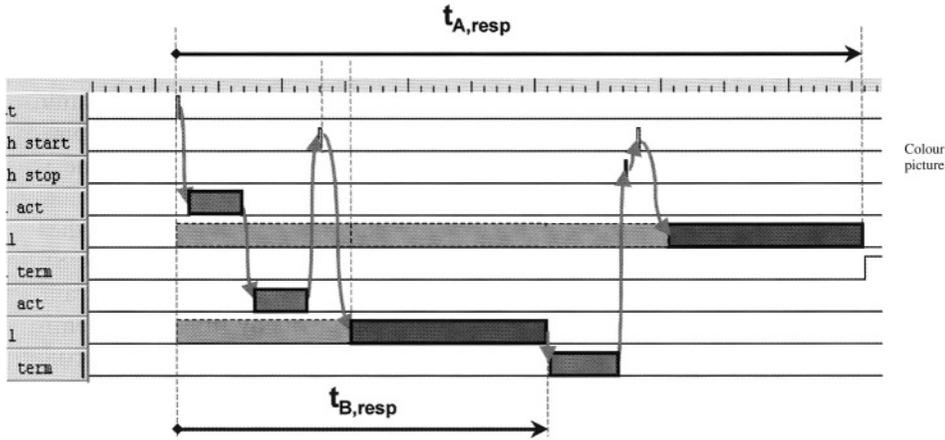


Figure 2-2. RTOS overhead when executing two preemptive tasks activated at the same time.

Table 2-2. RTOS functions timing.

RTOS functions	Measured time value scratchpad	Measured time value w/ cache	Measured time value w/o cache
ActivateTask pre	[0.9; 1.1] $\mu$ s	[1.4; 3.1] $\mu$ s	[2.0; 2.9] $\mu$ s
A act	[4.0; 4.1] $\mu$ s	[7.5; 8.2] $\mu$ s	[9.3; 9.9] $\mu$ s
B act	[4.0; 4.1] $\mu$ s	[6.1; 9.3] $\mu$ s	[9.9; 10.6] $\mu$ s
C act	[4.0; 4.1] $\mu$ s	[4.5; 8.5] $\mu$ s	[10.0; 10.1] $\mu$ s
Task pre	[2.5; 3.7] $\mu$ s	[6.1; 6.9] $\mu$ s	[6.5; 7.5] $\mu$ s
TerminateTask pre	[0.1; 0.2] $\mu$ s	[0.1; 0.6] $\mu$ s	[0.1; 0.6] $\mu$ s
A term	[6.4; 6.5] $\mu$ s	[9.5; 12.8] $\mu$ s	[17.8; 17.9] $\mu$ s
B term	[5.5; 6.7] $\mu$ s	[11.1; 13.3] $\mu$ s	[13.2; 18.5] $\mu$ s
C term	[5.5; 7.0] $\mu$ s	[11.6; 13.6] $\mu$ s	[12.9; 19.4] $\mu$ s
TerminateTask post	[0.7; 0.8] $\mu$ s	[2.5; 3.0] $\mu$ s	[2.5; 3.0] $\mu$ s

that with proper RTOS characterization, it is possible to obtain tight conservative bounds on RTOS overhead during task response time analysis.

Columns two and three in Table 2-2 show that the I-cache improved the performance of some of the RTOS functions. This will also be considered in the following section.

### 5.3. Single and networked ECU analysis

Single ECU analysis determines response times, and consequently schedulability for all tasks running on the ECU. It builds upon single-process timing (Section 5.1) and RTOS timing results (Section 5.2). On top of that, it considers task-activation patterns and the scheduling strategy.

Our example system consists of three periodically activated tasks without data-dependencies. The tasks are scheduled by a fixed-priority scheduler and each task's deadline is equal to its period. Such a system can be analyzed using the static-priority preemptive analysis which was developed by Liu and Layland [10] and extended in [16] to account for exclusive-resource access arbitration. Additionally, we have to consider the RTOS overhead as explained in the previous section. A snapshot of the possible behavior of our system is shown in Figure 2-3.

During analysis, we can account for the high priority level of the **X act** and **Activate Task pre** functions by treating them as independent periodic tasks at a very high priority level. The **X term** function corresponding to the current task will not influence the task response time. However, if a higher priority task interrupts a running task then the preemption time includes also the execution time of the **X term** function that corresponds to the preempting task. **Task pre** is added to the process core execution time, and **Terminate Task pre** and **Terminate Task post** to the execution time of the X term function. We extend the analysis of [10] to capture this behavior and obtain the worst case task response time.

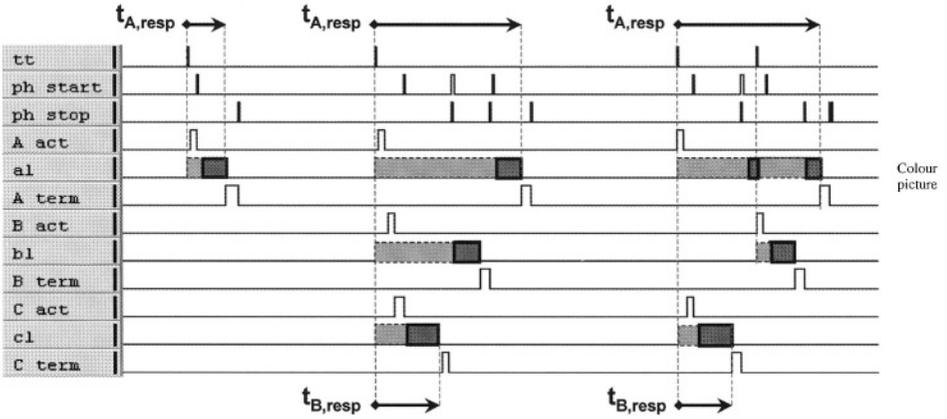


Figure 2-3. Sample preemption patterns and resulting response times for three tasks, including RTOS overhead.

$$\begin{aligned}
 R_i &= C_i + C_{j, Overhead} + \sum_{j \in HP(i)} (C_j + C_{j, Overhead}) \times \left\lceil \frac{R_i}{T_j} \right\rceil \\
 &+ \sum_k (C_{ActTask} + C_{ActTask, Overhead}) \times \left\lceil \frac{R_i}{T_k} \right\rceil \leq T_i
 \end{aligned}$$

This equation captures the worst case by assuming that all tasks are simultaneously activated, so each task experiences the maximum possible (worst case) number of preemptions by higher priority functions.  $C_i$  is the core execution time of task  $i$  from Section 5.1,  $R_i$  is the task response time and  $T_i$  its period. The different *Overhead* terms contain the various RTOS influences as explained above.  $HP(i)$  is the set of higher priority tasks with respect to task  $i$ . The equation above holds only if  $R_i \leq T_i$ , i.e. the deadline of each task has to be at the end of its period.

We still need to consider the effect of caches. Preemptive multitasking can influence the instruction cache behavior in two ways: (a) preemptions might overwrite cache lines resulting in additional cache misses after the preempted task resumes execution; and (b) repeatedly executed tasks might ‘find’ valid cache lines from previous task executions, resulting in additional cache hits. Both influences cannot be observed during individual process analysis but have to be considered during scheduling analysis.

Effect (a) is weak since for our controller code, which does not contain loops, the cache does not improve the execution time of individual processes in the first place (Section 5.1). Effect (b) cannot be observed for a realistic set of processes, since due to the limited I-cache size of 1 k each cache line is used by several processes. The CPU runs at 40 MHz and the word length is either 16 bits or 32 bits. The time it takes to read the complete cache is between 6.4  $\mu$ s and 12.8  $\mu$ s. However, the task periods are in the multi-ms

range. Therefore, no task will ever find a previously executed instruction in the cache, since other tasks will have filled the cache completely between two executions. However, Table 2-2 indicates that RTOS functions benefit from the cache, both because they may contain loops, and because they can be called multiple times shortly after each other, and their code thus is not replaced in-between calls.

Again, we used the setup described in Section 5.1 to perform several experiments to be checked against our analysis. We ran experiments executing the code from the external memory, both with enabled and disabled cache (see also Figure 2-3). The results are shown in Table 2-3. Task C has the highest, task A the lowest priority. The first and second columns show the worst case response times obtained by applying the above formula. In column one, worst case RTOS overhead with cache from Table 2-2 was used, in column two, worst case RTOS overhead without cache. The third and fourth columns show a range of measured response times during test-runs with and without cache.

As can be seen, even in the simple 3-task system presented here, there is a large response time variation between measurements. The size of the response time intervals becomes larger with decreasing task priority (due to more potential interrupts by higher priority tasks), but even the highest priority task experiences response time jitter due to the varying number of interrupts by RTOS functions running at even higher priority.

Our calculated worst case response times conservatively bound the observed behavior. The calculated worst case response times are only about 6% higher than the highest measured values, indicating the good tightness analysis can achieve if effects from scheduling, RTOS overhead, single process execution times and caches are all considered. Results would be even better with a more detailed RTOS model.

Another observation is that the cache and memory architecture is not well-adjusted for our target applications. We are currently looking into a TriCore version with 16 k 2-way associative cache for the production ECU. However, due to the characteristics of engine control code, we do not expect that single processes will benefit from a different cache architecture. Likewise, 16 k of cache are still by far too small to avoid complete replacement of process code between two activations of the same process.

At this point, we are still confined to single ECU applications.

Table 2-3. Task response times.

Task	Calculated worst-case response times (w/ cache)	Calculated worst-case response times (w/o cache)	Measured response time (w/ cache)	Measured response time (w/o cache)
A	243.8 $\mu$ s	260.6 $\mu$ s	[61.1; 231.0] $\mu$ s	[65.3; 247.3] $\mu$ s
B	174 $\mu$ s	185.0 $\mu$ s	[61.6; 164.5] $\mu$ s	[66.1; 178.9] $\mu$ s
C	103.2 $\mu$ s	107.8 $\mu$ s	[76.8; 95.9] $\mu$ s	[80.4; 102.4] $\mu$ s

Heterogeneous distributed architectures require more complex analysis techniques, such as developed in [13–15]. Based on the calculated response times, it is possible to derive event models which can be used to couple analysis techniques for single ECUs and busses.

## 6. CONCLUSION

In this chapter we presented a methodology for certifiable integration of automotive software components, which focuses on the exchange of performance information while IP is protected. After considering conventions for RTOS configuration, process communication and memory budgeting, we focused on an unsolved core integration issue, namely performance validation, in particular the validation of timing. We presented methods for performance characterization of software functions and the RTOS which also consider caches. We expect that ultimately each software component provider will have to characterize its components appropriately. First promising steps towards characterization of RTOS primitives have been taken by several RTOS vendors. Based on the characterization of single process and RTOS primitive timing we then showed how timing analysis of a single ECU can be performed by any integrator using appropriate scheduling analysis techniques, again considering caches.

We presented ongoing work on performance analysis of engine control software. This work can be extended to networked ECUs using the techniques from [14, 15]. In order to improve tightness of analysis, process and system contexts can additionally be considered [7].

## REFERENCES

1. G. Buttazzo. *Real-Time Computing Systems – Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, 2002.
2. ETAS. *ERCOS/ECU Automotive Real-Time Operating System*. [http://www.etas.info/html/products/ec/ercosek/en\\_products\\_ec\\_ercosek\\_index.php](http://www.etas.info/html/products/ec/ercosek/en_products_ec_ercosek_index.php).
3. ETAS. *ESCAPE Reference Guide*. [http://www.etas.info/download/ec\\_ercosek\\_rg\\_escape\\_en.pdf](http://www.etas.info/download/ec_ercosek_rg_escape_en.pdf).
4. C. Ferdinand and R. Wilhelm. “Efficient and Precise Cache Behavior Prediction for Real-Time Systems.” *Journal of Real-Time Systems, Special Issue on Timing Analysis and Validation for Real-Time Systems*, pp. 131–181, November 1999.
5. M. Hill. *DINERO III Cache Simulator: Source Code, Libraries and Documentation*. [www.ece.cmu.edu/ece548/tools/dinero/src/](http://www.ece.cmu.edu/ece548/tools/dinero/src/), 1998.
6. ISO. “TR 15504 Information Technology – Software Process Assessment ‘Spice’.” *Technical Report*, ISO IEC, 1998.
7. M. Jersak, K. Richter, R. Henia, R. Ernst, and F. Slomka. “Transformation of SDL Specifications for System-level Timing Analysis.” In *Tenth International Symposium on Hardware/Software Codesign (CODES’02)*, Estes Park, Colorado, USA, May 2002.
8. J. Lemieux. *Programming in the OSEK/VDX Environment*. CMP Books, 2001.
9. Y. S. Li and S. Malik. *Performance Analysis of Real-Time Embedded Software*. Kluwer Academic Publishers, 1999.

10. C. L. Liu and J. W. Layland. "Scheduling Algorithm for Multiprogramming in a Hard-Real-Time Environment." *Journal of the ACM*, Vol. 20, 1973.
11. LiveDevices Inc. *Realogy Real-Time Architect Overview*. <http://www.livedevices.com/realtime.shtml>.
12. OSEK/VXD. *OIL: OSEK Implementation Language*, version 2.3 edition, September 2001.
13. T. Pop, P. Eles, and Z. Peng. "Holistic Scheduling and Analysis of Mixed Time/Event-Triggered Distributed Embedded Systems." In *Tenth International Symposium on Hardware/Software Codesign (CODES'02)*, Estes Park, Colorado, USA, May 2002.
14. K. Richter and R. Ernst. "Event Model Interfaces for Heterogeneous System Analysis." In *Proceedings of Design, Automation and Test in Europe Conference (DATE'02)*, Paris, France, March 2002.
15. K. Richter, D. Ziegenbein, M. Jersak, and R. Ernst. "Model Composition for Scheduling Analysis in Platform Design." In *Proceedings of 39th Design Automation Conference*, New Orleans, USA, June 2002.
16. L. Sha, R. Rajkumar, and J. P. Lehoczky. "Priority Inheritance Protocols: An Approach to Real-Time Synchronization." *IEEE Transactions on Computers*, Vol. 39, No. 9, September 1990.
17. K. Tindell, H. Kopetz, F. Wolf, and R. Ernst. "Safe automotive Software Development." In *Proceedings of Design, Automation and Test in Europe (DATE'03)*, Munich, Germany, March 2003.
18. F. Wolf. *Behavioral Intervals in Embedded Software*. Kluwer Academic Publishers, 2002.
19. F. Wolf, J. Kruse, and R. Ernst. "Segment-Wise Timing and Power Measurement in Software Emulation." In *Proceedings of IEEE/ACM Design, Automation and Test in Europe Conference (DATE'01)*, Designers' Forum, Munich, Germany, March 2001.

## Chapter 3

# LIGHTWEIGHT IMPLEMENTATION OF THE POSIX THREADS API FOR AN ON-CHIP MIPS MULTIPROCESSOR WITH VCI INTERCONNECT

*A Micro-Kernel with standard API for SoCs with  
generic interconnects*

Frédéric Pétrou, Pascal Gomez and Denis Hommais

*ASIM Department of the LIP6, Université Pierre et Marie Curie, Paris, France*

**Abstract.** This paper relates our experience in designing from scratch a multi-threaded kernel for a MIPS *R3000* on-chip multiprocessor. We briefly present the target architecture build around an interconnect compliant with the Virtual Chip Interconnect (VCI), and the CPU characteristics. Then we focus on the implementation of part of the POSIX 1003.1b and 1003.1c standards. We conclude this case study by simulation results obtained by cycle true simulation of an MJPEG video decoder application on the multiprocessor, using several scheduler organizations and architectural parameters.

**Key words:** micro-kernel, virtual chip interconnect, POSIX threads, multiprocessor

## 1. INTRODUCTION

Applications targeted to *SoC* implementations are often specified as a set of concurrent tasks exchanging data. Actual co-design implementations of such specifications require a multi-threaded kernel to execute the parts of the application that has been mapped to software. As the complexity of applications grows, more computational power but also more programmable platforms are useful. In that situation, on-chip multiprocessors with several general purpose processors are emerging in the industry, either for low-end applications such as audio encoders/decoders, or for high end applications such as video decoders or network processors. Compared to multiprocessor computers, such integrated architectures feature a shared memory access with low latency and potentially very high throughput, since the number of wires on chip can be much greater than on a printed card board.

This paper relates our experience in implementing from scratch the POSIX thread API for an on-chip MIPS *R3000* multiprocessor architecture. We choose to implement the POSIX thread API for several reasons:

- It is standardized, and is *de facto* available on many existing computer systems;
- It is well known, taught in universities and many applications make use of it;
- The 1003.1c defines no more than 5 objects, allowing to have a compact implementation.

All these facts make the development of a bare kernel easier, because it relies on a hopefully well behaved standard and API, and allows direct functional comparison of the same application code on a commercial host and on our multiprocessor platform.

The main contribution of this paper is to relate the difficult points in developing a multiprocessor kernel general enough to support the POSIX API on top of a generic interconnect. The problems that we have encountered, such as memory consistency, compiler optimization avoidance, . . . , are outlined. We also want this kernel to support several types of organization: Symmetric using a single scheduler, Distributed using one scheduler per processor, Distributed with centralized synchronization, with or without task migration, etc, in order to compare them experimentally.

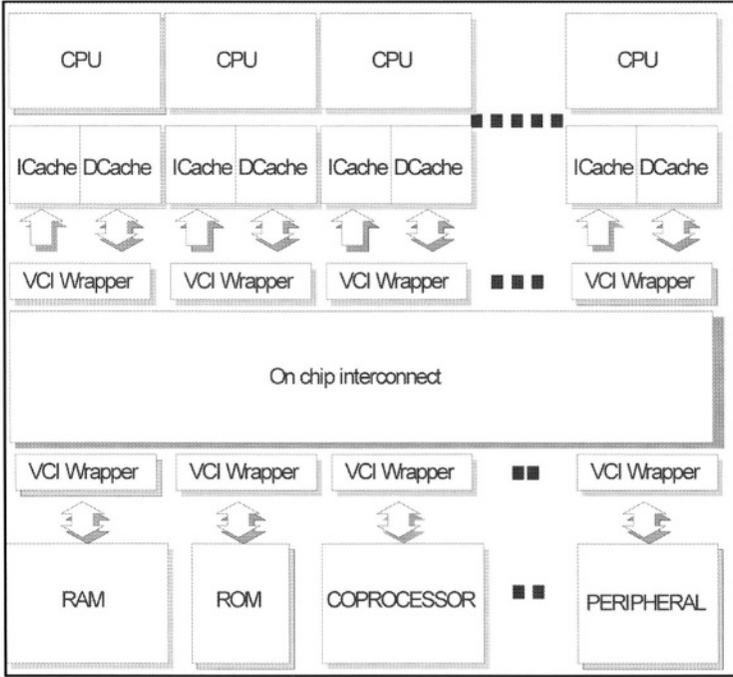
## 2. TARGET ARCHITECTURE AND BASIC ARCHITECTURAL NEEDS

The general architecture that we target is presented on the Figure 3-1. It makes use of one or more MIPS *R3000* as CPU, and a Virtual Chip Interconnect [1] compliant interconnect on the which are plugged memories and dedicated hardware when required. The MIPS CPU has been chosen because it is small and efficient, and also widely used in embedded applications [2].

It has the following properties:

- Two separated caches for instruction and data;
- Direct mapped caches, as for level one caches on usual computers;
- Write buffer with cache update on write hit and write through policy. Write back policy allows to minimize the memory traffic and allows to build bursts when updating memory, particularly useful for SD-RAM, but it is very complicated to ensure proper memory consistency [3];
- No memory management unit (MMU), logical addresses are physical addresses. Virtual memory isn't particularly useful on resource constrained hardware because the total memory is fixed at design time. Page protection is also not an issue in current *SoC* implementations. Finally, the multi-threaded nature of the software makes it natural to share the physical address space.

The interconnect is VCI compliant. This ensures that our kernel can be used with any interconnect that has VCI wrappers. This means:



Colour picture

Figure 3-1. General VCI based SOC architecture.

- It is basically a shared memory architecture, since all addresses that go through a VCI interface are seen alike by the all the targets;
- The actual interconnect is not known: only the services it provides are. VCI is basically a point to point data transfer protocol, and does not say anything about cache consistency or interrupt handling, cached or uncached memory space and so on.

On the *R3000*, we do not distinguish between *user* and *kernel* modes, and the whole application runs in *kernel* mode. This allows to spare cycles when a) accessing the uncached memory space (in *kernel* space in the *R3000*) that contains the semaphore engine and hardware module, b) using privileged instructions, to set the registers of coprocessor 0, necessary mainly to mask/unmask the interrupts and use the processor identifier. The number of cycles spared is at least 300, to save and restore the context and analyze the cause of the call. This is to be compared with the execution times of the kernel functions given in Table 3-2.

The architecture needs are the following:

**Protected access to shared data.**

This is done using a spin lock, whose implementation depends on the architecture. A spin lock is acquired using the `pthread_spin_lock` function,

and released using `pthread_spin_unlock`. Our implementation assumes that there is a very basic binary semaphore engine in uncached space such that reading a slot in it returns the value of the slots and sets it to 1. Writing in a slot sets it to 0. Other strategies can make use of the read modify write opcode of the VCI standard, but this is less efficient and requires that each target is capable of locking its own access to a given initiator, thus requiring more resources per target.

### **Cache coherency.**

If the interconnect is a shared bus, the use of a snoopy cache is sufficient to ensure cache coherence. This has the great advantage of avoiding any processor to memory traffic. If the interconnect is VCI compliant (either bus or network), an access to a shared variable requires either to flush the cache line that contains this variable to obtain a fresh copy of the memory or to have such variables placed in uncached address space. This is due to the fact that VCI does not allow the building of snoopy caches, because the VCI wrapper would have to know both the cache directory entries and be aware of all the traffic, and that is simply not possible for a generic interconnect. Both solutions are not very satisfactory as the generated traffic eats-up bandwidth and leads to higher power consumption, particularly costly for on-chip applications. However, this is the price to pay for interconnect genericity (including on-chip networks). In any case, synchronization for the access to shared data is mandatory. Using caches is meaningful even for shared data only used once because we benefit from the read burst transfers on the interconnect.

### **Processor identification.**

The CPUs must have an internal register allowing their identification within the system. Each CPU is assigned a number at boot time, as some startup actions should be done only once, such as clearing the blank static storage area (`.bss`) and creating the scheduler structure. Also, the kernel sometimes needs to know which processor it runs on to access processor specific data.

Compared to other initiatives, [4] for example, our kernel is designed for multiprocessor hardware. We target a lightweight distributed scheduler with shared data objects, each having its own lock, and task migration.

## **3. OVERVIEW OF THE *PTHREAD* SPECIFICATIONS**

The main kernel objects are the threads and the scheduler. A thread is created by a call to:

```
int pthread_create(pthread_t *thread, pthread_attr_t *attr,
    void *(*start)(void *), void *arg);
```

This creates and executes the thread whose behavior is the `start` function

called with `arg` as argument. The `attr` structure contains thread attributes, such as stack size, stack address and scheduling policies. Such attributes are particularly useful when dealing with embedded systems or *SoCs*, in the which the memory map is not standardized. The value returned in the `thread` pointer is a unique identifier for the thread.

A thread can be in one of 5 states, as illustrated by the Figure 3-2. Changing state is usually done using some *pthread* function on a shared object. Exceptions to this rule is going from `RUNNABLE` to `RUN`, which is done by the scheduler using a given policy, and backward from `RUN` to `RUNNABLE` using `sched_yield`. This function does not belong to the POSIX thread specifications in the which there is no way to voluntarily release the processor for a thread. This function is usually called in a timer interrupt handler for time sharing.

In our implementation, the thread identifier is a pointer to the thread structure. Note that POSIX, unlike more specialized API such as OSEK/VDX [5], does not provide a mean for static thread creation. It is a shortcoming because most embedded applications do not need dynamic task creation. A thread structure basically contains the context of execution of a thread and pointers to other threads.

The scheduler manages 5 lists of threads. It may be shared by all processors (Symmetrical Multi-Processor), or exist as such on every processor (Distributed). The access to the scheduler must be performed in critical section, and under the protection of a lock. However, this lock can be taken

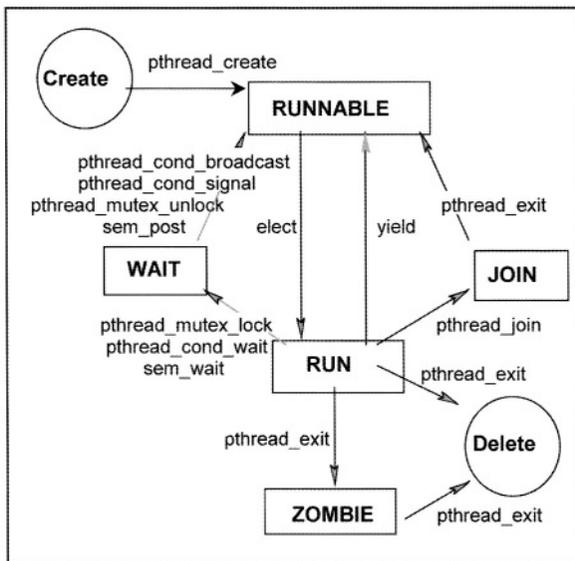


Figure 3-2. Pthread states.

for only very few instructions if the other shared objects have their own locks, which allows for greater parallelism when the hardware allows it.

Other implemented objects are the *spin locks*, *mutex*, *conditions*, and *semaphores*. *Spin locks* are the low level test and set accesses, that usually perform active polling. *Mutex* are used to sequentialize access to shared data by suspending a task as the access to the code that could modify the data is denied. *Conditions* are used to voluntarily suspend a thread waiting for a condition to become true. *Semaphores* are mainly useful when dealing with hardware, because `sem_post` is the only function that can be called in interruption handlers.

## 4. IMPLEMENTATION

Our implementation is a complete redesign that doesn't make use of any existing code. Most of it is written in C. This C is not particularly portable because it makes use of physical addresses to access the semaphore engine, the terminal, and so on. Some assembly is necessary for the deeply processor dependent actions: access to the MIPS coprocessor 0 registers, access to processor registers for context saving and restoring, interruption handling and cache line flushing.

To avoid a *big lock* on the scheduler, every *mutex* and *semaphore* has its own lock. This ensures that the scheduler lock will be actually acquired only if a thread state changes, and this will minimize scheduler locking time, usually around 10 instructions, providing better parallelism. In the following, we assume that access to all shared variables are a fresh local copy of the memory in the cache.

### 4.1. Booting sequence

Algorithm 3-1 describes the boot sequence of the multiprocessor. The identification of the processors is determined in a pseudo-random manner. For example, if the interconnect is a bus, the priorities on the bus will define this order. It shall be noted that there is no need to know how many processors are booting. This remains true for the whole system implementation.

Two implementation points are worth to be seen. A weak memory consistency model [6] is sufficient to access the shared variable `proc_id`, since it is updated after a synchronization point. This model is indeed sufficient for POSIX threads applications. The `scheduler_created` variable must be declared with the `volatile` type qualifier to ensure that the compiler will not optimize this seemingly infinite loop.

The `main` thread is executed by processor 0, and, if the application is multi-thread, it will create new threads. When available, these threads will be executed on any processor waiting for an available thread. Here the `execute()` function will run a new thread without saving the current context,

since the program will never come back at that point. The thread to execute is chosen by the *Elect()* function. Currently, we have only implemented a FIFO election algorithm.

*Algorithm 3-1. Booting sequence.*

```

Definition and statically setting shared variables to 0
scheduler_created  $\Rightarrow$  0           No scheduler exists
proc_id  $\Rightarrow$  0                     First processor is numbered 0
mask interruptions                 Done by all processors
Self numbering of the processors
spin_lock(lock), set_proc_id(proc_id++), spin_unlock(lock)
set local stack for currently running processor
if get_proc_id() = 0
    clear .bss section
    scheduler and main thread creation
    scheduler_created  $\Rightarrow$  1           Indicates scheduler creation
    enable interruptions
    jump to the main function
else
    Wait until scheduler creation by processor 0
    while scheduler_created = 0 endwhile
    Acquire the scheduler lock to execute a thread
    spin_lock(scheduler)
    execute(elect())
endif

```

## 4.2. Context switch

For the *R3000*, a context switch saves the current value of the CPU registers into the context variable of the thread that is currently executing and sets the values of the CPU registers to the value of the context variable of the new thread to execute. The tricky part is that the return address of the function is a register of the context. Therefore, restoring a context sends the program back where the context was saved, not to the current caller of the context switching routine. An important question is to define the registers and/or variables that belong to the context. This is architecture and kernel dependent: For example, a field of current compiler research concerns the use of a scratch pad memory instead of a data cache [7] in embedded multiprocessors. Assuming that the kernel allows the threads to be preempted, the main memory must be updated before the same thread is executed again. If this is not the case, the thread may run on an other processor and used stalled data from memory.

On a *Sparc* processor, the kernel must also define what windows, if not all, are to be saved/restored by context switches, and this may have an important impact on performance/power consumption.

### 4.3. CPU idle loop

When no tasks are `RUNNABLE`, the CPU runs some kind of idle loop. Current processors could benefit from this to enter a low power state. However, waking up from such a state is in the order of 100 ms [8] and its use would therefore be very application dependent.

An other, lower latency solution, would be to launch an *idle* thread whose role is to infinitely call the `sched_yield` function. There must be one such thread per CPU, because it is possible that all CPUs are waiting for some coprocessor to complete there work. These threads should not be made `RUN` as long as other threads exist in the `RUNNABLE` list. This strategy is elegant in theory, but it uses as many threads resources as processors and needs a specific scheduling policy for them.

Our current choice is to use a more *ad-hoc* solution, in the which all idle CPUs enter the same idle loop, that is described in Algorithm 3-2. This routine is called only once the scheduler lock has been acquired and the interruptions globally masked. We use the *save\_context* function to save the current thread context, and update the register that contains the function return address to have it point to the end of the function. This action is necessary to avoid going through the idle loop again when the *restore\_context* function is called. Once done, the current thread may be woken up again on an other processor, and therefore we may not continue to use the thread stack, since this would modify the local data area of the thread. This justifies the use of a dedicated stack (one for all processors in centralized scheduling and one per processor for distributed scheduling). The registers of the CPU can be modified, since they are not anymore belonging to the thread context.

The `wakeup` variable is a volatile field of the scheduler. It is needed to inform this idle loop that a thread has been made `RUNNABLE`. Each function that awakes (or creates) a thread decrements the variable. The `go` local variable enable each CPU to register for getting out of this loop. When a *pthread* function releases a *mutex*, signals a *condition* or posts a *semaphore*, the CPU with the correct `go` value is allowed to run the awoken thread. This takes places after having released the semaphore lock to allow the other functions to change the threads states, and also after having enable the interruptions in order for the hardware to notify the end of a computation.

*Algorithm 3-2. CPU Idle Loop.*

```

if current_thread exists
  save_context (current_thread)
  current.return_addr_register ← end of function address
  local_stack ← scheduler stack
  repeat
    go ← wakeup++
    spin_unlock(lock), global_interrupt_unmask
  while wakeup > go endwhile

```

```

    global_interrupt_mask, spin_lock(lock)
    thread ← elect()
    until thread exists
    restore_context(thread)
end of function

```

### 5. EXPERIMENTAL SETUP

The experiments detailed here use two applications. The first one is a multimedia application, a decoder of a flow of JPEG images (known as Motion JPEG), whose task graph is presented Figure 3-3. The second application is made of couple of tasks exchanging data through FIFO, and we call it COMM. COMM is a synthetic application in the which scheduler access occurs 10 times more often for a given time frame that in the MJPEG application. This allows to check the behavior of the kernel on a very system intensive applications. COMM spends from 56% to 79% of its time in kernel calls, depending on the number of processors.

The architecture the applications run on is presented Figure 3-4, but the number of processors vary from one experiment to another. This architecture is simulated using the CASS [9] cycle true simulator whose models are compatible with SystemC. The application code is cross-compiled and linked with the kernel using the GNU tools. Non disturbing profiling is performed to obtain the figures of merits of each kernel implementation.

We now want to test several implementations of our kernel. The literature defines several types of scheduler organization. We review here the three that we have retained for implementation and outline their differences.

- Symmetric Multiprocessor (SMP). There is a unique scheduler shared by all the processors and protected by a lock. The threads can run on any

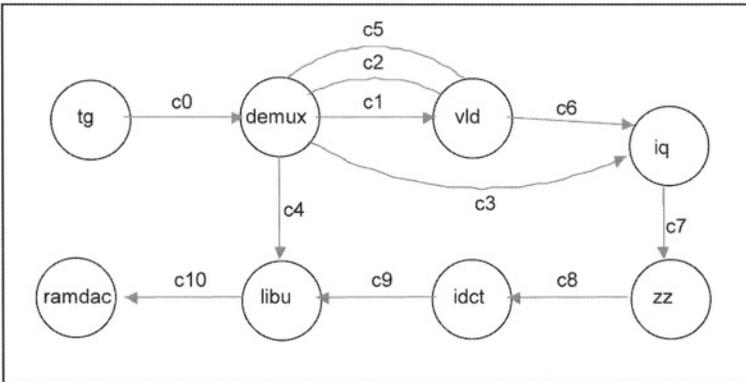


Figure 3-3. Motion JPEG application task graph.

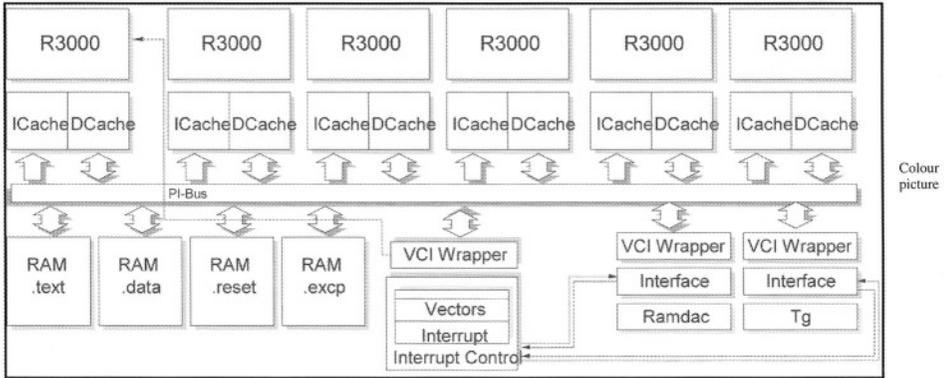


Figure 3-4. Target architecture simulated using cycle true models.

- processor, and thus migrate. This allows to theoretically evenly distribute the load on all CPUs at the cost of more cache misses;
- Centralized Non SMP (NON\_SMP\_CS). There is a unique scheduler shared by all processors and protected by a lock. Every thread is assigned to a given processor and can run only on it. This avoid task migration at the cost of less efficient use of CPUs cycles (more time spend in the CPU idle loop);
- Distributed Non SMP (NON\_SMP\_DS). There are as many schedulers as processors, and as many locks as schedulers. Every thread is assigned to a given processor and can run only on it. This allows a better parallelism by replicating the scheduler that is a key resource.

In both non SMP strategies, load balancing is performed so as to optimize CPU usage, with a per task load measured on a uniprocessor setup.

In all cases, the *spin locks*, *mutex*, *conditions* and *semaphores* are shared, and there is a *spin lock* per *mutex* and *semaphore*.

Our experimentation tries to give a quantitative answer to the choice of scheduler organization.

## 6. RESULTS

The Table 3-1 indicate the code size for the three versions of the scheduler. The NON\_SMP\_DS strategy grows dynamically of around 80 bytes per processor, whereas there is no change for the other ones.

The Figure 3-5 plots the execution time of the MJPEG application for 48 small pictures. The SMP and NON\_SMP\_CS approaches are more than 10% faster than the NON\_SMP\_DS one. The Figure 3-6 shows the time spent in the CPU Idle Loop. We see that the SMP kernel spends more than an order of magnitude less time in the Idle loops than the other strategies. This outline

Table 3-1. Kernel code size (in bytes).

Organization	SMP	NON_SMP_CS	NON_SMP_DS
Code size	7556	9704	10192

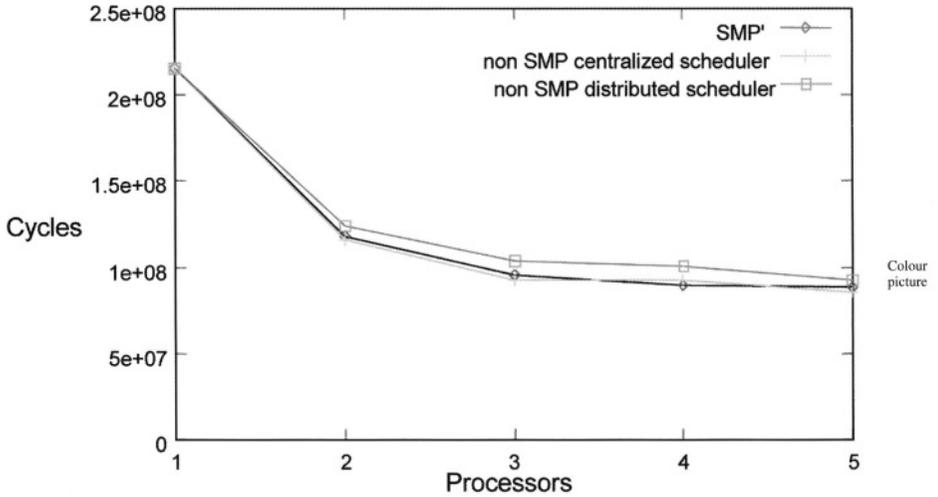


Figure 3-5. Execution times of the Motion JPEG application.

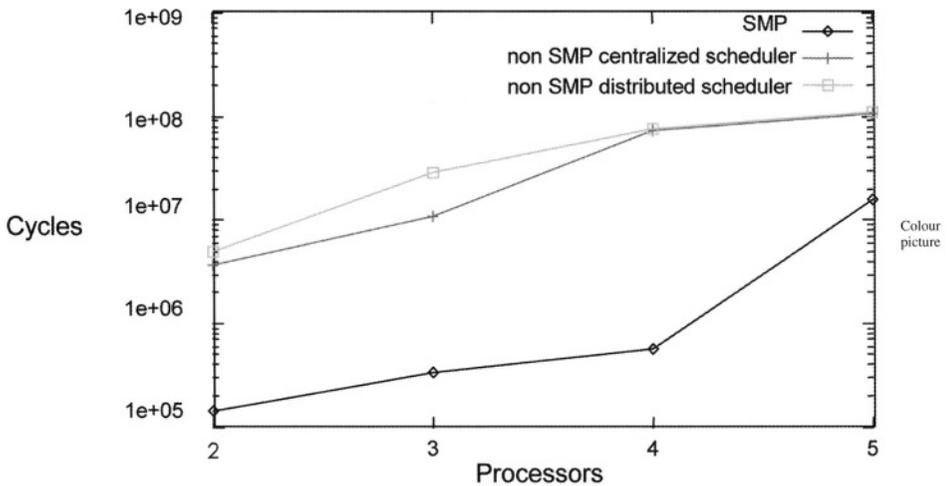


Figure 3-6. Total time spend in the CPUs Idle loops.

its capacity to use the CPU cycles more efficiently. However, task migration has a high cost in terms of cache misses, and therefore, the final cycle count is comparable to the other ones. It shall be noted that the SMP interest might become less clear if shared memory latency access increases too much. The NON\_SMP\_DS strategy is more complicated from an implementation point of view. This is the reason why it is less efficient in this case.

Our second application does not exchange data between processors, and the performances obtained are plotted on the Figure 3-7.

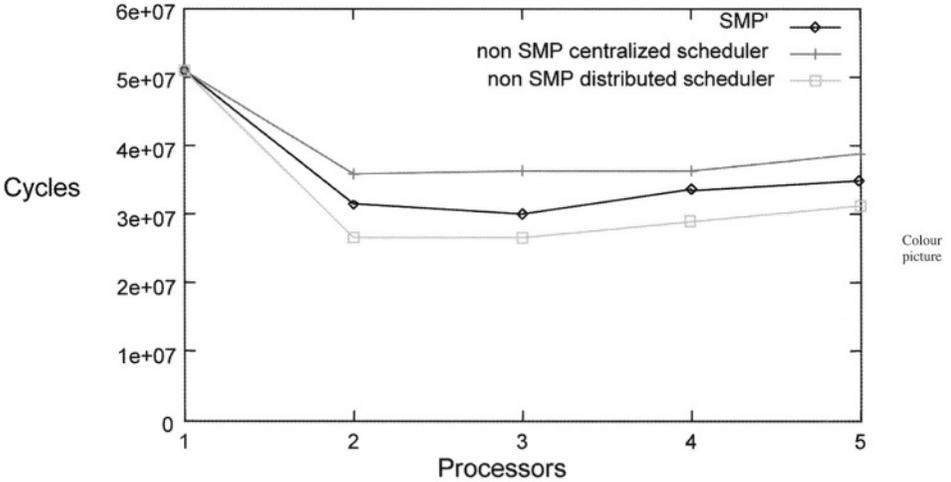


Figure 3-7. Execution times of the COMM application.

The benefit of having one scheduler per processor is very sensitive here, and visible on the NON\_SMP\_DS results. The only resource shared here is the bus, and since the caches are big enough to contain most of the application data, the application uses the processors at about full power.

The Table 3-2 shows the number of cycles necessary to perform the main POSIX function using our SMP kernel on the target architecture.

Theses values have been obtained by performing the mean for over 1000 calls. For the interrupts ( $\alpha$ ), all processors were interrupted simultaneously.

### 7. CONCLUSION AND FUTURE TRENDS

This paper relates our experience in implementing the POSIX threads for a MIPS multiprocessor based around a VCI interconnect. Compared to kernels for on-board multiprocessors, that have been extensively studied in the past, our setup uses a generic interconnect for which the exact interconnect is not known, the CPUs use physical addresses, and the latency to access shared

Table 3-2. Performance of the main kernel functions (in number of cycles).

Operation	Number of processors			
	1	2	3	4
Context switch	172	187	263	351
Mutex lock (acquired)	36	56	61	74
Mutex unlock	30	30	31	34
Mutex lock (suspended)	117	123	258	366
Mutex unlock (awakes a thread)	N/A	191	198	218
Thread creation	667	738	823	1085
Thread exit	98	117	142	230
Semaphore acquisition	36	48	74	76
Semaphore acquisition	36	50	78	130
Interrupt handler ( $\alpha$ )	200	430	1100	1900

memory is much lower. The implementation is a bit tricky, but quite compact and efficient. Our experimentations have shown that a POSIX compliant SMP kernel allowing task migration is an acceptable solution in terms of generality, performance and memory footprint for *SoC*.

The main problem due to the introduction of networks on chip is the increasing memory access latency. One of our goal in the short term is to investigate the use of latency hiding techniques for these networks. Our next experiment concerns the use of a dedicated hardware for semaphore and pollable variables that would queue the acquiring requests and put to sleep the requesting processors until a change occurs to the variable.

This can be effectively supported by the VCI interconnect, by the mean of its request/acknowledge handshake. In that case, the implementation of *pthread\_spin\_lock* could suspend the calling task. This could be efficiently taken care of if the processors that run the kernel are processors with multiple hardware contexts, as introduced in [10].

The SMP version of this kernel, and a minimal C library, is part of the Disydent tool suite available under GPL at [www-asim.lip6.fr/disydent](http://www-asim.lip6.fr/disydent).

## REFERENCES

1. VSI Alliance. *Virtual Component Interface Standard (OCB 2 2.0)*, August 2000.
2. T. R. Halfhill. "Embedded Market Breaks New Ground." *Microprocessor Report*, January 2000.
3. J. Archibald and J.-L. Baer. "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model." *ACM Transactions on Computer Systems*, Vol. 4, No. 4, pp. 273–298, 1986.
4. T. P. Baker, F. Mueller, and V. Rustagi. "Experience with a Prototype of the POSIX Minimal

- Tealtime System Profile.” In *Proceedings of 11th IEEE Workshop on Real-Time Operating Systems and Software*, pp. 12–16, Seattle, WA, USA, 1994.
5. OSEK/VDX. *OSEK/VDX Operating System Specification 2.2*, September 2001. <http://www.osek-idx.org>.
  6. A. S. Tanenbaum. *Distributed Operating Systems*, Chapter 6.3, pp. 315–333. Prentice Hall, 1995.
  7. M. Kandemir and A. Choudhary. “Compiler-Directed Scratch Pad Memory Hierarchy Design and Management.” In *Design Automation Conference*, pp. 628–633, New Orleans, LA, June 2002.
  8. J. Montanaro et al. “A 160mhz 32b 0.5w CMOS RISC Microprocessor.” In *ISSCC Digest of Technical Papers*, pp. 214–215, February 1996.
  9. F. Pétrot, D. Hommais, and A. Greiner. “Cycle Precise Core Based Hardware/Software System Simulation with Predictable Event Propagation.” In *Proceedings of the 23rd Euromicro Conference*, pp. 182–187, Budapest, Hungary, September 1997. IEEE.
  10. Jr. R. H. Halstead and T. Fujita. “MASA: A Multithreaded Processor Architecture for Parallel Symbolic Computing.” In *15th Annual International Symposium on Computer Architecture*, pp. 443–451, June 1988.

## Chapter 4

# DETECTING SOFT ERRORS BY A PURELY SOFTWARE APPROACH: METHOD, TOOLS AND EXPERIMENTAL RESULTS

B. Nicolescu and R. Velazco

*TIMA Laboratory, "Circuit Qualification" Research Group, 46, Av. Félix Viallet, 38031, Grenoble, France*

**Abstract.** A software technique allowing soft errors detection occurring in processor-based digital architectures is described. The detection mechanism is based on a set of rules allowing the transformation of the target application into a new one, having the same functionality but being able to identify bit-flips arising in memory areas as well as those perturbing the processor's internal registers. Experimental results, issued from both fault injection sessions and preliminary radiation test campaigns performed in a complex digital signal processor; provide objective figures about the efficiency of the proposed error detection technique.

**Key words:** bit flips, SEU, SET, detection efficiency, error rate

### 1. INTRODUCTION

Technological progress achieved in the microelectronics technology has as a consequence the increasing sensitivity to effects of the environment (i.e. radiation, EMC). Particularly, processors operating in space environment are subject to different radiation phenomena, whose effects can be permanent or transient [1]. This paper strictly focuses on the transient effects, also called SEUs (Single Event Upsets) occurring as the consequence of the impact of charged particles with sensitive areas of integrated circuits. The SEUs are responsible for the modification of memory cells content, with consequences ranging from erroneous results to system control problem. The consequences of the SEUs depend on both the nature of the perturbed information and the bit-flips occurrence instants.

For complex processor architectures, the sensitivity to SEUs is strongly related to the amount of internal memory cells (registers, internal memory). Moreover, it is expected that future deep submicron circuits operating at very high frequencies will be also subject to transient errors in combinational parts, as a result of the impact of a charged particle. This phenomenon, so-called SET (Single Event Transient) could constitute a serious source of errors not only for circuits operating in space, but also for digital equipment operating in the Earth's atmosphere at high altitudes (avionics) and even at ground level [2]. In the new areas where computer-based dependable systems are currently

being introduced, the cost (and hence the design and development time) is often a major concern, and the adoption of standard components (Commercial Off-The-Shelf or COTS products) is a common practice. As a result, for this class of applications software fault tolerance is a highly attractive solution, since it allows the implementation of dependable systems without incurring the high costs coming from designing custom hardware or using hardware redundancy. On the other side, relying on software techniques for obtaining dependability often means accepting some overhead in terms of increased code size and reduced performance. However, in many applications, memory and performance constraints are relatively loose, and the idea of trading off reliability and speed is often easily acceptable.

Several approaches have been proposed in the past to achieve fault tolerance (or just safety) by modifying only the software. The proposed methods can mainly be categorized in two groups: those proposing the replication of the program execution and the check of the results (i.e., Recovery Blocks [3] and N-Version Programming [4]) and those based on introducing some control code into the program (e.g., Algorithm Based Fault Tolerance (ABFT) [5], Assertions [6], Code Flow Checking [7], procedure duplication [8]). None of the mentioned approaches is at the same time *general* (in the sense that it can be used for any fault type and any application, no matter the algorithm it implements) and *automatic* (in the sense that it does not rely on the programmer's skills for its effective implementation). Hence, none of the above methods is enough complete and suitable for the implementation of low-cost safety-critical microprocessor-based systems.

To face the gap between the available methods and the industry requirements, we propose an error detection technique which is based on introducing data and code redundancy according to a set of transformation rules applied on high-level code. The set of rules is issued from a thorough analysis of the one described in [9]. In this paper, we report experimental results of SEU effects on an industrial software application, obtained by performing fault injection experiments in commercial microprocessors. In Section 2 the software detection rules are briefly presented. The main features of the used fault injection technique are summarized in Section 3. Experiments were performed by injecting faults in selected targets during a randomly selected clock cycle. Experimental results obtained through both software fault injection and radiation testing campaign are analyzed and discussed in Section 4. Finally, Section 5 presents concluding remarks and future work.

## 2. SOFTWARE BASED FAULT TOLERANCE

This section describes the investigated methodology to provide error detection capabilities through a purely software approach. Subsection 2.1 describes the software transformation rules, while subsection 2.2 proposes an automatic generation of the hardened programs.

## 2.1. Transformation rules

The studied approach exploits several code transformation rules. The rules are classified in three basic groups presented in the following.

### 2.1.1. Errors affecting data

This group of rules aims at detecting those faults affecting the data. The idea is to determine the interdependence relationships between the variables of the program and to classify them in two categories according to their purpose in the program:

- intermediary variables: they are used for the calculation of other variables;
- final variables: they do not take part in calculation of any other variable.

Once the variables relationships are drawn up, all the variables in the program are duplicated. For each operation carried out on an original variable, the operation is repeated for its replica, that we will call *duplicated variable*. Thus, the interdependence relationships between the duplicated variables are the same with those between the original variables. After each write operation on the *final variables*, a consistency check between the values of the two variables (*original* and *duplicated*) is introduced. An error is signaled if there is a difference between the value of the original variable and that of the duplicated variable.

The proposed rules are:

- Identification of the relationships between the variables;
- Classification of the variables according to their purpose in the program: *intermediary variable* and *final variable*;
- Every variable  $x$  must be duplicated: let  $x1$  and  $x2$  be the names of the two copies;
- Every operation performed on  $x$  must be performed on  $x1$  and  $x2$ ;
- After each write operation on the *final variables*, the two copies  $x1$  and  $x2$  must be checked for consistency, and an error detection procedure is activated if an inconsistency is detected.

Figure 4-1 illustrates the application of these rules to a simple instruction sequence consisting of two arithmetical operations performed on four variables (Figure 4-1a). The interdependence relationships between the variables are:  $a = f(b, c)$  and  $d = f(a = f(b, c), b)$ . In this case only  $d$  is considered as a *final variable* while  $a$ ,  $b$  and  $c$  are *intermediary variables*. Figure 4-1b shows the transformations issued from the set of rules presented.

### 2.1.2. Errors affecting basic instructions

This group of rules aims at detecting those faults modifying the code provoking the execution of incorrect jumps (for instance by modification of the

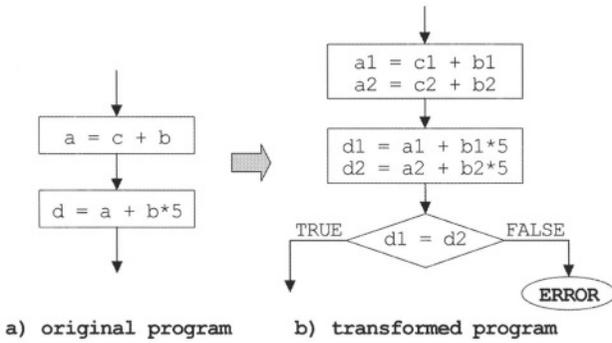


Figure 4-1. Transformations rules targeting errors affecting data.

operand of an existing jump, or by transforming an instruction into a jump instruction) producing thus a faulty execution flow. To detect this type of error, we associate to each basic block in the program a boolean flag called *status\_block*. This flag takes the value “0” if the basic block is active and “1” for the inactive state. At both the beginning and the end of each basic block, the value of the *status\_block* is incremented modulo 2. In this way the value of the *status\_block* is always “1” at the beginning of each basic block and “0” at the end of each block. If an error provokes a jump to the beginning of a wrong block (including an erroneous restarting of the currently executed block) the *status\_block* will take the value “0” for an active block and “1” for an inactive state block. This abnormal situation is detected at the first check performed on *gef* flag due to the inappropriate values taken by the control flow flags. The application of these rules is illustrated in Figure 4-2.

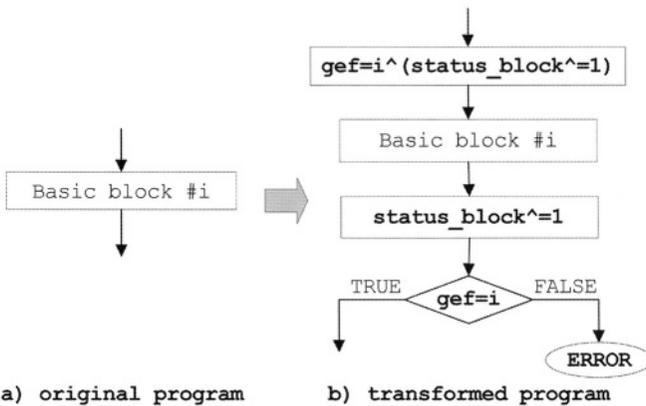


Figure 4-2. Transformations targeting errors affecting the basic instructions.

According to these modifications, the studied rules become the following:

- A boolean flag *status\_block* is associated to every basic block *i* in the code; “1” for the inactive state and “0” for the active state;
- An integer value *ki* is associated with every basic block *i* in the code;
- A global execution check flag (*gef*) variable is defined;
- A statement assigning to *gef* the value of  $(ki \& (status\_block = status\_block + 1) \bmod 2)$  is introduced at the beginning of every basic block *i*; a test on the value of *gef* is also introduced at the end of the basic block.

2.1.3. Errors affecting control instructions

These rules aim at detecting faults affecting the control instructions (branch instructions). According to the branching type, the rules are classified as:

- Rules addressing errors affecting the conditional control instructions (i.e. test instructions, loops);
- Rules addressing errors affecting the unconditional control instructions (i.e. calls and returns from procedures).

2.1.3.1. Rules targeting errors affecting the conditional control instructions

The principle is to recheck the evaluation of the condition and the resulting jump. If the value of the condition remains unchanged after re-evaluation, we assume that no error occurred. In the case of alteration of the condition an error procedure is called, as illustrated by the example in Figure 4-3.

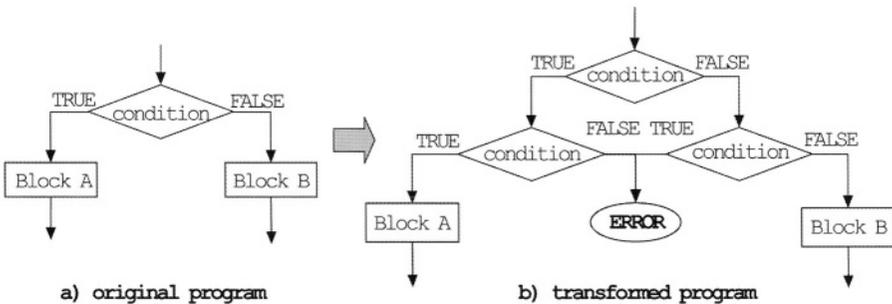


Figure 4-3. Transformations targeting errors affecting the conditional control instructions.

2.1.3.2. Rules targeting errors affecting the unconditional control instructions

The idea of these rules consists in the association of a control branch (called *ctrl\_branch*) flag to every procedure in the program. At the beginning of each procedure a value is assigned to *ctrl\_branch*. This flag is checked for consistency before and after any call to the procedure in order to detect possible errors affecting this type of instructions. The application of these rules is illustrated in Figure 4-4.

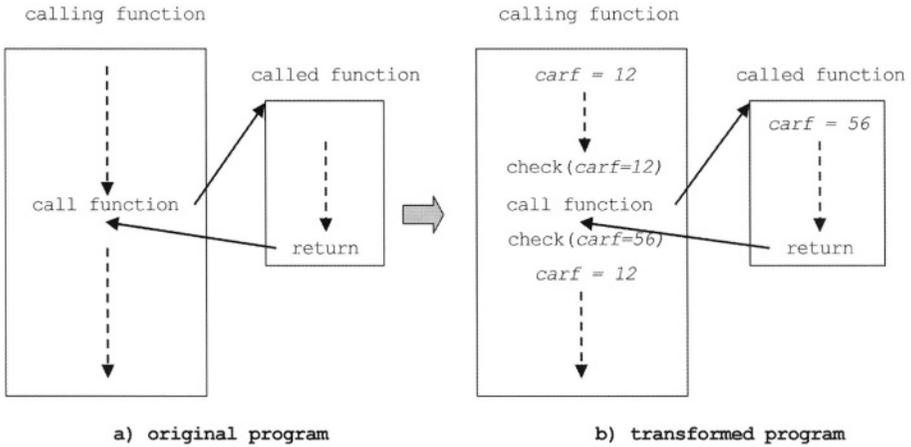


Figure 4-4. Transformations targeting errors affecting the unconditional control instructions.

In summary, the rules are defined as follows:

- For every test statement the test is repeated at the beginning of the target basic block of both the true and false clause. If the two versions of the test (the original and the newly introduced) produce different results, an error is signaled;
- A flag *ctrl\_branch* is defined in the program;
- An integer value *kj* is associated with any procedure *j* in the code;
- At the beginning of every procedure, the value *kj* is assigned to *ctrl\_branch*; a test on the value of *ctrl\_branch* is introduced before and after any call to the procedure.

#### 2.1.4. Transformation tool – C2C Translator

We built a prototype tool, called C2C Translator, able to automatically implement the above described transformation rules. The main objective is to transform a program into a functionally equivalent one, but including capabilities to detect upsets arising in the architecture's sensitive area (internal registers, internal or external memory area, cache memory . . .).

The C2C Translator accepts as an input a C code source producing as output the C code corresponding to the hardened program according to a set of options. In fact, these options correspond to each group of rules presented in Section 2 and they can be applied separately or together. Figure 4-5 shows the entire flow of C2C translator. From the resulting C code, the assembly language code for a targeted processor can be obtained, using an ad-hoc compiler.

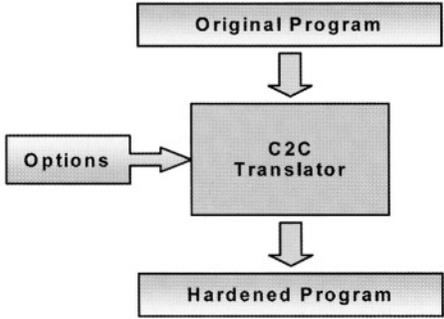


Figure 4-5. C2C Translator – schema.

### 3. EXPERIMENTAL RESULTS

In order to evaluate both the feasibility and the effectiveness of the proposed approach, we selected a C program to be used as a benchmark. We then applied the detection approach by means of C2C translator, to get the *hardened* version of the target program. Finally, we performed a set of fault injection experiments and radiation ground testing on the original program as well as in the modified version, aiming at comparing the two program’s behaviors.

During fault injection experiments, single bit flips were injected in memory areas containing the program’s code, data segment and the internal processor’s registers by software simulation [10] using a commercially available tool: the *d3sim* simulator [11]. To achieve this, the generic command file was generated (see Figure 4-6).

Command (1) tells to *d3sim* the condition when *t* (state machine counter) will be greater than *INSTANT* (the desired fault injection instant). When this condition is true, the execution of the program in progress is suspended, the XOR operation between *TARG* (the chosen SEU-target corresponding to the selected register or memory byte) and an appropriate binary mask (XOR-VALUE) is achieved, prior resuming main program execution. *TARG* symbolizes a memory location or a register. *XOR\_VALUE* specifies the targeted bit.

Command (2) sets a breakpoint at the end of the program. When the program counter reaches there, the simulator will print the memory segment

```
(0) when t>TOTAL_CYCLES{"Lost Sequence " ; quit}
(1) when t>=INSTANT {TARG=TARG^XOR_VALUE; cont}
(2) bp end_program {mem > "res.dat"; quit}
(3) run
```

Figure 4-6. Simulation commands modeling a bit flip.

of interest (where the results are stored) in a file *res.dat*. Then it closes *d3sim*.

Finally, command (3) is needed to run the studied application in the simulator.

Since particular bit flips may perturb the execution flow provoking critical dysfunction grouped under the “sequence loss” category (infinite loops or execution of invalid instructions for instance) it is mandatory to include in the “.ex” file a command implementing a watch-dog to detect such faults. A watchdog is implemented by command (0) which indicates that the program has lost the correct sequencing flow. TOTAL\_CYCLES is referring to the total number of application states machine.

In order to automate the fault injection process, a **software test bench** (Figure 4-7) was developed.

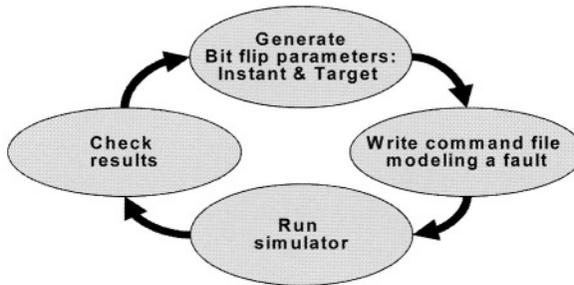


Figure 4-7. Software Test Bench automating bit flip injection.

First step of this test bench activates twice a random number generator [12] to get the two needed bit-flip parameters: the occurrence instant and the target bit, the latter chosen among all possible targets (memory locations and registers). With these parameters, a command file is written and *d3sim* simulator is started. The *d3sim* will simulate the execution of the studied application by the DSP32C according to the commands of the generic file. The last step will check results and compare them with the expected pattern this to classify the injected faults.

### 3.1. Main characteristics of the studied programs

The application that we considered for the experimentation was a Constant Modulus Algorithm (CMA), used in space communications. This application will be called in the following *CMA Original*. The set of rules above described were automatically applied on the *CMA Original* program, getting a new programs called in the following *CMA Hardened*. Main features of these programs are summarized in Table 4-1.

The experiment consisted in several fault injection sessions. For each program were performed single bit flip injections targeting the DSP32C

Table 4-1. Main characteristics of the studied programs.

	CMA original	CMA hardened	Overhead
Execution time (cycles)	1,231,162	3,004,036	2.44
Code size (bytes)	1,104	4,000	3.62
Data size (bytes)	1,996	4,032	2.02

registers, the program code and the application workspace. During fault injection sessions, faults were classified according to their effects on the program behavior. The following categories were considered:

- *Effect-less*: The injected fault does not affect the program behavior.
- *Software Detection*: The implemented rules detect the injected fault.
- *Hardware Detection*: The fault triggers some hardware mechanism (e.g., illegal instruction exception).
- *Loss Sequence*: The program under test triggers some time-out condition (e.g., endless loop).
- *Incorrect Answer*: The fault was not detected in any way and the result is different from the expected one.

In order to quantify the error detection capabilities, two metrics were introduced: the *detection efficiency* ( $\xi$ ) and the *failure rate* ( $\tau$ ).

$$\xi = \frac{D}{D + E + L + H} \tag{Eq. 4-1}$$

$$\tau = \frac{E + L}{\# \text{ Injected Faults}} \tag{Eq. 4-2}$$

Where:

- D represents the number of *Software Detection* faults
- L is the number of *Loss-Sequence* faults
- E is the number of *Incorrect Answer* faults
- H is the number of *Hardware Detection* faults

### 3.1.1. Fault injection in the DSP32C registers

Table 4-2 reports the results obtained by injecting faults modeling a bit flip in the internal registers of DSP32C processor. To obtain realistic results, the number of injected faults was selected according to the total number of processor cycles needed to execute the programs; indeed the faults injected into the processor can affect the program results only during its execution. Thus, 8000 faults were injected into the *CMA Original*, while 19520 were injected for the *CMA Hardened* (according to the penalty time factor about 2 times given in Table 4-1).

Table 4-2. Experimental results – faults injected in the DSP32C registers.

Program version	#Injected faults	#Effect less	Detected-faults		Undetected-faults	
			#Software detection	#Hardware detection	#Incorrect answer	#Loss sequence
Original CMA	8000	7612 (95.15%)	–	88 (1.10%)	114 (1.43%)	186 (2.32%)
Hardened CMA	19520	18175 (93.11%)	1120 (5.74%)	193 (0.99%)	25 (0.13%)	7 (0.03%)

In order to compare both versions of the *CMA* program, the detection efficiency and the error rate were calculated according to equations 4-1 and 4-2; Table 4-3 summarizes the obtained figures.

As shown in Table 4-3, for the hardened *CMA* program were detected more than 80% of the injected faults which modify the program's behavior, while the *error rate* was drastically reduced; a factor higher than 20.

Table 4-3. Detection efficiency and error rate for both program versions.

	CMA Original	CMA Hardened
Detection efficiency ( $\xi$ )	none	83.27%
Error rate ( $\tau$ )	3.75%	0.16%

### 3.1.2. Fault injection in the program code

Results gathered from fault injection sessions when faults were injected in the program code itself are shown in Table 4-4. The number of injected faults in the program code was chosen as being proportional to the memory area occupied by each tested program.

Table 4-5 illustrates the corresponding *detection efficiency* and *error rate* calculated from results obtained when faults were injected in the program

Table 4-4. Experimental results – faults injected in the program code.

Program version	#Injected faults	#Effect less	Detected-faults		Undetected-faults	
			#Software detection	#Hardware detection	#Incorrect answer	#Loss sequence
Original CMA	2208	1422 (64.40%)	–	391 (17.71%)	217 (9.83%)	178 (8.06%)
Hardened CMA	8000	5038 (62.98%)	1658 (20.73%)	1210 (15.12%)	50 (0.62%)	44 (0.55%)

Table 4-5. Detection efficiency and error rate for both program versions.

	CMA Original	CMA Hardened
Detection Efficiency (?)	none	55.97 %
Error Rate (t)	17.89 %	1.18 %

code. This results show that about 60% of injected faults changing the program’s behavior have been detected and a significant decrease of *error rate* (about 17 times) for the hardened application.

### 3.1.3. Fault injection in the data memory area

When faults have been injected in the data segment area (results illustrated in Table 4-6), the hardened CMA program was able to provide 100% detection efficiency. It is important to note that output data (program results) were located in a “non targeted” zone aiming at being as close as possible to a real situation where output samples will get out the equalizer by means of parallel or serial ports without residing in any internal RAM zone.

Table 4-6. Experimental results – faults injected in the data segment area.

Program version	#Injected faults	#Effect less	Detected-faults		Undetected-faults	
			#Software detection	#Hardware detection	#Incorrect answer	#Loss sequence
Original CMA	5000	4072 (81.44%)	–	–	928 (18.56%)	–
Hardened CMA	10000	6021 (60.21%)	3979 (39.79%)	–	–	–

## 3.2. Preliminary radiation testing campaign

To confirm the results gathered from fault injection experiments, we performed a radiation test campaign. The radiation experiments consisted in exposing only the processor to the charged particles issued from a Californium (Cf225) source while executing both versions of the CMA program. Table 4-7 shows the main features of the radiation experiments.

Table 4-7. Main characteristics for the radiation test campaign.

Program version	Flux	Estimated upsets	Exposure time
Original CMA	285	387	525
Hardened CMA	285	506	660

Where:

- Flux represents the number of particles reaching the processor per square unit and time unit;
- *Time exposure* is the duration of the experiment (sec.)
- Estimated Upsets represents the number of upsets expected during the entire radiation experiment.

The obtained results are illustrated in Table 4-8. As expected the number of observed upsets is double for the hardened CMA program. This is due to the longer execution under radiation. Even if the number of particles hitting the processor is higher for the hardened CMA application, the number of undetected upsets is reduced about 3.2 times.

In accordance with the number of estimated upsets occurring in the storage elements (internal memory and registers) of the DSP processor, we calculated the *detection efficiency* and *error rate* for both versions of the CMA program. Table 4-9 shows that the error rate was reduced about 4 times for the hardened application while the detection efficiency is higher than 84%, thus proving that the error detection technique is efficient in real harsh environments.

Table 4-8. Experimental results – faults injected in the data area.

Program Version	#Observed upsets	Detected-Faults		Undetected-Faults	
		#Software detection	#Hardware detection	#Incorrect answer	#Loss sequence
Original CMA	48	–	–	47	1
Hardened CMA	99	84	–	15	–

Table 4-9. Detection efficiency and error rate for both program versions.

	CMA Original	CMA Hardened
Detection efficiency ( $\xi$ )	none	84.85%
Error rate ( $\tau$ )	12.40%	2.96%

## 4. CONCLUSIONS

In this paper we presented a software error detection method and a tool for automatic generation of *hardened* applications. The technique is exclusively based on the modification of the application code and does not require any special hardware. Consequently, we can conclude that the method is suitable for usage in low-cost safety-critical applications, where the high constraints involved in terms of memory overhead (about 4 times) and speed decrease

(about 2.6 times) can be balanced by the low cost and high reliability of the resulting code.

We are currently working to evaluate the efficiency of the proposed error detection approach and the C2C Translator when applied to an industrial application executed on different complex processors. We are also investigating the possibility to apply the error detection technique to an OS (Operating System) and to evaluate it through radiation test experiments.

## REFERENCES

1. T. P. Ma and P. Dussendorfer. *Ionizing Radiation Effects in MOS Devices and Circuits*. Wiley, New-York, 1989.
2. E. Normand. *Single Event Effects in Avionics*. IEEE Trans. on Nuclear Science, Vol. 43, no. 2, pp. 461–474, April 1966.
3. B. Randell. *System Structure for Software Fault Tolerant*. IEEE Trans. on Software Engineering, Vol. 1, no. 2, June 1975, pp. 220–232.
4. A. Avizienis. *The N-Version Approach to Fault-Tolerant Software*. IEEE Trans. on Software Engineering, Vol. 11, No. 12, pp. 1491–1501, December 1985.
5. K. H. Huang and J. A. Abraham. *Algorithm-Based Fault Tolerance for Matrix Operations*. IEEE Trans. on Computers, Vol. 33, pp. 518–528, December 1984.
6. Z. Alkhalifa, V. S. S. Nair, N. Krishnamurthy, and J. A. Abraham. *Design and Evaluation of System-level Checks for On-line Control Flow Error Detection*. IEEE Trans. on Parallel and Distributed Systems, Vol. 10, No. 6, pp. 627–641, June 1999.
7. S. S. Yau and F. C. Chen. *An Approach to Concurrent Control Flow Checking*. IEEE Trans. on Software Engineering, Vol. 6, No. 2, pp. 126–137, March 1980.
8. M. Zenha Relá, H. Madeira, and J. G. Silva. “Experimental Evaluation of the Fail-Silent Behavior in Programs with Consistency Checks.” *Proc. FTCS-26*, pp. 394–403, 1996.
9. M. Rebaudengo, M. Sonza Reorda, M. Torchiano, and M. Violante. “Soft-error Detection through Software Fault-Tolerance Techniques.” *DFT’99: IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, Austin (USA), November 1999, pp. 210–218.
10. R. Velazco, A. Corominas, and P. Ferreyra. “Injecting Bit Flip Faults by Means of a Purely Software Approach: A Case Studied.” *Proceedings of the 17th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT 2002)*.
11. WE DSP32 and DSP32C Support Software Library. User Manual.
12. M. Matsumoto, T. Nishimura, and Mersenne Twister. *A 623-Dimensionally Equidistributed Uniform Pseudorandom Number Generator*. ACM Trans. on Modeling and Computer Simulation, Vol. 8, No. 1, pp. 3–30, January 1998.

*This page intentionally left blank*

PART II:

OPERATING SYSTEM ABSTRACTION AND TARGETING

*This page intentionally left blank*

## Chapter 5

# RTOS MODELING FOR SYSTEM LEVEL DESIGN

Andreas Gerstlauer, Haobo Yu and Daniel D. Gajski

*Center for Embedded Computer Systems, University of California, Irvine, Irvine, CA 92697, USA; E-mail: {gerstl,haoboy,gajski}@cecs.uci.edu; Web-site: <http://www.cecs.uci.edu>*

**Abstract.** System level synthesis is widely seen as the solution for closing the productivity gap in system design. High-level system models are used in system level design for early design exploration. While real time operating systems (RTOS) are an increasingly important component in system design, specific RTOS implementations cannot be used directly in high level models. On the other hand, existing system level design languages (SLDL) lack support for RTOS modeling. In this chapter, we propose a RTOS model built on top of existing SLDLs which, by providing the key features typically available in any RTOS, allows the designer to model the dynamic behavior of multi-tasking systems at higher abstraction levels to be incorporated into existing design flows. Experimental result shows that our RTOS model is easy to use and efficient while being able to provide accurate results.

**Key words:** RTOS, system level design, SLDL, modeling, refinement, methodology

## 1. INTRODUCTION

In order to handle the ever increasing complexity and time-to-market pressures in the design of systems-on-chip (SOCs), raising the level of abstraction is generally seen as a solution to increase productivity. Various system level design languages (SLDL) [3, 4] and methodologies [11, 13] have been proposed in the past to address the issues involved in system level design. However, most SLDLs offer little or no support for modeling the dynamic real-time behavior often found in embedded software. Embedded software is playing an increasingly significant role in system design and its dynamic real-time behavior can have a large influence on design quality metrics like performance or power. In the implementation, this behavior is typically provided by a real time operating system (RTOS) [1, 2]. At an early design phase, however, using a detailed, real RTOS implementation would negate the purpose of an abstract system model. Furthermore, at higher levels, not enough information might be available to target a specific RTOS. Therefore, we need techniques to capture the abstracted RTOS behavior in system level models.

In this chapter, we address this design challenge by introducing a high level RTOS model for system design. It is written on top of existing SLDLs and doesn't require any specific language extensions. It supports all the key concepts found in modern RTOS like task management, real time scheduling,

preemption, task synchronization, and interrupt handling [5]. On the other hand, it requires only a minimal modeling effort in terms of refinement and simulation overhead. Our model can be integrated into existing system level design flows to accurately evaluate a potential system design (e.g. in respect to timing constraints) for early and rapid design space exploration.

The rest of this chapter is organized as follows: Section 2 gives an insight into the related work on software modeling and synthesis in system level design. Section 3 describes how the RTOS model is integrated with the system level design flow. Details of the RTOS model, including its interface and usage as well as the implementation are covered in Section 4. Experimental results are shown in Section 5, and Section 6 concludes this chapter with a brief summary and an outlook on future work.

## 2. RELATED WORK

A lot of work recently has been focusing on automatic RTOS and code generation for embedded software. In [9], a method for automatic generation of application-specific operating systems and corresponding application software for a target processor is given. In [6], a way of combining static task scheduling and dynamic scheduling in software synthesis is proposed. While both approaches mainly focus on software synthesis issues, the papers do not provide any information regarding high level modeling of the operating systems integrated into the whole system.

In [15], a technique for modeling fixed-priority preemptive multi-tasking systems based on concurrency and exception handling mechanisms provided by SpecC is shown. However, the model is limited in its support for different scheduling algorithms and inter-task communication, and its complex structure makes it very hard to use.

Our method is similar to [7] where a high-level model of a RTOS called SoCOS is presented. The main difference is that our RTOS model is written on top of existing SLDLs whereas SoCOS requires its own proprietary simulation engine. By taking advantage of the SLDL's existing modeling capabilities, our model is simple to implement yet powerful and flexible, and it can be directly integrated into any system model and design flow supported by the chosen SLDL.

## 3. DESIGN FLOW

System level design is a process with multiple stages where the system specification is gradually refined from an abstract idea down to an actual heterogeneous multi-processor implementation. This refinement is achieved in a stepwise manner through several levels of abstraction. With each step, more implementation detail is introduced through a refined system model. The

purpose of high-level, abstract models is the early validation of system properties before their detailed implementation, enabling rapid exploration.

Figure 5-1 shows a typical system level design flow [10]. The system design process starts with the specification model. It is written by the designer to specify and validate the desired system behavior in a purely functional, abstract manner, i.e. free of any unnecessary implementation details. During system design, the specification functionality is then partitioned onto multiple processing elements (PEs) and a communication architecture consisting of busses and bus interfaces is synthesized to implement communication between PEs. Note that during communication synthesis, interrupt handlers will be generated inside the PEs as part of the bus drivers.

Due to the inherently sequential nature of PEs, processes mapped to the same PE need to be serialized. Depending on the nature of the PE and the data inter-dependencies, processes are scheduled statically or dynamically. In case of dynamic scheduling, in order to validate the system model at this point a representation of the dynamic scheduling implementation, which is usually handled by a RTOS in the real system, is required. Therefore, a high level model of the underlying RTOS is needed for inclusion into the system model during system synthesis. The RTOS model provides an abstraction of

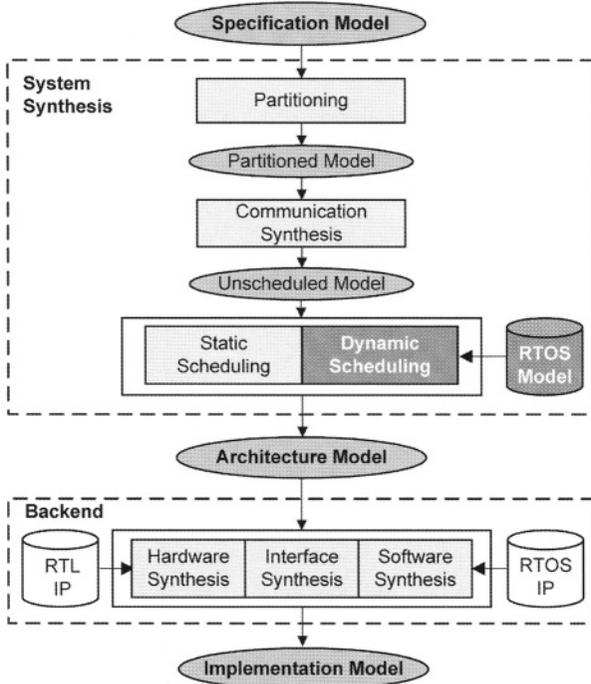


Figure 5-1. Design flow.

the key features that define a dynamic scheduling behavior independent of any specific RTOS implementation.

The dynamic scheduling step in Figure 5-1 refines the unscheduled system model into the final architecture model. In general, for each PE in the system a RTOS model corresponding to the selected scheduling strategy is imported from the library and instantiated in the PE. Processes inside the PEs are converted into tasks with assigned priorities. Synchronization as part of communication between processes is refined into OS-based task synchronization. The resulting architecture model consists of multiple PEs communicating via a set of busses. Each PE runs multiple tasks on top of its local RTOS model instance. Therefore, the architecture model can be validated through simulation or verification to evaluate different dynamic scheduling approaches (e.g. in terms of timing) as part of system design space exploration.

In the backend, each PE in the architecture model is then implemented separately. Custom hardware PEs are synthesized into a RTL description. Bus interface implementations are synthesized in hardware and software. Finally, software synthesis generates code from the PE description of the processor in the architecture model. In the process, services of the RTOS model are mapped onto the API of a specific standard or custom RTOS. The code is then compiled into the processor’s instruction set and linked against the RTOS libraries to produce the final executable.

#### 4. THE RTOS MODEL

As mentioned previously, the RTOS model is implemented on top of an existing SLDL kernel [8]. Figure 5-2 shows the modeling layers at different steps of the design flow. In the specification model (Figure 5-2(a)), the application is a serial-parallel composition of SLDL processes. Processes communicate and synchronize through variables and channels. Channels are implemented using primitives provided by the SLDL core and are usually part of the communication library provided with the SLDL.

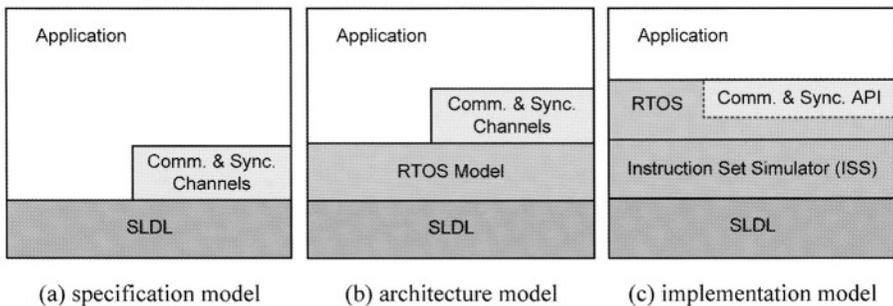


Figure 5-2. Modeling layers.

In the architecture model (Figure 5-2(b)), the RTOS model is inserted as a layer between the application and the SLDL core. The SLDL primitives for timing and synchronization used by the application are replaced with corresponding calls to the RTOS layer. In addition, calls of RTOS task management services are inserted. The RTOS model implements the original semantics of SLDL primitives plus additional details of the RTOS behavior on top of the SLDL core, using the existing services of the underlying SLDL simulation engine to implement concurrency, synchronization, and time modeling. Existing SLDL channels (e.g. semaphores) from the specification are reused by refining their internal synchronization primitives to map to corresponding RTOS calls. Using existing SLDL capabilities for modeling of extended RTOS services, the RTOS library can be kept small and efficient. Later, as part of software synthesis in the backend, RTOS calls and channels are implemented by mapping them to an equivalent service of the actual RTOS or by generating channel code on top of RTOS primitives if the service is not provided natively.

Finally, in the implementation model (Figure 5-2(c)), the compiled application linked against the real RTOS libraries is running in an instruction set simulator (ISS) as part of the system co-simulation in the SLDL.

We implemented the RTOS model on top of the SpecC SLDL [1, 8, 11]. In the following sections we will discuss the interface between application and the RTOS model, the refinement of specification into architecture using the RTOS interface, and the implementation of the RTOS model.

Due to space restrictions, implementation details are limited. For more information, please refer to [16].

#### 4.1. RTOS interface

Figure 5-3 shows the interface of the RTOS model. The RTOS model provides four categories of services: operating system management, task management, event handling, and time modeling.

Operating system management mainly deals with initialization of the RTOS during system start where *init* initializes the relevant kernel data structures while *start* starts the multi-task scheduling. In addition, *interrupt\_return* is provided to notify the RTOS kernel at the end of an interrupt service routine.

Task management is the most important function in the RTOS model. It includes various standard routines such as task creation (*task\_create*), task termination (*task\_terminate*, *task\_kill*), and task suspension and activation (*task\_sleep*, *task\_activate*). Two special routines are introduced to model dynamic task forking and joining: *par\_start* suspends the calling task and waits for the child tasks to finish after which *par\_end* resumes the calling task's execution. Our RTOS model supports both periodic hard real time tasks with a critical deadline and non-periodic real time tasks with a fixed priority. In modeling of periodic tasks, *task\_endcycle* notifies the kernel that a periodic task has finished its execution in the current cycle.

---

```

1  interface RTOS
   {
       /* OS management */
       void init();
5     void start(int sched_alg);
       void interrupt_return();

       /* Task management */
       Task task_create(const char *name, int type,
10          sim_time period);
       void task_terminate();
       void task_sleep();
       void task_activate(Task t);
       void task_endcycle();
15     void task_kill(Task t);
       Task par_start();
       void par_end(Task t);

       /* Event handling */
20     Task enter_wait();
       void wakeup_wait(Task t);

       /* Delay modeling */
       void time_wait(sim_time nsec);
25 };

```

---

Figure 5-3. Interface of the RTOS model.

Event handling in the RTOS model sits on top of the basic SLDL synchronization events. Two system calls, *enter\_wait* and *wakeup\_wait*, are wrapped around each SpecC **wait** primitive. This allows the RTOS model to update its internal task states (and to reschedule) whenever a task is about to get blocked on and later released from a SpecC **event**.

During simulation of high-level system models, the logical time advances in discrete steps. SLDL primitives (such as **waitfor** in SpecC) are used to model delays. For the RTOS model, those delay primitives are replaced by *time\_wait* calls which model task delays in the RTOS while enabling support for modeling of task preemption.

## 4.2. Model refinement

In this section, we will illustrate application model refinement based on the RTOS interface presented in the previous section through a simple yet typical example of a single *PE* (Figure 5-4). In general, the same refinement steps are applied to all the PEs in a multi-processor system. The unscheduled model (Figure 5-4(a)) executes behavior *B1* followed by the parallel composition of

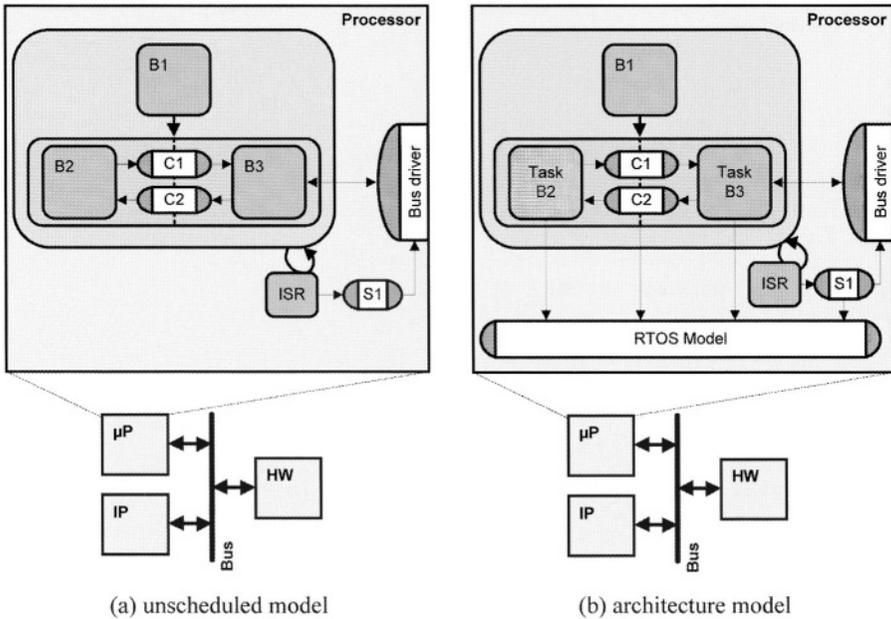


Figure 5-4. Model refinement example.

behaviors  $B2$  and  $B3$ . Behaviors  $B2$  and  $B3$  communicate via two channels  $C1$  and  $C2$  while  $B3$  communicates with other PEs through a bus driver. As part of the bus interface implementation, the interrupt handler  $ISR$  for external events signals the main bus driver through a semaphore channel  $S1$ .

The output of the dynamic scheduling refinement process is shown in Figure 5-4(b). The RTOS model implementing the RTOS interface is instantiated inside the PE in the form of a SpecC channel. Behaviors, interrupt handlers and communication channels use RTOS services by calling the RTOS channel’s methods. Behaviors are refined into three tasks.  $Task\_PE$  is the main task that executes as soon as the system starts. When  $Task\_PE$  finishes executing  $B1$ , it spawns two concurrent child tasks,  $Task\_B2$  and  $Task\_B3$ , and waits for their completion.

#### 4.2.1. Task refinement

Task refinement converts parallel processes/behaviors in the specification into RTOS-based tasks in a two-step process. In the first step (Figure 5-5), behaviors are converted into tasks, e.g. behavior  $B2$  (Figure 5-5(a)) is converted into  $Task\_B2$  (Figure 5-5(b)). A method *init* is added for construction of the task. All **waitfor** statements are replaced with RTOS *time\_wait* calls to model task execution delays. Finally, the main body of the task is enclosed in a pair of *task\_activate* / *task\_terminate* calls so that the RTOS kernel can control the task activation and termination.

---

```

1 behavior B2 ()
  {
    void main(void)
      {
5         ...
          /* model execution delay */
          waitfor(BLOCK1_DELAY);
          ...
          /* model execution delay */
10        waitfor(BLOCK2_DELAY);
          ...
      }
  };

```

---

(a) unscheduled model

---

```

1 behavior task_B2(RTOS os) implements Init
  {
    Task h;

5    void init(void) {
        h = os.task_create("B2", APERIODIC, 0);
    }

    void main(void) {
10     os.task_activate(h);
        ...
        /* model execution delay */
        os.time_wait(BLOCK1_DELAY);
15     ...
        /* model execution delay */
        os.time_wait(BLOCK2_DELAY);
        ...
        os.task_terminate(h);
20    }
  };

```

---

Figure 5-5. Task modeling.

The second step (Figure 5-6) involves dynamic creation of child tasks in a parent task. Every **par** statement in the code (Figure 5-6(a)) is refined to dynamically fork and join child tasks as part of the parent's execution (Figure 5-6(b)). The *init* methods of the children are called to create the child tasks. Then, *par\_start* suspends the calling parent task in the RTOS layer before the children are actually executed in the **par** statement. After the two child tasks finish execution and the **par** exits, *par\_end* resumes the execution of the parent task in the RTOS layer.

<pre> 1 <b>behavior</b> B2B3 ()   {     B2 b2 ();     B3 b3 (); 5     <b>void</b> main(<b>void</b>)     {       <b>par</b> 10      {         b2.main();         b3.main();       }     } 15  }; </pre>	<pre> 1 <b>behavior</b> B2B3 (RTOS os)   {     Task_B2 task_b2(os);     Task_B3 task_b3(os); 5     <b>void</b> main(<b>void</b>) {       Task t;        task_b2.init();       task_b3.init();        t = os.par_start();       <b>par</b> { 10        b2.main();         b3.main();       }       os.par_end(t);     } 15  }; </pre>
(a) before	(b) after

Figure 5-6. Task creation.

4.2.2. Synchronization refinement

In the specification model, all synchronization in the application or inside communication channels is implemented using SLDL events. Synchronization refinement wraps corresponding event handling routines of the RTOS model around the event-related primitives (Figure 5-7). Each **wait** statement in the code is enclosed in a pair of *enter\_wait / wakeup\_wait* calls to notify the RTOS model about corresponding task state changes. Note that there is no need to refine **notify** primitives as the state of the calling task is not influenced by those calls.

After model refinement, both task management and synchronization are implemented using the system calls of the RTOS model. Thus, the dynamic system behavior is completely controlled by the RTOS model layer.

4.3. Implementation

The RTOS model library is implemented in approximately 2000 lines of SpecC channel code [16]. The library contains models for different scheduling strategies typically found in RTOS implementations, e.g. round-robin or priority-based scheduling [14]. In addition, the models are parametrizable in terms of task parameters, preemption, and so on.

Task management in the RTOS models is implemented in a customary

<pre> 1  channel C1()    {        event eReady;        event eAck; 5        void send(...)        {            ...            notify eRdy; 10          ...            wait(eAck);            ...        } 15  }; </pre>	<pre> 1  channel C1(RTOS os)    {        event eReady;        event eAck; 5        void send(...) {            Task t;            ...            notify eRdy; 10          ...            t = os.enter_wait();            wait(eAck);            os.wakeup_wait(t);            ...        } 15  }; </pre>
(a) before	(b) after

Figure 5-7. Synchronization refinement.

manner where tasks transition between different states and a task queue is associated with each state [5]. Task creation (*task\_create*) allocates the RTOS task data structure and *task\_activate* inserts the task into the ready queue. The *par\_start* method suspends the task and calls the scheduler to dispatch another task while *par\_end* resumes the calling task's execution by moving the task back into the ready queue.

Event management is implemented by associating additional queues with each event. Event creation (*event\_new*) and deletion (*event\_del*) allocate and deallocate the corresponding data structures in the RTOS layer. Blocking on an event (*event\_wait*) suspends the task and inserts it into the event queue whereas *event\_notify* moves all tasks in the event queue back into the ready queue.

In order to model the time-sharing nature of dynamic task scheduling in the RTOS, the execution of tasks needs to be serialized according to the chosen scheduling algorithm. The RTOS model ensures that at any given time only one task is running on the underlying SLDL simulation kernel. This is achieved by blocking all but the current task on SLDL events. Whenever task states change inside a RTOS call, the scheduler is invoked and, based on the scheduling algorithm and task priorities, a task from the ready queue is selected and dispatched by releasing its SLDL event. Note that replacing SLDL synchronization primitives with RTOS calls is necessary to keep the internal task state of the RTOS model updated.

In high level system models, simulation time advances in discrete steps based on the granularity of **waitfor** statements used to model delays (e.g. at behavior or basic block level). The time-sharing implementation in the RTOS

model makes sure that delays of concurrent task are accumulative as required by any model of serialized task execution. However, additionally replacing **waitfor** statements with corresponding RTOS time modeling calls is necessary to accurately model preemption. The *time\_wait* method is a wrapper around the **waitfor** statement that allows the RTOS kernel to reschedule and switch tasks whenever time increases, i.e. in between regular RTOS system calls.

Normally, this would not be an issue since task state changes can not happen outside of RTOS system calls. However, external interrupts can asynchronously trigger task changes in between system calls of the current task in which case proper modeling of preemption is important for the accuracy of the model (e.g. response time results). For example, an interrupt handler can release a semaphore on which a high priority task for processing of the external event is blocked. Note that, given the nature of high level models, the accuracy of preemption results is limited by the granularity of task delay models.

Figure 5-8 illustrates the behavior of the RTOS model based on simulation results obtained for the example from Figure 5-4. Figure 5-8(a) shows the simulation trace of the unscheduled model. Behaviors *B2* and *B3* are executing truly in parallel, i.e. their simulated delays overlap. After executing for time  $d_1$ , *B3* waits until it receives a message from *B2* through the channel *C1*. Then it continues executing for time  $d_2$  and waits for data from another PE. *B2* continues for time  $(d_6 + d_7)$  and then waits for data from *B3*. At time

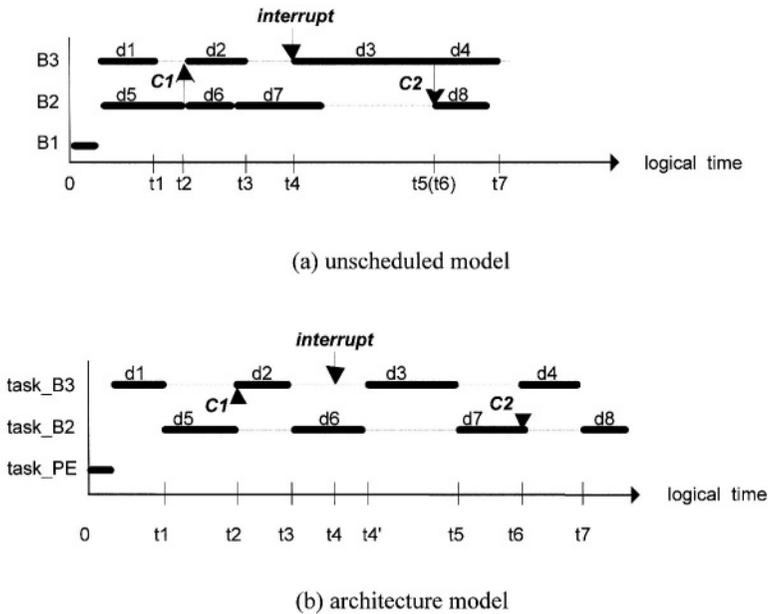


Figure 5-8. Simulation trace for model example.

$t_4$ , an interrupt happens and  $B3$  receives its data through the bus driver.  $B3$  executes until it finishes. At time  $t_5$ ,  $B3$  sends a message to  $B2$  through the channel  $C2$ , which wakes up  $B2$ , and both behaviors continue until they finish execution.

Figure 5-8(b) shows the simulation result of the architecture model for a priority based scheduling. It demonstrates that in the refined model  $task\_B2$  and  $task\_B3$  execute in an interleaved way. Since  $task\_B3$  has the higher priority, it executes unless it is blocked on receiving or sending a message from/to  $task\_B2$  ( $t_1$  through  $t_2$  and  $t_5$  through  $t_6$ ), it is waiting for an interrupt ( $t_3$  through  $t_4$ ), or it finishes ( $t_7$ ). At those points execution switches to  $task\_B2$ . Note that at time  $t_4$ , the interrupt wakes up  $task\_B3$ , and  $task\_B2$  is preempted by  $task\_B3$ . However, the actual task switch is delayed until the end of the discrete time step  $d_6$  in  $task\_B2$  based on the granularity of the task's delay model. In summary, as required by priority based dynamic scheduling, at any time only one task, the ready task with the highest priority, is executing.

## 5. EXPERIMENTAL RESULTS

We applied the RTOS model to the design of a voice codec for mobile phone applications [12]. The Vocoder contains two tasks for encoding and decoding in software, assisted by a custom hardware co-processor. For the implementation, the Vocoder was compiled into assembly code for the Motorola DSP56600 processor and linked against a small custom RTOS kernel that uses a scheduling algorithm where the decoder has higher priority than the encoder, described in more detail in [12].

In order to perform design space exploration, we evaluated different architectural alternatives at the system level using our RTOS model. We created three scheduled models of the Vocoder with varying scheduling strategies: round-robin scheduling and priority-based scheduling with alternating relative priorities of encoding and decoding tasks.

Table 5-1 shows the results of this architecture exploration in terms of

Table 5-1. Experimental results.

	Modeling		Simulation	
	Lines of code	Simulation time	Context switches	Transcoding delay
Unscheduled	11,313	27.3 s	0	9.7 ms
Round-robin	13,343	28.6 s	3,262	10.29 ms
Encoder > Decoder	13,356	28.9 s	980	11.34 ms
Decoder > Encoder	13,356	28.5 s	327	10.30 ms
Implementation	79,096	~ 5 h	327	11.7 ms

modeling effort and simulation results. The Vocoder models were exercised by a testbench that feeds a stream of 163 speech frames corresponding to 3.26 s of speech into encoder and decoder. Furthermore, the models were annotated to deliver feedback about the number of RTOS context switches and the transcoding delay encountered during simulation. The transcoding delay is the latency when running encoder and decoder in back-to-back mode and is related to response time in switching between encoding and decoding tasks.

The results show that refinement based on the RTOS model requires only a minimal effort. Refinement into the three architecture models was done by converting relevant SpecC statements into RTOS interface calls following the steps described in Section 4.2. For this example, manual refinement took less than one hour and required changing or adding 104 lines or less than 1% of code. Moreover, we have developed a tool that performs the refinement of unscheduled specification models into RTOS-based architecture models automatically. With automatic refinement, all three models could be created within seconds, enabling rapid exploration of the RTOS design alternatives.

The simulation overhead introduced by the RTOS model is negligible while providing accurate results. As explained by the fact that both tasks alternate with every time slice, round-robin scheduling causes by far the largest number of context switches while providing the lowest response times. Note that context switch delays in the RTOS were not modeled in this example, i.e. the large number of context switches would introduce additional delays that would offset the slight response time advantage of round-robin scheduling in a final implementation. In priority-based scheduling, it is of advantage to give the decoder the higher relative priority. Since the encoder execution time dominates the decoder execution time this is equivalent to a shortest-job-first scheduling which minimizes wait times and hence overall response time. Furthermore, the number of context switches is lower since the RTOS does not have to switch back and forth between encoder and decoder whenever the encoder waits for results from the hardware co-processor. Therefore, priority-based scheduling with a high-priority decoder was chosen for the final implementation. Note that the final delay in the implementation is higher due to inaccuracies of execution time estimates in the high-level model.

In summary, compared to the huge complexity required for the implementation model, the RTOS model enables early and efficient evaluation of dynamic scheduling implementations.

## **6. SUMMARY AND CONCLUSIONS**

In this chapter, we proposed a RTOS model for system level design. To our knowledge, this is the first attempt to model RTOS features at such high abstraction levels integrated into existing languages and methodologies. The model allows the designer to quickly validate the dynamic real time behavior of multi-task systems in the early stage of system design by providing accurate

results with minimal overhead. Using a minimal number of system calls, the model provides all key features found in any standard RTOS but not available in current SLDLs. Based on this RTOS model, refinement of system models to introduce dynamic scheduling is easy and can be done automatically. Currently, the RTOS model is written in SpecC because of its simplicity. However, the concepts can be applied to any SLDL (SystemC, Superlog) with support for event handling and modeling of time.

Future work includes implementing the RTOS interface for a range of custom and commercial RTOS targets, including the development of tools for software synthesis from the architecture model down to target-specific application code linked against the target RTOS libraries.

## REFERENCES

1. QNX[online]. Available: <http://www.qnx.com/>.
2. VxWorks[online]. Available: <http://www.vxworks.com/>.
3. SpecC[online]. Available: <http://www.specc.org/>.
4. SystemC[online]. Available: <http://www.systemc.org/>.
5. G. C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer Academic Publishers, 1999.
6. J. Cortadella. "Task Generation and Compile Time Scheduling for Mixed Data-Control Embedded Software." In *Proceedings of Design Automation Conference (DAC)*, June 2000.
7. D. Desmet, D. Verkest, and H. De Man. "Operating System Based Software Generation for System-on-Chip." In *Proceedings of Design Automation Conference (DAC)*, June 2000.
8. R. Dömer, A. Gerstlauer, and D. D. Gajski. *SpecC Language Reference Manual, Version 2.0*. SpecC Technology Open Consortium, December 2002.
9. L. Gauthier, S. Yoo, and A. A. Jerraya. "Automatic Generation and Targeting of Application-Specific Operating Systems and Embedded Systems Software." *IEEE Transactions on CAD*, November 2001.
10. A. Gerstlauer and D. D. Gajski. "System-Level Abstraction Semantics." In *Proceedings of International Symposium on System Synthesis (ISSS)*, October 2002.
11. A. Gerstlauer, R. Dömer, J. Peng, and D. D. Gajski. *System Design: A Practical Guide with SpecC*. Kluwer Academic Publishers, 2001.
12. A. Gerstlauer, S. Zhao, D. D. Gajski, and A. Horak. "Design of a GSM Vocoder using SpecC Methodology." Technical Report ICS-TR-99-11, UC Irvine, February 1999.
13. T. Grötter, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.
14. D. Steppner, N. Rajan, and D. Hui. "Embedded Application Design Using a Real-Time OS." In *Proceedings of Design Automation Conference (DAC)*, June 1999.
15. H. Tomiyama, Y. Cao, and K. Murakami. "Modeling Fixed-Priority Preemptive Multi-Task Systems in SpecC." In *Proceedings of Workshop on Synthesis and System Integration of Mixed Technologies (SASIMI)*, October 2001.
16. H. Yu, A. Gerstlauer, and D. D. Gajski. "RTOS Modeling in System Level Synthesis." Technical Report CECS-TR-02-25, UC Irvine, August 2002.

## Chapter 6

# MODELING AND INTEGRATION OF PERIPHERAL DEVICES IN EMBEDDED SYSTEMS

Shaojie Wang<sup>1</sup>, Sharad Malik<sup>1</sup> and Reinaldo A. Bergamaschi<sup>2</sup>

<sup>1</sup> *Department of Electrical Engineering, Princeton University, NJ, USA;* <sup>2</sup> *IBM T. J. Watson Research Center, NY, USA*

**Abstract.** This paper describes automation methods for device driver development in IP-based embedded systems in order to achieve high reliability, productivity, reusability and fast time to market. We formally specify device behaviors using event driven finite state machines, communication channels, declaratively described rules, constraints and synthesis patterns. A driver is synthesized from this specification for a virtual environment that is platform (processor, operating system and other hardware) independent. The virtual environment is mapped to a specific platform to complete the driver implementation. The illustrative application of our approach for a USB device driver in Linux demonstrates improved productivity and reusability.

**Key words:** embedded software, embedded software synthesis, peripheral modeling

## 1. INTRODUCTION

Device drivers provide a bridge between a peripheral device and the upper layers of the operating system and the application software. They are critical software elements that significantly affect design quality and productivity. Given the typical lifecycle of chipsets being only 12 to 24 months, system designers have to redesign the hardware and software regularly to keep up with the pace of new product releases. This requires constant updates of the device drivers. Design and verification of device drivers is very complicated due to necessity of thorough knowledge about chips and boards, processors, peripherals, operating systems, compilers, logic and timing requirements; each of which is considered to be tedious. For example, Motorola MPC860 PowerQUICC is an SoC micro-controller used in communications and networking applications. Its board support package (BSP) (essentially drivers) has 25000 lines of C code [6] – an indication of its complexity. With time-to-market requirements being pushed below one year, driver development is quickly becoming a bottleneck in IP-based embedded system design. Automation methods, software reusability and other approaches are badly needed to improve productivity and are the subject of this paper.

The design and implementation of reliable device drivers is notoriously

difficult and constitutes the main portion of system failures. As an example, a recent report on Microsoft Windows XP crash data [7] shows that 61% of XP crashes are caused by driver problems. The proposed approach addresses reliability in two ways. Formal specification models provide for the ability to validate the specification using formal analysis techniques. For example, the event-driven state machine models used in our approach are amenable to model checking techniques. Correct by construction synthesis attempts to eliminate implementation bugs. Further, the formal specifications can be used as manuals for reusing this component, and inputs for automating the composition with other components.

Another key concern in driver development is portability. Device drivers are highly platform (processor, operating system and other hardware) dependent. This is especially a problem when design space exploration involves selecting from multiple platforms. Significant effort is required to port the drivers to different platforms. Universal specifications that can be rapidly mapped to a diverse range of platforms, such as provided by our approach, are required to shorten the design exploration time.

The approach presented in this paper addresses the complexity and portability issues raised above by proposing a methodology and a tool for driver development. This methodology is based on a careful analysis of devices, device drivers and best practices of expert device driver writers. Our approach codifies these by clearly defining a device behavior specification, as well as a driver development flow with an associated tool. We formally specify device behaviors by describing clearly demarcated behavior components and their interactions. This enables a designer to easily specify the relevant aspects of the behavior in a clearly specified manner. A driver is synthesized from this specification for a virtual environment that is platform (processor, operating system and other hardware) independent. The virtual environment is then mapped to a specific platform to complete the driver implementation.

The remainder of this paper is organized as follows: Section 2 reviews related work; Section 3 describes the framework for our methodology; Section 4 presents the formal specification of device behavior using the Universal Serial Bus (USB) as an example; Section 5 discusses driver synthesis; Section 6 describes our case study; and finally Section 7 discusses future work and directions.

## **2. RELATED WORKS**

Recent years have seen some attention devoted to this issue in both academia and industry. Devil [3] defines an interface definition language (IDL) to abstract device register accesses, including complex bit-level operations. From the IDL specification, it generates a library of register access functions and supports partial analysis for these functions. While Devil provides some abstraction for the developer by hiding the low-level details of bit-level

programming, its approach is limited to register accesses and it does not address the other issues in device driver development outlined above.

In the context of co-design automation, O’Nils and Jantsch [4] propose a regular language called ProGram to specify hardware/software communication protocols, which can be compiled to software code. While there are some interesting features, this does not completely address the device driver problems as described here, particularly due to its inability to include an external OS. Other efforts in the co-design area [1, 2] are limited to the mapping of the communications between hardware and software to interrupt routines that are a small fraction of a real device driver.

I2O (Intelligent Input Output) [8] defines a standard architecture for intelligent I/O that is independent of both the specific device being controlled and the host operating system. The device driver portability problem is handled by specifying a communication protocol between the host system and the device. Because of the large overhead of the implementation of the communication protocol, this approach is limited to high-performance markets. Like I2O, UDI [9] (Uniform Driver Interface) is an attempt to address portability. It defines a set of Application Programming Interfaces (APIs) between the driver and the platform. Drivers and operating systems are developed independently. UDI API’s are OS and platform neutral and thus source-code level reuse of driver code is achieved. Although UDI and our methodology share the common feature of platform and OS neutral service abstraction, our methodology is based on a formal model that enables verification and synthesis.

### 3. METHODOLOGY FRAMEWORK

Devices are function extensions of processors. They exchange data with processors, respond to processor requests and actively interact with processors, typically through interrupts. Processors control and observe devices through the *device-programming interface*, which defines I/O registers and mapped memories. Figure 6-1 sketches the relationship between devices, processors, the operating system and device drivers.

Processors access devices through memory mapped I/O, programmed I/O or DMA. A *data path* (or communication channel) is a specific path for exchanging data between the processor and the device as illustrated in Figure 6-1.

To hide the details of device accesses, device drivers are designed to be a layer between the high-level software and low-level device. In most cases, a device driver is part of the kernel. Application software, which resides in user space, uses system calls to access the kernel driver. System calls use traps to enter kernel mode and dispatch requests to a specific driver. Hence we can partition a device driver into three parts as illustrated in Figure 6-1 and explained below:

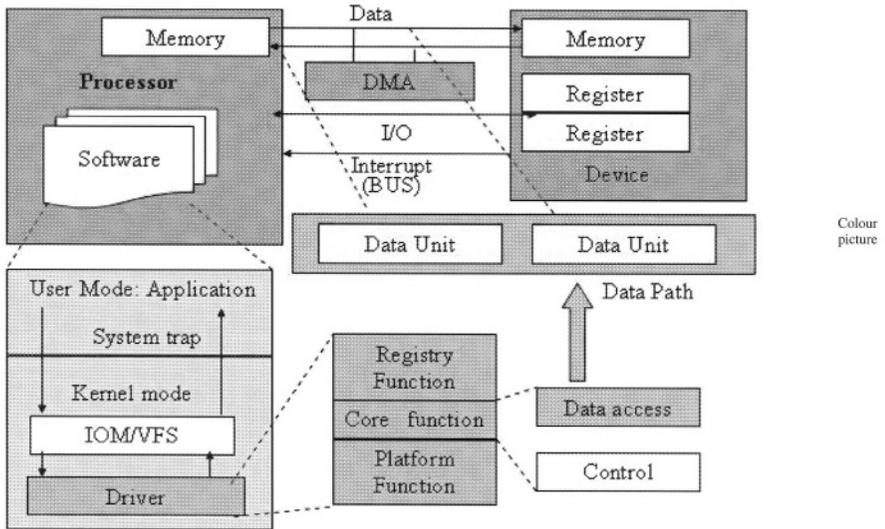


Figure 6-1. Driver environment.

- *Core functions*, which trace the states of devices, enforce device state transitions required for certain operations, and operate data paths. Because such actions depend on the current states of the device, synchronization with the device is necessary. Common synchronization approaches are interrupts, polling and timer delay. In our approach, core functions are synthesized from a device specification. They interact with a platform independent framework called *virtual environment*. The device specification itself is explained in Section 4, and synthesis of core functions in Section 5.
- *Platform functions* that glue the core functions to the hardware and OS platform. The virtual environment abstracts architectural behaviors such as big or little endian, programmed I/O, memory mapped I/O. It also specifies OS services and OS requirements such as memory management, DMA/bus controllers, synchronization mechanisms etc. This virtual environment is then mapped to a particular platform (hardware and OS) by providing the platform functions that have platform specific code for the above details.
- *Registry functions* that export driver services into the kernel or application name space. For example, the interface can register a driver class to the NT I/O manager (IOM) or fill one entry of the VFS (Virtual File System) structure of the Linux kernel.

Figure 6-2 outlines our framework and illustrates how the three parts of the driver come together. It takes as input (1) the device specification, which is platform neutral and (2) the driver configuration, which specifies the device

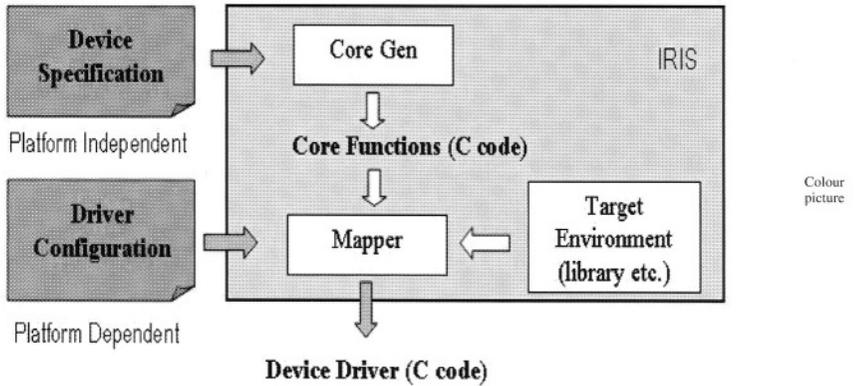


Figure 6-2. Framework overview.

instance and environment; and outputs the device driver (C program) for a particular device in a particular environment.

The device specification provides all the information about the device required by the driver core function synthesis process (Core Gen). Core functions are implemented in the virtual environment. The *mapper* maps core functions in the virtual environment to the target environment. This mapping process does (1) platform mapping, by binding virtual OS service functions to the targeted OS, and virtual hardware functions to the targeted hardware; (2) registry mapping by translating core functions to the OS specific registry functions. It does this using the developer specified *driver configuration*. The driver configuration defines the target platform and necessary parameters. It includes the OS name, processor name, bus name, device instance parameters (such as interrupt vector number, base address), driver type (such as char device, network device), driver type specific parameters (such as maximum transfer unit for a network device) etc. These configurations are specified by keywords. Figure 6-3 shows the sketch of an example. This direct specification is sufficient for the mapper to target the core functions to a specific platform. While this requires the specification writer to have specific knowl-

```
.drv udc ( .config {
.os Linux 2.4.x .processor SA1100
.irq 13 .io_base 0x80000000
.io_ext 0x30 .drvType Ethnet
...
} ) { ... }
```

Figure 6-3. SA1100 USB Device Controller (UDC) driver configuration.

edge of the various elements of the configuration, information such as driver type and driver type specific parameters are encoded once and then reused across platform specifications.

## 4. DEVICE SPECIFICATION

In this section we describe the different parts of the device specification, which is the input to the generation of the core functions (see Figure 6-2). Based on our study of devices and device drivers, we partition this specification into the following parts: data path, device control (including event handlers and control functions) and device programming interface. Figure 6-1 provides some motivation for this division. The data path describes the transfer of data between the processor and the device. The device control describes the transitions between the states of the device using event driven finite state machines (EFSM) [5]. The processing of events in the event driven state machines is specified in event handlers. The core functions provide the device services to the upper layers of the software – the OS and the application. The device-programming interface provides the low-level access functions to the device registers. Eventually it is the core functions that need to be synthesized in C as the driver – however, this synthesis will need all the other parts of this specification to understand the complete device behavior. This partitioning of the specification allows for a separation of the various domains, which the device driver has to interact with, and is similar to the approach used by expert device driver writers.

### 4.1. Data path

A *data unit* is a data block moving between the device and the software by a primitive hardware operation. For example, we can define a DMA access as a primitive hardware operation. A data unit is modeled as a tuple of data unit size, the event enabling a new data unit operation, starter function, the event indicating the end of a data unit operation, stopper function, the events indicating errors and access operations. Starter functions are the operations performed before data accesses. Stopper functions are the cleanup operations performed when the hardware data access operations end. Figure 6-4 defines the DMA data unit of the transmit channel of the SA1100 USB device controller (UDC). While the details of the specification are beyond the scope of the paper, this illustrates the nature of the information required here.

A *data path* is defined as a stream of data units. The last part of Figure 6-4 provides an example of a data path specification. It specifies data transfer direction and the data unit of the data path.

```

## DMA transmit FIFO
.dma udc_wt = dr_fifo@ep2_space .o
.setup %{ IUDC_VAR_CS2_TPC_FLIP;
        IUDC_REG_IMP_WRITE(IDMA_SIZE-1); %}
.cleanup %{ IUDC_VAR_CS2_SST_FLIP; %}
.abort ep2_not_ready
        %{ IUDC_VAR_CS2_SST_FLIP; %}
.abort ep2_tpe_or_tur
        %{ IUDC_VAR_CS2_SST_FLIP; %};

## data unit
.du ep2_du = udc_wt[64];

## data path
.dp ep2 = ep2_du .o .interface

```

Figure 6-4. SA1100 UDC Transmit Channel Data Unit and Data Path Specification.

## 4.2. Device control

The *device control* specifies the state changes of a device. Because a device may be composed of several sub-devices operating in parallel, we use a concurrency model, event driven finite state machines [5], to formally model the device control. It has several *synchronization properties*: (1) the execution of event handlers and state transitions are free of race conditions, and (2) finite state machines do not share data and communicate through events if necessary. The synthesizer enforces the race condition free property by disabling the context switches appropriately. Figure 6-5 gives the sketch of an example of device control specification of a sub-component, USB setup protocol, of SA1100 UDC. Again, detailed syntax and semantics of the specification are beyond the scope of this paper, we will focus on illustrating the salient features of the specification.

There are four types of events: *hardware events*, *input events*, *output events* and *internal events*. Devices generate hardware events to indicate changes of hardware states. A typical hardware event is an interrupt. Higher layer modules send input events (also called service requests) to the driver. Events sent to higher-layer modules are called output events. As an example, USB host assigns a USB address to the UDC. The driver can emit an output event that carries the address information when the host assigns the address. The upper-layer software observes the USB address of the device through the output event. All other events are internal. Events can be considered as messages conveying information. In addition to the event name, an event may convey information by carrying a parameter. Events are stored in a global event queue.

As shown in Figure 6-5, an event handler handles a particular event for a particular state of a state machine. The event handlers of a finite state machine

```

.fsm usb_protocol {
## The start state is Disabled. It accepts event start.
## Event handler is enclosed in %! and %!. The
## destination state is ZombieSuspend.

Disabled start ZombieSuspend %!
    IUDC_VAR_CR_UDD_WRITE(0);
    iudelay(100);
    IEvtOut(ep1_reset);
    IEvtOut(ep2_reset);
    %!;
    ...
}

## Control function. The request event is start. This
## function moves the usb_protocol state machine to
## ZombieSuspend state, ep1 and ep2 to idle state.
## ( Note: ep1 and ep2 are the implicit state
## machines for data flows ep1 and ep2. )

.ctrl start (ep1, idle) (ep2, idle)
(usb_protocol, ZombieSuspend);

```

Figure 6-5. SA1100 UDC USB Setup Specification.

may share data. Event handlers are non-blocking. As a result, *blocking behaviors are explicitly specified by states*. To remove a blocking operation from an event handler, the specification writer can restructure the state machine by splitting the handler at the operation and inserting a new state. Thus, we describe synchronization behaviors declaratively using states, rather than procedurally, which enables better checking. Specifically, interrupt processing is modeled as a couple of hardware event handlers.

### 4.3. Control function specification

A core function is responsible for providing the device services to the upper layers of software. As illustrated in Figure 6-1, core functions are responsible for managing data accesses and manipulating device states. Data accesses follow well-defined semantics, which specify, for example, that a block of data has to be transferred between the processor and device. Control functions are responsible for changing the control state of the device according to the state machines. As illustrated in the lower part of Figure 6-5, a *core control function* has an input event and a final state set. It accomplishes specific functions by triggering a sequence of transitions. Because a transition may emit multiple output events, multiple transitions can be enabled at one time.

The core control function selectively fires eligible transitions, i.e. finds a transition path from the current state to the final state set. For example, the *start* function of SA1100 UDC is defined as the union of *start* event and a set of states, as shown in Figure 6-5. When the application calls the start function, it generates the input event (service request) *start*. It then finds a transition path to states (usb\_protocol, ZombieSuspend), (ep1, idle), (ep2, idle) and completes. If the current state does not accept the *start* event, the function returns abnormally. Timeout is optionally defined to avoid infinite waiting.

Although a data flow does not have an explicit control state machine specified, a state machine is implicitly generated for it to manage the control states such as reset and blocking.

#### 4.4. Device programming interface specification

To model device register accesses, we use the concepts of *register* and *variable* (see Figure 6-6) – similar definitions can be found in Devil [3]. Event handlers and core functions access the device registers through a set of APIs such as *IUDC\_REG\_name\_READ* and *IUDC\_REG\_name\_WRITE* that use these registers and variables. These API's are synthesized from the register and variable specifications, and extend the virtual environment. In addition to a basic programming interface, we provide additional features that enable common access mechanisms to be described succinctly. For example, FIFO is a common device-programming interface. It is always accessed through a register. Different devices use different mechanisms to indicate the number of data in the FIFO. To enable the synthesis of FIFO access routines, we have defined the concept of a hardware FIFO mapped register that is not illustrated in detail here because of limited space.

```

## Eight bit register cs2 is at offset 0x18. The 6th and
## 7th bits of udccs2 are reserved and read as 0.
.reg cs2 = base[0x18] .mask <00----->
    .slow : 8 bit;

## Variable definition for register cs2: except for
## the two reserved bits, each bit of register cs2
## is a variable. We generate functions such as
## "set", "clear" for them.
.vars cs2 [ *, *, fst, sst, tur, tpe,
            tpc, tfs];

```

Figure 6-6. SA1100 UDC Transmit Channel Register and Variable Specification Example.

## 5. SYNTHESIS

Given all the parts of the specification, the synthesis process synthesizes the C code for the entire driver. The functions that need to be provided are the core functions that provide the device services to the upper layers of software. In synthesizing these functions, all parts of the specification are used. This section outlines the synthesis process.

### 5.1. Platform function and registry function mapping

The platform interface includes fundamental virtual data types and a basic virtual API for the following categories:

- a) Synchronization functions;
- b) Timer management functions;
- c) Memory management functions;
- d) DMA and bus access functions;
- e) Interrupt handling (setup, enable, disable, etc.);
- f) Tracing functions;
- g) Register/memory access functions.

All virtual data types and API are mapped into platform specific types and API by implementing them on the platform. This part is done manually based on an understanding of the platform. Note that while this is not synthesized, this approach provides for significant reuse as this needs to be done only once for each platform (or part of a platform).

We adopt a template-based approach to map the registry function interface to a specific platform by creating a library of platform specific templates. The synthesizer generates appropriate code to tailor the template for a particular device. Although the registry function generation is platform specific, it is reused for drivers for different devices.

### 5.2. Device core function synthesis

As illustrated in Figure 6-1, device driver core functions are either data access functions or control functions. We synthesize data access functions from the data path specification, and control functions from the device control specification.

Section 3 mentioned that driver core functions synchronize with the device through one of the 3 synchronization mechanisms: interrupt, poll and timer delay. As a result, our driver core function synthesizer synthesizes both the driver functions and the synchronization routines that are sufficient to completely define the platform independent part of the driver.

### 5.2.1. Synchronization mechanism

Let us consider the synchronization mechanisms first. An interrupt is essentially an asynchronous communication mechanism whereby the interrupt handler is called when the interrupt occurs. Both polling and timer delay are either asynchronous or blocking. For example, a timer delay can be a busy wait that is blocking. On the other hand, we can register a routine that is executed by the system when the timer expires which is an asynchronous behavior. For brevity, we only describe the synthesis of asynchronous communication routines (e.g., interrupt handlers for interrupts), which are critical to synchronization mechanism synthesis.

### 5.2.2. Data access function

A data access function in our framework is asynchronous: it does not wait for the completion of data access but returns right after the data access is enabled. This asynchronous behavior enables the overlap of data transfer and computation. A callback function can be registered to notify the completion of a data transfer. Synchronous communication is achieved by synchronizing the caller and the callback function.

A data path is modeled as a stream of data units. Hence, the data access function is implemented by iterating data unit accesses until the entire data block is transferred. If an error occurs, the whole data path access aborts. The device control can block or reset a data path.

### 5.2.3. Control function

A control function moves the device to a particular state. The execution of application software depends on such a state change. Hence, a control function is blocking (synchronous) in our framework, i.e., it will not return unless the final state set is reached. The control functions do not return values. The device is observed through output events, as illustrated in Section 4. We declare a variable for each output event. Device status is observed by reading such variables.

A control function involves a series of state transitions. It completes when the final state set is reached. Since our model is based on event driven finite state machines, a control function essentially consists of a sequence of event firings. The state machines are checked to see if the final state set of the control function is reached. If the control function is not time out and the final state set is reached, it returns successfully. Figure 6-7 shows the sketches of control functions.

### 5.2.4. Execution of EFSM

We schedule the finite state machines using a round-robin scheduling policy. The events are consumed in a First Come First Served manner. A multi-

```

void ctrl(DEV_PRIV_DS *priv,
          IEsmEvtVal val) {
    Enqueue the input event;
    while (true) {
        while(firable transitions exist){
            Fire the transition;
            if(final state set is reached)
                return;
        }
        block(time_out_value);
    }
}

```

Figure 6-7. Control function.

processor safe, non-preemptive queue is implemented. Driver functions are executed in the context of a process/thread. We view the execution of interrupt handlers as a special kernel thread. Hence, the essence of the core function synthesis is to distribute the event handler executions and state transitions to the interrupt handlers and driver functions. We dynamically make this decision by checking whether there is a blocked process/thread. If yes, the driver function does the work; otherwise, the interrupt handler does it. Figure 6-8 shows the sketches of interrupt handlers.

```

void interrupt() {
    clear the interrupt;
    Enqueue the hardware event;
    if (event queue was not empty)
        return;
    if (a process/thread is blocked) {
        unblock it and return;
    } else {
        while (firable transitions exist)
            Fire the transition;
    }
}

```

Figure 6-8. Interrupt handler.

## 6. CASE STUDY

The Intel StrongArm SA1100 [10] is one of the more popular micro-processors used in embedded systems. Its peripheral control module contains a universal serial bus (USB) endpoint controller. This USB device controller (UDC) operates at half-duplex with a baud rate of 12 Mbps. It supports three

endpoints: endpoint 0 (ep0) through which a USB host controls the UDC, endpoint 1 (ep1) which is the transmit FIFO and end point 2 (ep2) which is the receive FIFO.

The Linux UDC (USB device controller) driver (ported to Strong-Arm SA1100) implements USB device protocol [11] (data transfer and USB connection setup) through coordination with the UDC hardware. An Ethernet device driver interface has been implemented on top of the USB data flow.

To evaluate our methodology, we modeled UDC and synthesized a UDC Linux device driver that has an Ethernet driver interface and manages data transfer and USB connection setup. UDC ep1 is modeled as an OUT data path. Its data unit is a data packet of no more than 64 bytes transferred via DMA. Similarly UDC ep2 is modeled as an IN data path. The UDC control has 4 state machines: one state machine for each endpoint and one state machine managing the USB protocol state of the device. The state machines for data flow ep1 and ep2 are implicitly generated. Table 6-1 shows a comparison of code sizes for the original Linux driver, the specification in our framework and the final synthesized driver.

The reduction in code size is one measure of increased productivity. More importantly, the format and definition of the specification enables less experienced designers to relatively easily specify the device behavior. The declarative feature of our specification also enables easy synthesis of drivers of different styles. For example, the original UDC character device driver interface (without comments and blank lines) has 498 lines of code while our specification only requires a few extra lines. Furthermore, the code synthesized is only slightly bigger than the original code.

The correctness of the synthesized UDC driver is tested on a HP iPaq3600 handheld, a SA1100 based device. We setup a USB connection between the handheld and a 686 based desktop through a USB cradle. Familiar v0.5.3 of Linux kernel 2.4.17 is installed on the iPaq and RedHat 7.2 of Linux kernel 2.4.9 is installed on the desktop. We compiled the synthesized code to a loadable kernel module with a cross-compiler for ARM, loaded the newly created module on the iPaq, bound IP socket layer over our module and successfully tested the correctness with the standard TCP/IP command *ping*.

## 7. CONCLUSIONS

Device driver development has traditionally been cumbersome and error prone. This trend is only worsening with IP based embedded system design where

Table 6-1. Comparison of code size.

	Linux UDC driver	Iris specification	Driver synthesized
Line count	2002	585	2157

different devices need to be used with different platforms. The paper presents a methodology and a tool for the development of device drivers that addresses the complexity and portability issues. A platform independent specification is used to synthesize platform independent driver code, which is then mapped to a specific platform using platform specific library functions. The latter need to be developed only once for each component of the platform and can be heavily reused between different drivers. We believe this methodology greatly simplifies driver development as it makes it possible for developers to provide this specification and then leverage the synthesis procedures as well as the library code reuse to derive significant productivity benefits.

## REFERENCES

1. F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*, Kluwer Academic Press, June 1997.
2. I. Bolsen, H. J. De Man, B. Lin, K. van Rompaey, S. Vercauteren, and D. Verkest. "Hardware/Software Co-design of Digital Telecommunication Systems", *Proceeding of the IEEE*, Vol. 85, No. 3, pp. 391–418, 1997.
3. F. Merillon, L. Reveillere, C. Consel, R. Marlet, and G. Muller. "Devil: An IDL for Hardware Programming." 4th Symposium on Operating Systems Design and Implementation, San Diego, October 2000, pp. 17–30.
4. M. O'Bils and A. Jantsch. "Device Driver and DMA Controller Synthesis from HW/SW Communication Protocol Specifications." *Design Automation for Embedded Systems*, Vol. 6, No. 2, pp. 177–205, Kluwer Academic Publishers, April 2001.
5. E. A. Lee. "Embedded Software," to appear in *Advances in Computers* (M. Zelkowitz, editor), Vol. 56, Academic Press, London, 2002.
6. <http://www.aisysinc.com>, November 200.
7. <http://www.microsoft.com/winhec/sessions2001/DriverDev.htm>, March 2002.
8. <http://www.intelligent-io.com>, July 2002.
9. <http://www.projectudi.org>, July 2002.
10. <http://www.intel.com/design/strong/manuals/27828806.pdf>, October 2002.
11. <http://www.usb.org/developers/docs.html>, July 2002.

## Chapter 7

# SYSTEMATIC EMBEDDED SOFTWARE GENERATION FROM SYSTEMC

F. Herrera, H. Posadas, P. Sánchez and E. Villar

*TEISA Dept., E.T.S.I. Industriales y Telecom., University of Cantabria, Avda. Los Castros s/n, 39005 Santander, Spain; E-mail: {fherrera, posadash, sanchez, villar}@teisa.unican.es*

**Abstract.** The embedded software design cost represents an important percentage of the embedded-system development costs [1]. This paper presents a method for systematic embedded software generation that reduces the software generation cost in a platform-based HW/SW codesign methodology for embedded systems based on SystemC. The goal is that the same SystemC code allows system-level specification and verification, and, after SW/HW partition, SW/HW co-simulation and embedded software generation. The C++ code for the SW partition (processes and process communication including HW/SW interfaces) is systematically generated including the user-selected embedded OS (e.g.: the eCos open source OS).

**Key words:** Embedded Software generation, SystemC, system-level design, platform based design

## 1. INTRODUCTION

The evolution of technological processes maintains its exponential growth; 810 Mtrans/chip in 2003 will become 2041 Mtrans/chip in 2007. This obliges an increase in the designer productivity, from 2.6 Mtrans/py in 2004 to 5.9 Mtrans/py in 2007, that is, a productivity increase of 236% in three years [2]. Most of these new products will be embedded System-on-Chip (SoC) [3] and include embedded software. In fact, embedded software now routinely accounts for 80% of embedded system development costs [1].

Today, most embedded systems are designed from a RT level description for the HW part and the embedded software code separately. Using a classical top-down methodology (synthesis and compilation) the implementation is obtained. The 2001 International Technology Roadmap for Semiconductors (ITRS) predicts the substitution (during the coming years) of that waterfall methodology by an integrated framework where codesign, logical, physical and analysis tools operate together. The design step being where the designer envisages the whole set of intended characteristics of the system to be implemented, system-level specification acquires a key importance in this new

---

This work has been partially supported by the Spanish MCYT through the TIC-2002-00660 project.

design process since it is taken as the starting point of all the integrated tools and procedures that lead to an optimal implementation [1, 2].

The lack of a unifying system specification language has been identified as one of the main obstacles bedeviling SoC designers [4]. Among the different possibilities proposed, languages based on C/C++ are gaining a wider-consensus among the designer community [5], SystemC being one of the most promising proposals. Although, the first versions of SystemC were focused on HW design, the latest versions (SystemC2.x [6]) include some system-level oriented constructs such as communication channels or process synchronization primitives that facilitate the system specification independently of the final module implementation in the HW or SW partition.

Embedded SW generation and interface synthesis are still open problems requiring further research [7]. In order to become a practical system-level specification language, efficient SW generation and interface synthesis from SystemC should be provided. Several approaches for embedded SW design have been proposed [8–10]. Some of them are application-oriented (DSP, control, systems, etc.), where others utilise input language of limited use.

SoCOS [11, 12] is a C++ based system-level design environment where emphasis is placed on the inclusion of typical SW dynamic elements and concurrency. Nevertheless, SoCOS is only used for system modeling, analysis and simulation.

A different alternative is based on the synthesis of an application-specific RTOS [13–15] that supports the embedded software. The specificity of the generated RTOS gives efficiency [16] at the expense of a loss of verification and debugging capability, platform portability and support for application software (non firmware). Only very recently, the first HW/SW co-design tools based on C/C++-like input language have appeared in the marketplace [17]. Nevertheless, their system level modeling capability is very limited.

In this paper, an efficient embedded software and interface generation methodology from SystemC is presented. HW generation and cosimulation are not the subject of this paper. The proposed methodology is based on the redefinition and overloading of SystemC class library elements. The original code of these elements calls the SystemC kernel functions to support process concurrency and communication. The new code (defined in an implementation library) calls the embedded RTOS functions that implement the equivalent functionality. Thus, SystemC kernel functions are replaced by typical RTOS functions in the generated software. The embedded system description is not modified during the software and interface generation process. The proposed approach is independent of the embedded RTOS. This allows the designer to select the commercial or open source OS that best matches the system requirements. In fact, the proposed methodology even supports the use of an application-specific OS.

The contents of the paper are as follows. In this section, the state of the art, motivation and objectives of the work have been presented. In section 2, the system-level specification methodology is briefly explained in order to

show the input description style of the proposed method. In section 3, the embedded SW generation and communication channel implementation methodology will be described. In section 4, some experimental results will be provided. Finally, the conclusions will be drawn in section 5.

## 2. SPECIFICATION METHODOLOGY

Our design methodology follows the ITRS predictions toward the integration of the system-level specification in the design process. SystemC has been chosen as a suitable language supporting the fundamental features required for system-level specification (concurrency, reactivity, . . .).

The main elements of the proposed system specification are processes, interfaces, channels, modules and ports. The system is composed of a set of asynchronous, reactive processes that concurrently perform the system functionality. Inside the process code no event object is supported. As a consequence, the *notify* or *wait* primitives are not allowed except for the “timing” wait, *wait(sc\_time)*. No signal can be used and processes lack a sensitivity list. Therefore, a process will only block when it reaches a “timing” wait or a *wait* on event statement inside a communication channel access. All the processes start execution with the sentence *sc\_start()* in the function *sc\_main()*. A process will terminate execution when it reaches its associated end of function.

The orthogonality between communication and process functionality is a key concept in order to obtain a feasible and efficient implementation. To achieve this, processes communicate among themselves by means of channels. The channels are the implementation of the behavior of communication interfaces (a set of methods that the process can access to communicate with other processes). The behavior determines the synchronization and data transfer procedures when the access method is executed. For the channel description at the specification level it is possible to use *wait* and *notify* primitives. In addition, it is necessary to provide platform implementations of each channel. The platform supplier and occasionally the specifier should provide this code. The greater the number of appropriate implementations for these communication channels on the platform, the greater the number of partition possibilities, thus improving the resulting system implementation.

Figure 7-1 shows a process representation graph composed of four kinds of nodes. The process will resume in a start node and will eventually terminate in a finish node. The third kind of node is an internal node containing timing wait statements. The fourth kind of node is the channel method access node. The segments are simply those code paths where the process executes without blocking.

Hierarchy is supported since processes can be grouped within modules. Following the IP reuse-oriented design recommendations for intermodule communications, port objects are also included. Therefore, communication

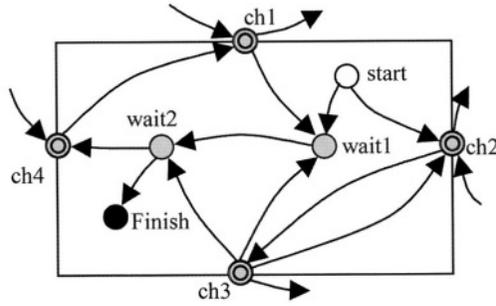


Figure 7-1. Process representation.

among processes of different module instances passes through ports. Port and module generation is static. Granularity is established at the module level and a module may contain as many processes as needed. Communication inside the inner modules must also be performed through channels.

The previously commented set of restrictions on how SystemC can be used as a system specification language do not constrain the designer in the specification of the structure and functionality of any complex system. Moreover, as the specification methodology imposes a clear separation between computation and communication, it greatly facilitates the capture of the behavior and structure of any complex system ensuring a reliable and efficient co-design flow.

### 3. SOFTWARE GENERATION

Today, most industrial embedded software is manually generated from the system specification, after SW/HW partition. This software code includes several RTOS function calls in order to support process concurrency and synchronization. If this code is compared with the equivalent SystemC description of the module a very high correlation between them is observed. There is a very close relationship between the RTOS and the SystemC kernel functions that support concurrency. Concerning interface implementation, the relationship is not so direct. SystemC channels normally use *notify* and *wait* constructions to synchronize the data transfer and process execution while the RTOS normally supports several different mechanisms for these tasks (interruption, mutex, flags, . . .). Thus, every SystemC channel can be implemented with different RTOS functions [18].

The proposed software generation method is based on that correlation. Thus, the main idea is that the embedded software can be systematically generated by simply replacing some SystemC library elements by behaviourally equivalent procedures based on RTOS functions. It is a responsibility of the platform designer to ensure the required equivalence between the SystemC

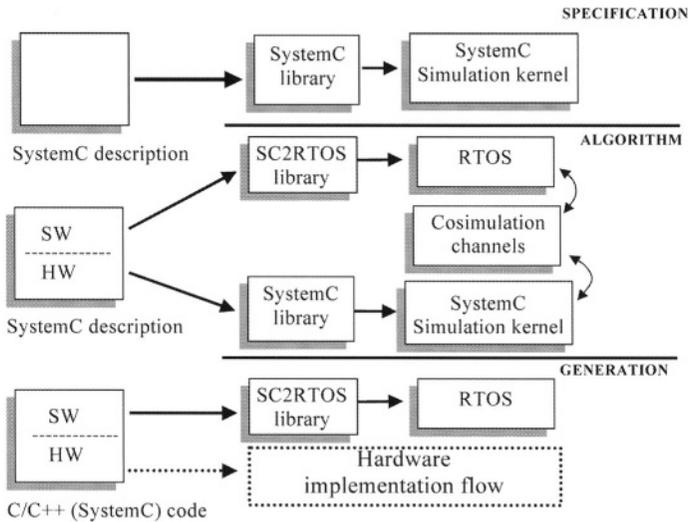


Figure 7-2. Proposed SW generation flow.

functions and their implementation. Figure 7-2 shows the proposed software generation flow. The design process begins from a SystemC specification. This description verifies the specification methodology proposed in the previous section. At this level (“Specification”) SW/HW partition has not been performed yet. In order to simulate the “Specification level” description, the code has to include the SystemC class library (“systemc.h” file) that describes the SystemC simulation kernel.

After partition, the system is described in terms of SW and HW algorithms (“Algorithmic level”). At this level, the code that supports some SystemC constructions in the SW part has to be modified. This replacement is performed at library level, also so it is totally hidden from the designer who sees these libraries as black boxes. Thus, the SystemC user code (now pure C/C++ code) is not modified during the software generation flow, constituting one of the most important advantages of this approach. A new library SC2RTOS (SystemC to RTOS) redefines the SystemC constructions whose definition has to be modified in the SW part. It is very important to highlight that the number of SystemC elements that have to be redefined is very small.

Table 7-1 shows these elements. They are classified in terms of the element functionality. The table also shows the type of RTOS function that replaces the SystemC elements. The specific function depends on the selected RTOS. The library could be made independent of the RTOS by using a generic API (e.g.: EL/IX [19]).

At algorithmic level, the system can be co-simulated. In order to do that, an RTOS-to-SystemC kernel interface is needed. This interface models the relation between the RTOS and the underlying HW platform (running both over the same host).

Table 7-1. SystemC elements replaced.

	Hierarchy	Concurrency	Communication
SystemC Elements	SC_MODULE SC_CTOR sc_module sc_module_name	SC_THREAD SC_HAS_PROCESS sc_start	wait(sc_time) sc_time sc_time_unit sc_interface sc_channel sc_port
RTOS Functions		Thread management	Synchronization management Timer management Interruption management Memory access

Another application of the partitioned description is the objective of this paper: software generation. In this case (“Generation” level in Figure 7-2), only the code of the SW part has to be included in the generated software. Thus, the code of the SystemC constructions of the HW part has to be re-defined in such a way that the compiler easily and efficiently eliminates them.

The analysis of the proposed SW generation flow concludes that the SystemC constructions have to be replaced by different elements depending on the former levels (Specification, Algorithm or Generation) and the partition (SW or HW) in which the SystemC code is implemented. This idea is presented in Figure 7-3:

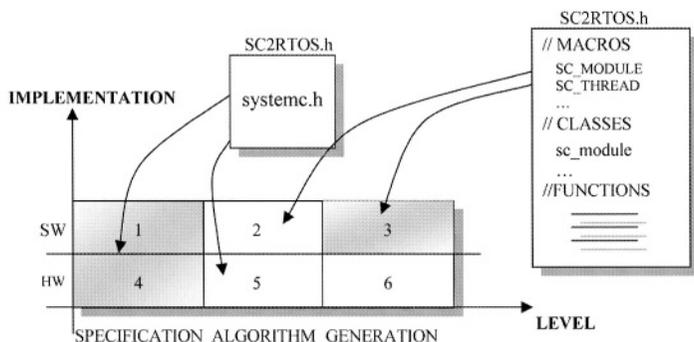


Figure 7-3. SC2RTOS library implementation.

Thus, there are 6 possible implementations of the SystemC constructions. In the proposed approach all these implementations are included in a file (SC2RTOS.h) whose content depends on the values of the LEVEL and IMPLEMENTATION variables. For example, at SPECIFICATION level only the LEVEL variable is taken into account (options 1 and 4 are equal) and the SC2RTOS file only includes the SystemC standard include file (systemc.h).

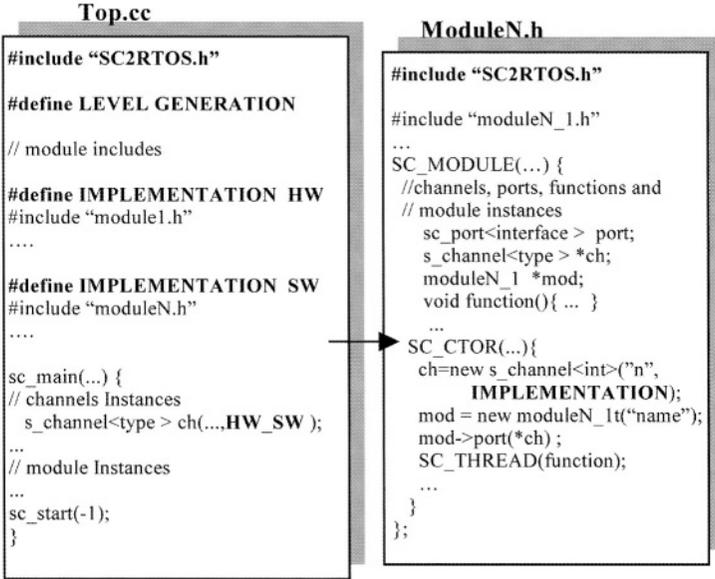


Figure 7-4. SystemC description example.

However, in option 3, for example, the SC2RTOS file redefines the SystemC constructions in order to insert the RTOS. Figure 7-4 presents an example (a SystemC description) that shows these concepts.

The “#define” preprocessing directives will be introduced by the partitioning tool. These directives are ignored at specification level. In this paper, it is assumed that the platform has only one processor, the partition is performed at module level in this top hierarchical module and every module is assigned to only one partition. Thus, only the modules instantiated in the top module are assigned to the hardware or software partition. Hierarchical modules are assigned to the same parent-module partition. Only the channels instantiated in the top hierarchical module can communicate processes assigned to different partitions. The channels instantiated inside the hierarchy will communicate processes that are assigned to the same partition. This is a consequence of the assignment of the hierarchical modules.

In Figure 7-4, the top hierarchical module (described in file *top.cc*) and one of the hierarchical modules (described in file *ModuleN.h*) are presented. All the statements that introduce partition information in these descriptions have been highlighted with bold fonts. The LEVEL and IMPLEMENTATION variables are specified with pre-processor statements. With a little code modification, these variables could be defined, for example, with the compiler command line options. In this case, the partition decisions will only affect the compiler options and the SystemC user code will not be modified during the software generation process.

Before a module is declared, the library “SC2RTOS.h” has to be included. This allows the compiler to use the correct implementation of the SystemC constructions in every module declaration. Concerning the channels, an additional argument has been introduced in the channel declaration. This argument specifies the type of communication (HW\_SW, SW\_HW, SW or HW) of a particular instance of the channel. This parameter is used to determine the correct channel implementation. The current version of the proposed software generation method is not able to determine automatically the partitions of the processes that a particular instance of a channel communicates, thus this information must be explicitly provided.

#### 4. APPLICATION EXAMPLE

In order to evaluate the proposed technique a simple design, a car Anti-lock Braking System (ABS) [17] example, is presented in this section. The system description has about 200 SystemC code lines and it includes 5 modules with 6 concurrent processes.

The system has been implemented in an ARM-based platform that includes an ARM7TDMI processor, 1 Mb RAM, 1 Mb Flash, two 200 Kgate FPGAs, a little configuration CPLD and an AMBA bus. The open source eCos operating system has been selected as embedded RTOS. In order to generate software for this platform-OS pair, a SystemC-to-eCos Library has to be defined. This library is application independent and it will allow the generation of the embedded software for that platform with the eCos RTOS.

The SC2ECOS (SystemC-to-eCos) Library has about 800 C++ code lines and it basically includes the concurrency and communication support classes that replace the SystemC kernel. This library can be easily adapted to a different RTOS.

The main non-visible elements of the concurrency support are the *uc\_thread* and *exec\_context* classes. The *uc\_thread* class maintains the set of elements that an eCos thread needs for its declaration and execution. The *exec\_context* class replaces the SystemC *sc\_simcontext* class during software generation. It manages the list of declared processes and its resumption. These elements call only 4 functions of eCos (see Table 7-2):

Table 7-2. eCos functions called by the SC2eCos library.

	Thread management	Synchronization management	Interruption management
eCos Functions	cyg_thread_create cyg_thread_resume cyg_user_start cyg_thread_delay	cyg_flag_mask_bits cyg_flag_set_bits cyg_flag_wait	cyg_interrupt_create cyg_interrupt_attach cyg_interrupt_acknowledge cyg_interrupt_unmask

In order to allow communication among processes, several channel models have been defined in the SC2ECOS library. The ABS application example uses two of them; a blocking channel (one element `sc_fifo` channel) and an extended channel (that enables blocking, non-blocking and polling accesses).

The same channel type could communicate SW processes, HW processes or a HW and a SW process (or vice versa). Thus, the proposed channel models have different implementations depending on the HW/SW partition. In order to implement these channel models only 7 eCos functions have been called (center and right columns of Table 7-2). These eCos functions control the synchronization and data transfer between processes.

The ABS system has 5 modules: the top (“Abs\_system”), 2 modules that compute the speed and the acceleration (“Compute Speed” and “Compute Acceleration”), another (“Take decision”) that decides the braking, and the last one (“Multiple Reader”) that provides speed data to the “Compute Acceleration” and “Take Decision” modules. Several experiments, with different partitions have been performed.

Several conclusions have been obtained from the analysis of the experimental results shown in Figure 7-5:

- There is a minimum memory size of 53.2 Kb that can be considered constant (independent of the application). This fixed component is divided in two parts: a 31 K contribution due to the default configuration of the eCos kernel, that includes the scheduler, interruptions, timer, priorities, monitor and debugging capabilities and a 22.2 Kb contribution, necessary to support dynamic memory management, as the C++ library makes use of it.
- There is a variable component of the memory size that can be divided into two different contributions. One of them is due to the channel implementation and asymptotically increases to a limit. This growth is non-linear due to the compiler optimizing the overlap of necessary resources among the

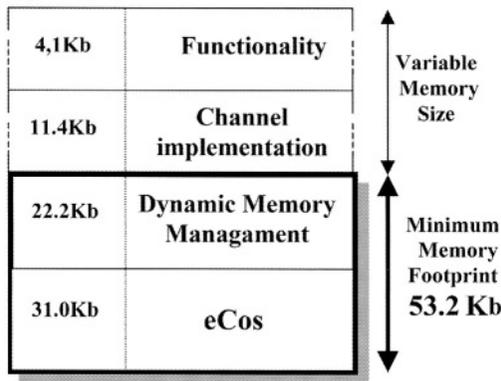


Figure 7-5. Memory footprint for the ABS SW implementation.

new channel implementations. It reaches a limit depending on the number of channel implementations given for the platform. Thus, each platform enables a different limit. The channel implementation of the ABS system requires 11.4 Kb. The other component depends on the functionality and module specification code. This growth can be considered linear (at a rate of approximately 1 Kb per 50–70 source code lines in the example), as can be deduced from Table 7-3.

The total software code generated is limited to 68.7 Kb.

Table 7-3 shows in more detail how the functionality size component in the ABS system is distributed in each module:

Table 7-3. Code size introduced by each module.

SC_MODULE	Abs System	Compute Speed	Compute Acceleration	Take Decision	Multiple Readers
C++ Code lines	30	44	46	110	12
Blocking channels	3	0	0	1	0
Extended channels	1	0	0	0	0
Generated SW	1130 bytes	700 bytes	704 bytes	3510 bytes	140 bytes

The addition of module memory sizes gives the value of 6.2 Kb. This exceeds by 2.1 Kb the 4.2 Kb shown in Figure 7-5. This is due to the shared code of instanced channels that appears in modules including structural description (Take Decision and ABS System).

The overall overhead introduced by the method respect to manual development is negligible because the SystemC code is not included but substituted by a C++ equivalent implementation that uses eCos. In terms of memory, only the 22.2 Kbytes could be reduced if an alternative C code avoiding dynamic memory management is written.

## 5. CONCLUSIONS

This paper presents an efficient embedded software generation method based on SystemC. This technique reduces the embedded system design cost in a platform based HW/SW codesign methodology. One of its main advantages is that the same SystemC code is used for the system-level specification and, after SW/HW partition, for the embedded SW generation. The proposed methodology is based on the redefinition and overloading of SystemC class library construction elements. In the software generated, those elements are replaced by typical RTOS functions. Another advantage is that this method is independent of the selected RTOS and any of them can be supported by simply writing the corresponding library for that replacement. Experimental results demonstrate that the minimum memory footprint is 53.2 Kb when the

eCos RTOS is used. This overhead is relatively low taking into account the great advantages that it offers: it enables the support of SystemC as unaltered input for the methodology processes and gives reliable support of a robust and application independent RTOS, namely eCos, that includes an extensive support for debugging, monitoring, etc. . . . To this non-recurrent SW code, the minimum footprint has to be increased with a variable but limited component due to the channels. Beyond this, there is a linear size increment with functionality and hierarchy complexity of the specification code.

## REFERENCES

1. A. Allan, D. Edenfeld, W. Joyner, A. Kahng, M. Rodgers, and Y. Zorian. "2001 Technology Roadmap for Semiconductors". *IEEE Computer*, January 2002.
2. International Technology Roadmap for Semiconductors. 2001 Update. *Design*. Editions at <http://public.itrs.net>.
3. J. Borel. "SoC design challenges: The EDA Medea Roadmap." In E. Villar (ed.), *Design of HW/SW Embedded Systems*. University of Cantabria. 2001.
4. L. Geppert. "Electronic design automation." *IEEE Spectrum*, Vol. 37, No. 1, January 2000.
5. G. Prophet. "System Level Design Languages: to C or not to C?." *EDN Europe*. October 1999. [www.ednmag.com](http://www.ednmag.com).
6. "SystemC 2.0 Functional specification", [www.systemc.org](http://www.systemc.org), 2001.
7. P. Sánchez. "Embedded SW and RTOS." In E. Villar (ed.), *Design of HW/SW Embedded Systems*. University of Cantabria. 2001.
8. PtolemyII. <http://ptolemy.eecs.berkeley.edu/ptolemyII>.
9. F. Baladin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vicentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-Software Co-design of Embedded Systems: The POLIS Approach*. Kluwer. 1997.
10. R. K. Gupta. "Co-synthesis of Hardware and Software for Digital Embedded Systems." Ed. Kluwer. August 1995. ISBN 0-7923-9613-8.
11. D. Desmet, D. Verkest, and H. de Man. "Operating System Based Software Generation for System-On-Chip." *Proceedings of Design Automation Conference*, June 2000.
12. D. Verkest, J. da Silva, C. Ykman, K. C. Roes, M. Miranda, S. Wuytack, G. de Jong, F. Catthoor, and H. de Man. "Matisse: A System-on-Chip Design Methodology Emphasizing Dynamic Memory Management." *Journal of VLSI signal Processing*, Vol. 21, No. 3, pp. 277–291, July 1999.
13. M. Diaz-Nava, W. Cesário, A. Baghdadi, L. Gauthier, D. Lyonard, G. Nicolescu, Y. Paviot, S. Yoo, and A. A. Jerraya. "Component-Based Design Approach for Multicore SoCs." *Proceedings of Design Automation Conference*, June 2002.
14. L. Gauthier, S. Yoo, and A. A. Jerraya. "Automatic Generation of Application-Specific Operating Systems and Embedded Systems Software." *Proceedings of Design Automation and Test in Europe*, March 2001.
15. S. Yoo; G. Nicolescu; L. Gauthier, and A. A. Jerraya: "Automatic Generation of fast Simulation Models for Operating Systems in SoC Design." *Proceedings of DATE'02*, IEEE Computer Society Press, 2002.
16. K. Weiss, T. Steckstor, and W. Rosenstiel. "Emulation of a Fast Reactive Embedded System using a Real Time Operating System." *Proceedings of DATE'99*. March 1999.
17. Coware, Inc., "N2C", <http://www.coware.com/N2C.html>.
18. F. Herrera, P. Sánchez, and E. Villar. "HW/SW Interface Implementation from SystemC for Platform Based Design." *FDL'02*. Marseille. September 2002.
19. EL/IX Base API Specification DRAFT. <http://sources.redhat.com/elix/api/current/api.html>.

*This page intentionally left blank*

PART III:

EMBEDDED SOFTWARE DESIGN AND IMPLEMENTATION

*This page intentionally left blank*

## Chapter 8

# EXPLORING SW PERFORMANCE USING SOC TRANSACTION-LEVEL MODELING

Imed Moussa, Thierry Grellier and Giang Nguyen

**Abstract.** This paper presents a Virtual Instrumentation for System Transaction (VISTA), a new methodology and tool dedicated to analyse system level performance by executing full-scale SW application code on a transaction-level model of the SoC platform. The SoC provider provides a cycle-accurate functional model of the SoC architecture using the basic SystemC Transaction Level Modeling (TLM) components provided by VISTA : bus models, memories, IPs, CPUs, and RTOS generic services. These components have been carefully designed to be integrated into a SoC design flow with an implementation path for automatic generation of IP HW interfaces and SW device drivers. The application developer can then integrate the application code onto the SoC architecture as a set of SystemC modules. VISTA supports cross-compilation on the target processor and back annotation, therefore bypassing the use of an ISS. We illustrate the features of VISTA through the design and simulation of an MPEG video decoder application.

**Key words:** Transaction Level Modeling, SoC, SystemC, performance analysis

### 1. INTRODUCTION

One of the reasons for the focus on SW is the lagging SW design productivity compared to rising complexity. Although SoCs and board-level designs share the general trend toward using software for flexibility, the criticality of the software reuse problem is much worse with SoCs. The functions required of these embedded systems have increased markedly in complexity, and the number of functions is growing just as fast. Coupled with quickly changing design specifications, these trends have made it very difficult to predict development cycle time. Meanwhile, the traditional embedded systems software industry has so far not addressed issues of “hard constraints,” such as reaction speed, memory footprint and power consumption, because they are relatively unimportant for traditional, board-level development systems [1–4]. But, these issues are critical for embedded software running on SoCs. VISTA is targeted to address the system-level design needs and the SW design reuse needs for SoC design.

## 2. VISTA METHODOLOGY APPROACH

System level architects and application SW developers look for performance analysis and overall behavior of the system. They do not necessarily need and cannot make use of a cycle-accurate model of the SoC platform. However, a pure un-timed C model is not satisfactory either since some timing notions will still be required for performance analysis or power estimation. The capability of VISTA to bring HW/SW SoC modeling and characterization to the fore is key as it is shown in Figure 8-1.

VISTA can be decomposed into at least two major use paradigms:

1. Creation of the SoC virtual platform for system analysis and architecture exploration.
2. Use of the SoC virtual platform for SW development and system analysis by the systems houses SW or system designers.

Before system performance analysis, the designer has to leverage the existing HW library delivered with VISTA library to create the appropriate SoC platform architecture. The generic VISTA elements delivered in the library include: Bus (STBus, AHB/APB), Memories, Peripherals (timers, DMA, IOs, etc.) and RTOS elements.

Using the provided virtual model of the SoC, including the hardware abstraction layer and the RTOS layer, system designers and application SW designers can use VISTA to simulate and analyze system performance in terms

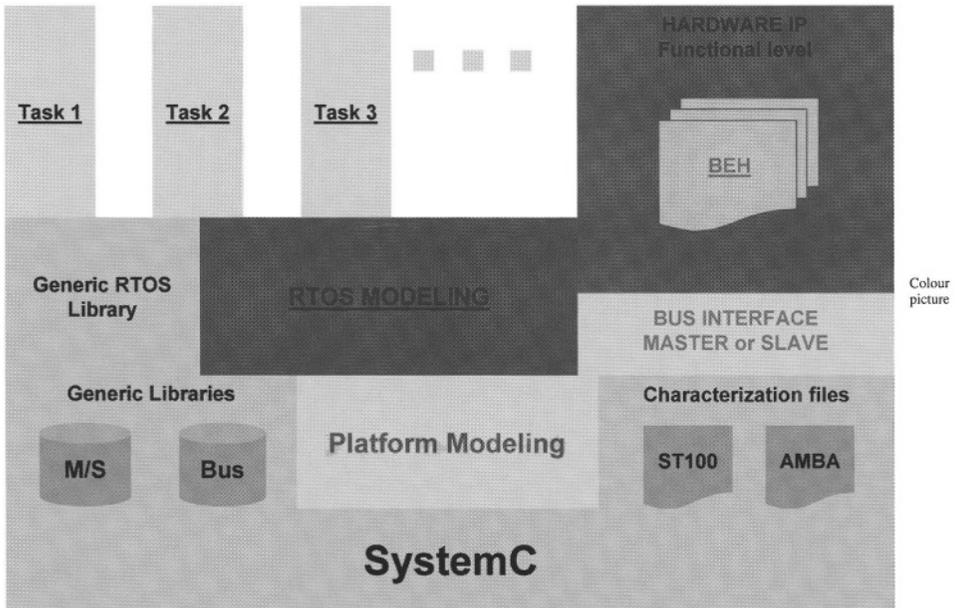


Figure 8-1. VISTA design and analysis flow.

of latency, bus loading, memory accesses, arbitration policy, tasks activity. These parameters allow for exploring SW performance as it can be seen for the video decoder application presented in the next section.

The first step during the design flow is to create the application tasks in C language. Next, using VISTA graphical front-end, the applications tasks and RTOS resources can be allocated on the “black-box”. Preliminary simulation can now be performed by executing the application code on the virtual platform to ensure that the code is functionally correct. Then, the code is cross-compiled on the target processor. VISTA will process the output from the cross-compilation phase in order to create timing information, which can be back annotated on the original application code to reflect *in situ* execution of the code on the target platform. Finally, timed simulation can be performed to analyze for system performance and power consumption. The information extracted from this phase can enable the SW developer to further develop and refine the application code.

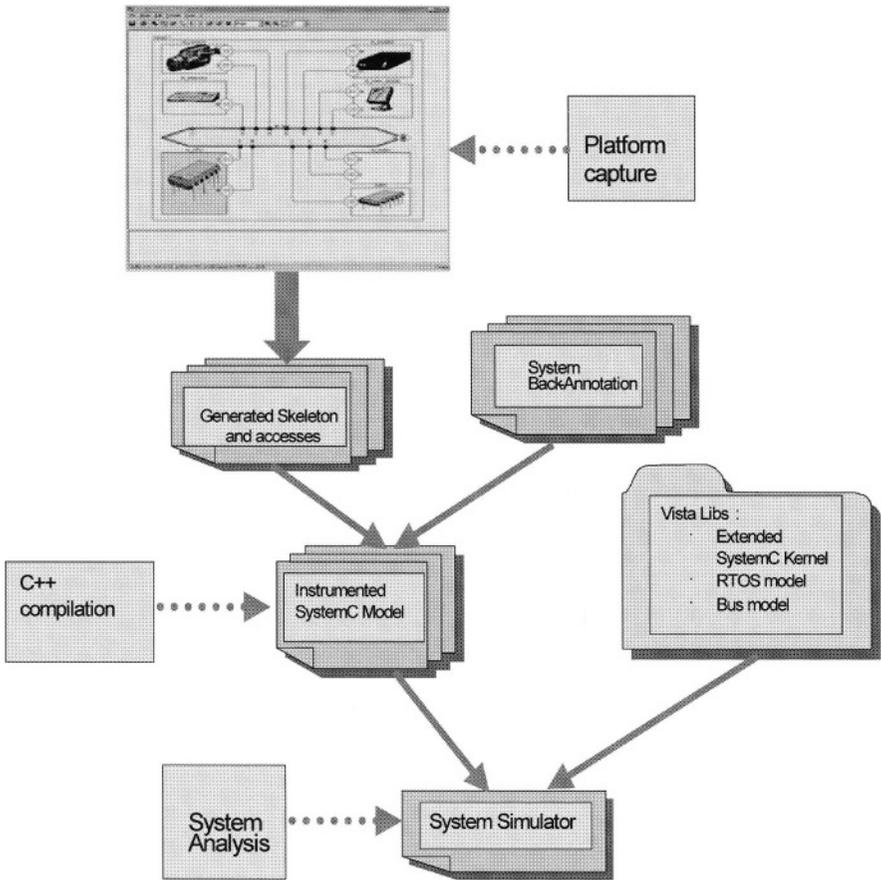
VISTA’s simulator is a compiled C++ program that takes advantage of the SystemC2.0 kernel, modified so as to get proper debugging and tracing information. The simulator is launched either as a stand-alone process or as a dynamic library loaded into the process of the GUI. Ideally both options should be available as option one is easier to debug and option two is faster. Several simulation modes are possible. First, interactive simulation: the user can set graphical input devices on ports of the diagram, like sliders, knobs, etc. Outputs can be monitored similarly by setting waveform viewers like the one shown in Figure 8-2, which gathers in one single window the functionality of an oscilloscope (continuous traces) and of a logic analyzer (discrete traces). Interactive simulation is essential in the prototyping phases of a software development. Secondly, batch simulation will also be possible.

## 2.1. Abstract communication

The accesses are a mechanism to abstract the communication between the modules and allow a seamless refinement of the communication into the IP modules from the functional level to the transactional level. SystemC2.0 already provides ports and channels for this purpose. We have enriched the channel notion with introducing a pattern of three related modules:

### 2.1.1. Channels models

1. *The slave access* first publishes the operations to the channel. It is notified the channel transactions corresponding to these operation invocations, and accepts or rejects them. For example, the amba slave access notifies the amba bus channel whether a transaction is ok, fails or needs to be split. Note that the publishing mechanism makes that a slave access doesn’t need to reproduce the slave module interfaces. A slave access can be shared by several modules, but can only be bound to a single channel.



Colour picture

Figure 8.2. VISTA design and analysis flow.

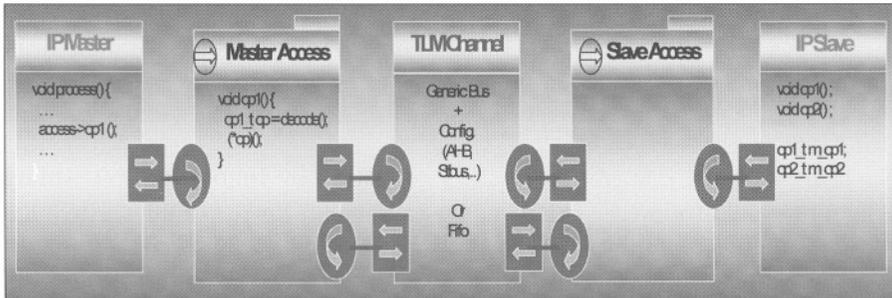
2. *The channel* is a pure transactional model of the communication. It guards the actual invocation of the slave operation until the communication transactions are completed. Data are not circulating through the channel, so far they are transmitted within the parameters of the operation. So only the relevant information for an arbitration are given, such as the amount of bytes to transmit, and the word size. If we refer to the simple bus transactional interface defined in [5], a vista channel interface is quite similar, only the parameters regarding the data transmission are discarded. The channel can also contain a runtime modifiable configuration to analyse a system performance. For example a bus channel can have a configurable cycle rate, a FIFO channel may have a configurable depth. It may also contain some measures from which revealing statistics can be extracted or computed within the channel model.
3. *The master access* is used to encapsulate the invocation of a slave

operation through a channel. Each encapsulated operation is called in two steps, the first step is the decode phase to identify the serving slave operation, and then the invocation of the operation through the channel. This invocation adds new parameters to the slave operation which configures the channel transactions to be generated. Note that a master access only addresses a single channel, and that it can invoke a subset of the operations of several slaves, so that it has its own interface which may differ from the union of the slaves interfaces and doesn't require to be an explicit `sc_interface`.

2.1.2. Communication refinement

Defining the accesses module is very straight forward. They follow a simple pattern. This is why our tool is able of generating the accesses code with very little configuration information once the communication channel is chosen. Figure 8-3 shows the example of accesses to a bus communication channel. It combines UML class diagrams to describe modules and SystemC2.0 [6] graphical notation for ports and interfaces. Furthermore a graphical notation has been introduced to stereotype the accesses modules. It can be used as a full replacement to the class diagram. Both the master and the slave accesses can be customized by a policy. In the case of the master access, this policy indicates whether the two concurrent operation calls by the master IP are serialized or have interlaced transactions within the access. The channel is indeed only capable of managing the concurrency between master accesses, but remains a concurrency between the threads of the modules sharing a master access. The slave access policy determines the concurrency on the operations provided by the slave IP. Several masters can generate several interleaved transactions to the same slave, especially when the slave splits the transactions, requiring the slave to have a customizable concurrency policy.

At last, one may think of composing channels. For example, a slave IP can be first attached to a FIFO channel which master access could then become



Colour picture

Figure 8-3. Abstract communication access.

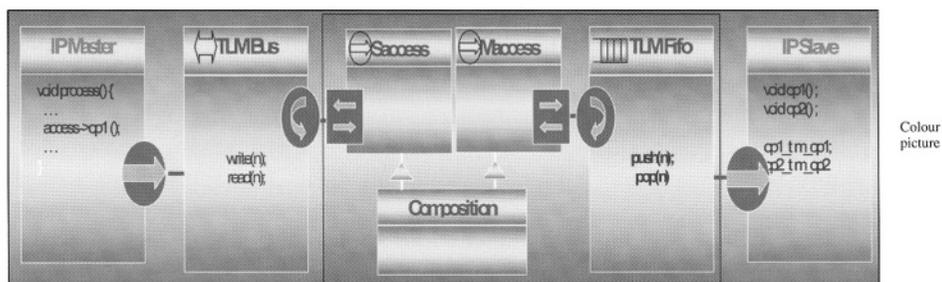


Figure 8-4. Channel composition: Master and slave accesses composition.

the slave access of a bus channel as show on Figure 8-4. These compositions can be the beginning of a methodology to refine the communication of a system and progressively going from a purely functional model to an architectural timed model. A first step would be to detach some functions of the sequential algorithm into modules that prefigures the architecture. Then regarding the iteration on the data, between the functions, one may start ahead the detached function as soon as it has enough data to start with, for example a subset of a picture. This is quite the same idea of setting a pipeline communication, so one may then prepare a communication through a FIFO channel to synchronize the execution flow of the detached functions. Note that the FIFO can also store control so far these a operation calls which are stacked. At this stage we a system level model very similar to what we can have with SDL. We can introduce timing at this stage to analyze the parallelism of the all system. Then one may decide the partitioning of the system and thus share the communication resources. Hence a bus can refine the FIFO communications, some data passing can be referring a shared memory. This is generally the impact of hardware level resource sharing which is ignored by the other specification languages which stops their analysis at the FIFO level.

### 2.1.3. Dynamic pattern

The Figure 8.5 shows how the protocol between the channel and its master and slave accesses is played.

A slave registers, at the end of elaboration phase, its provided operations to the channel through the slave access. These operations are identified with a key made of an address, and an optional qualifier, if the address is overloaded by the slave.

Then a master can initiate a slave operation by invoking the sibling operation of its master access. The master access retrieves first the operation in the decoding step and then notify the channel to execute the pre-transactions, before the channel allows it to effectively reach the slave operation. Once the guarded operation is done, the master access notifies the post-transaction transactions.

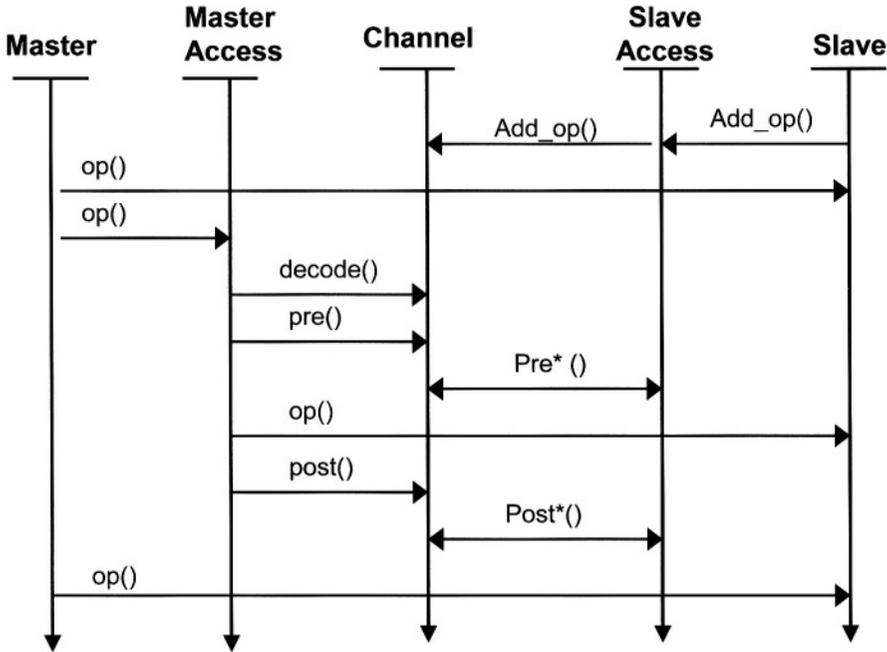


Figure 8-5. Channel protocol pattern.

The channel may use a subsequent protocol with the slave to play the pre and post transactions.

In the case of the AMBA bus, the `pre()` and `post()` are respectively corresponding to management of the write transactions giving the in parameters of the operations and the read transactions to retrieve the out parameters of the operations.

Figure 8-6 shows how the pattern has been applied for the AMBA-bus. The `write_txs()` and `read_txs` give the user provided data describing the transaction configuration to the channel. These data notably contains the number of bytes to be transmitted, and the width of the words. They do not contain the actual data transmitted to the slave, these are given by the parameters on the function call.

These macro transactions are then decomposed accordingly to the bus specification (burst are bound to 1 Ko address space) and the arbitration policy selected for the bus.

The bus notifies the slave access that some bytes must be written to its buffer. The slave access will provide a delayed answer to the bus write transaction query. This is the slave access that count the number of cycles used to complete the bus transaction.

The slave access can give several replies accordingly to the amba bus specifications:

- Split: to stop the current transaction. The bus will memorize the state of the bus write transaction.
- Retry: the slave access notifies the bus that it can resume the transaction. Thus a new write query is done which deduces the already transmitted bytes.
- Error: notifies the bus that the transaction can not be accepted. It will have to retry it.
- Ok notifies the bus that the transaction is completed. When the last the bus write transaction of the macro transaction has completed, the bus returns from write\_txs, allowing the master access to call the slave IP operation.

The same schema applies to the macro read transaction called once the operation of the slave IP has completed. One may derive the slave access class to use a specific statistical model for the consumption of the data by the slave.

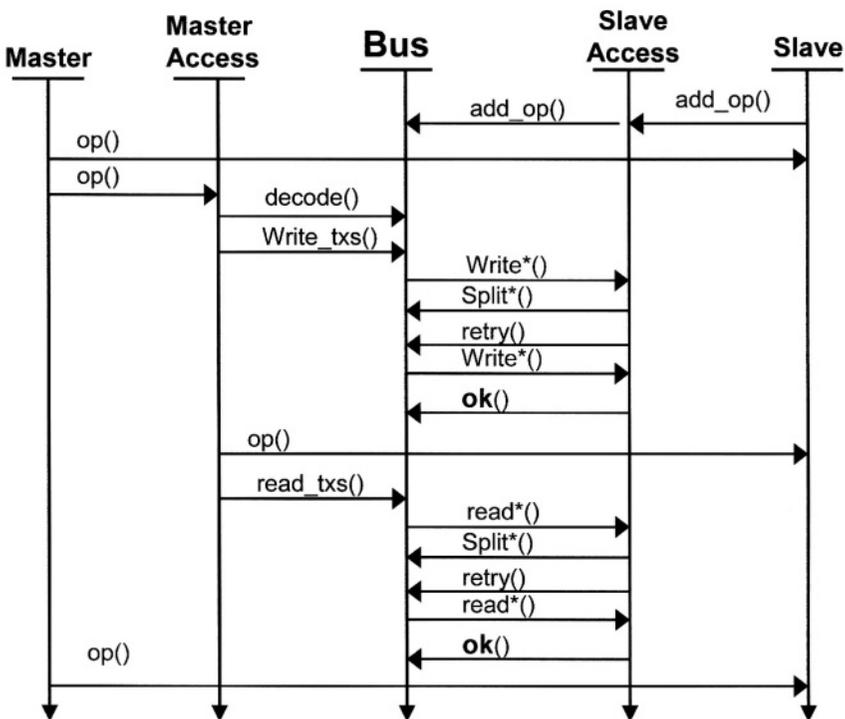


Figure 8-6. AMBA-bus Channel protocol.

## 2.2. Timing annotation

Providing timing annotations is a complex task. The camera module example shows the limited underlying systemc 2.0 mechanisms. Mainly we rely on an event which is never notified but this event has a timeout to resume the execution. This is satisfying for the hardware models, not for the software models. These models shall distinguish active wait from passive wait. The approach is very similar to SystemC 3.0 direction [8], with the wait and consume functions, though some other aspects are also taken into consideration which leads to have several consume-like functions.

The goal is to avoid annotating and recompiling anew the system each time a new cpu is selected, or each time a cpu frequency changes. Hence the annotation doesn't contain a timing, but a timing table index. The table indexes are the cpu index and the code chunk index. The cpu index is maintained by the simulation engine.

```
void consume(int cycle_idx, int ad_idx = 0);  
vs_timing_table_t consume(vs_timing_table_t, int cycle_idx,  
vs_address_table_t = 0, int ad_idx);
```

Because of the function inlining, preserving the modularity of the compilation requires to have a separate consume function that allows to change the current timing table and thus using the inline function tables. The timing in the table doesn't include absolute time, but a number of CPU cycles so that the elapsed time can be calculated accordingly to the CPU frequency. The last parameter is for a future version: the idea is to use a code offset to take into account the effect of an instruction cache. The consume functions shall be regarded as transaction calls to the operating system which guards the execution of the code. Consume suspends the execution of the code until the cycles required to execute this code have been allocated to that particular segment. Then one have to decide of the good tradeoffs for the simulation speed and the granularity of the annotations.

## 3. MPEG VIDEO DECODER CASE STUDY

The MPEG video decoder, illustrated in Figure 8-7, is a application built to show the way of modeling a system at transaction level with the access mechanism. Its purpose was to get a hint on how fast a simulation of this level could, and which details are necessary to build an analysis. The first step was to design the platform architecture. The edit mode of the tool allows to draw the hierarchical blocks of the design with their operations. We can either define new modules or reuse existing ones that are already included into a component library. Some components are provided with the tool such as the AMBA AHB bus.

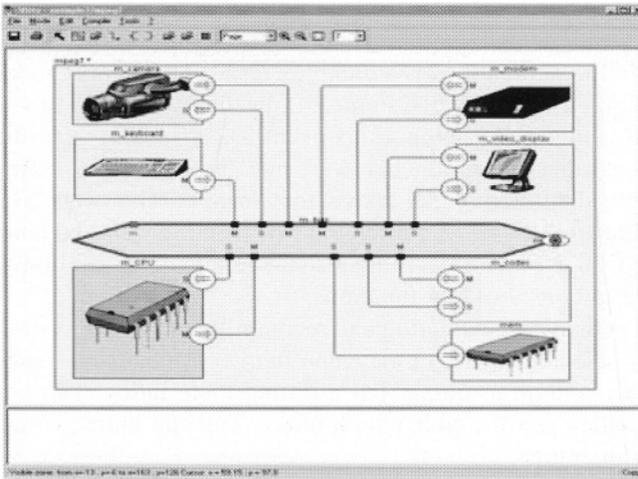


Figure 8-7. MPEG system capture.

We made a simplified system with 8 modules having one to two `sc_threads`:

- A control unit (SW) organizing all the data flows between the modules. It contains 2 threads, one to control the reception and the other to control the emission. The CPU has not been modeled.
- A memory, present uniquely to model the transactions to it.
- A camera (HW), getting the speaker video sequence, and writing video frames to memory. An extract of the code is shown in Figure 8-8 and

```
// master access definition
struct camera_master:vs_amba_ahb_master {
void write(const vs_address_t ad, const int num_bytes)
{ vs_amba_ahb_transaction tx(
// slave access definition
typedef vs_amba_ahb_slave camera_slave;
typedef camera_slave::communication_if::
op<void, TYPELIST_1(const vs_address_t)>
vs_amba_ahb_transaction::HSZ_32_BITS,
vs_amba_ahb_transaction::BRST_INCR*256,
vs_amba_ahb_transaction::TR_SEQUENTIAL);
(*m_write)(vs_bus_base::MASTER_EXCLUSIVE,
1,&tx, ad, num_bytes); }
memory_write_op* m_write;
void end_of_elaboration()
{decode(m_write, mem::BEGIN, mem::WRITE);}
camera_master(const char* name)
: vs_amba_ahb_master(name) {}
};
```

Figure 8-8. Access code.

Figure 8-9. The module define an operation `start_capture` following the access pattern. It also contains a `sc_thread` that represents the filming action. A `sc_event` is used to trigger the `sc_thread` within `start_capture`. Once the camera is filming the `sc_thread` has a timed execution with a `wait`. This `wait` is not triggered by the `start_capture` but by a `timeout`. SystemC 2.0 currently enforces the usage of an event here though.

- A graphic card (HW), reading and composing the in and out images to print them.
- A codec which emulates the memory transactions to encode and decode the images. One would have prefer reusing an IP which hasn't been delivered in time, so we have made it a transaction generator.

```

// serving module definition
SC_MODULE(camera) {
  sc_port<vs_slave_access_if> m_slave_access;
  sc_port<camera_master> m_master_access;

  void start_capture(const vs_address_t) {
    { m_dest = dest;
      m_capturing = true;
      m_vid_seq.reset();
      m_do_capture.notify(); }
    camera_start_capture_op m_start_capture;

    enum {
      START_CAPTURE = 0x000000F0,
      ...
    };

    void end_of_elaboration()
    { m_slave_access->add_op(&m_start_capture,
                          START_CAPTURE);
      ... }

    void capture_image()
    { while (1) {
      if (m_capturing)
        vs_wait(17, SC_MS); // 60 img/second
      else wait(m_do_capture);
      m_vid_seq.next_frame();
    }
  }

  sc_event m_do_capture;
  bool m_capturing;
  qcif_reader m_vid_seq;
  vs_address_t m_dest;

  SC_CTOR(camera) :
  m_start_capture(this,
                  &camera::start_capture),
  m_vid_seq("back"), m_capturing(false), ...
  { SC_THREAD(capture_image);...;end_module(); }
};

```

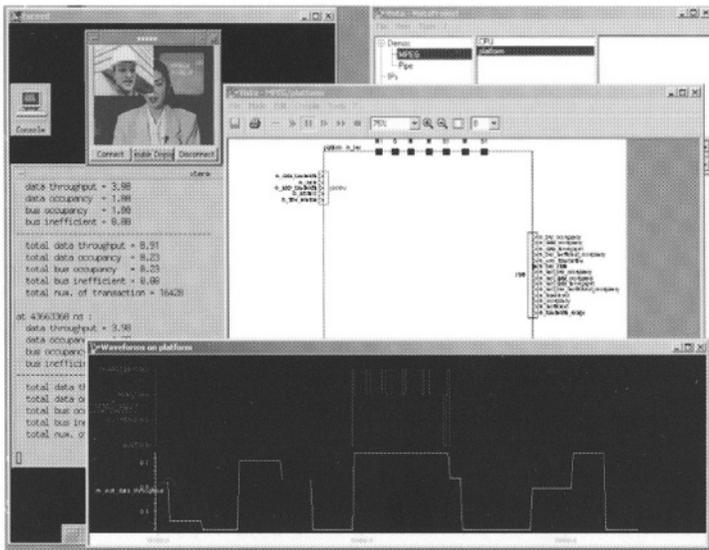
Figure 8-9. IP Module Code.

- A modem (HW) which emulates the transaction to emit the encoded picture and receive a picture accordingly to a baud rate.
- A video card which combines both pictures and synchronize with a GUI posix thread tracing the images.
- A keyboard to activate the flows and switch between simple and double display.

The tool generates code for the accesses to the bus and a skeleton of module to be completed with the operation implementation. Then everything is compiled and linked to obtain a simulator of the platform. The simulator can be launched as a self standing simulator like a classical SystemC2.0 program, or one may use the VISTA tool to control and monitor the execution of the simulation (step by step, continuous, pause, resume).

The tool also allows to drag observable variables to an oscilloscope which displays their states. These variables are figured like kind of output ports in the tool. The tool can interact with the simulation to change some settings of the system for analysis purpose. Some configuration variables can be introduced in a system model. They are figured like kind of inputs ports. These observable and configuration variables are provided with the vista library: there are no signals at the transactional level of abstraction.

VISTA embeds a protocol which allows to run the monitoring and the simulator tools on two different hosts. It even allows to monitor SystemC programs without having modeled them with the tool. Using these vista variables allows to drive an analysis session illustrated in Figure 8-10. For example, one may think of increasing or decreasing the baud rate of the



Colour picture

Figure 8-10. VISTA analysis session.

modem or its buffer size. One may also maintain a an observable Boolean variable, true as long as a percentile of missed frame occurs.

#### 4. CONCLUSION

We have presented a new methodology and tool for modeling SoC virtual platform for SW development and system level performance analysis and exploration. Using our VISTA methodology approach, we have been able to simulate the performances of a limited Visio phone system by generating 10,000 bus transactions per second, and running 0.5 second real time simulation in 20s while having the traces activated.

This environment is not primary intended for debugging a system. Some defects may be rather difficult to be shown with the tool, and the assertions may hardly be used for other purposes than performance and parallelism analysis due to the computational model.

But we think that such a level of modelling can allow to extract a formal specification of the system, if the communication channels are previously formalized [8, 9].

#### REFERENCES

1. Kanishka Lahiri, Anand Raghunathan, Sujit Dey , “Fast Performance Analysis of Bus-Based System-On-Chip Communication Architectures”. Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD), November 1999.
2. Frederic Doucet, Rajesh K. Gupta, “Microelectronic System-on-Chip Modeling using Objects and their Relationships”; in IEEE D&T of Computer 2000.
3. A. Clouard, G. Mastrorocco, F. Carbognani, A. Perrin, F. Ghenassia. “Towards Bridging the Precision Gap between SoC Transactional and Cycle Accurate Levels”, DATE 2002.
4. A. Ferrari and A. Sangiovanni-Vincentelli, System Design. “Traditional Concepts and New Paradigms”. Proceedings of the 1999 Int. Conf. On Comp. Des, Oct 1999, Austin.
5. Jon Connell and Bruce Johnson, “Early Hardware/Software Integration Using SystemC2.0”; in Class 552, ESC San Francisco 2002.
6. “Functional Specification for SystemC 2.0”, Version 2.0-P, Oct 2001
7. Thorsten Grotker. “Modeling Software with SystemC3.0”. 6th European SystemC User Group Meeting, Italy October 2002.
8. Fabrice Baray. “Contribution à l’intÈgration de la vÈrification de modÈle dans le processus de conception Codesign”. Thesis, University of Clermont Ferrand 2001.
9. S. Dellacherie, S. Devulder, J-L. Lamber. “Software verification based on linear programming”. Formal Verification Conference 1999.

*This page intentionally left blank*

## Chapter 9

# A FLEXIBLE OBJECT-ORIENTED SOFTWARE ARCHITECTURE FOR SMART WIRELESS COMMUNICATION DEVICES

Marco Götze

*Institut für Mikroelektronik- und Mechatronik-Systeme (IMMS) gGmbH, Ilmenau, Germany*

**Abstract.** This chapter describes the design considerations of and conclusions drawn from a project dealing with the design of a software architecture for a family of so-called *smart wireless communication devices (SWCDs)*. More specifically, based on an existing hardware platform, the software architecture was modeled using UML in conjunction with suitable framework and product line approaches to achieve a high degree of flexibility with respect to variability at both the hardware and application software end of the spectrum. To this effect, the overall design was split into a middleware framework encapsulating specifics of the underlying hardware platform and OS, and product line modeling of applications on top of it.

**Key words:** communications, frameworks, product lines, UML, SWCDs

## 1. INTRODUCTION

As is the case in about all areas of IT, the communications industry is in the firm grip of a trend towards growing complexity of products while components are being miniaturized. This is reflected by complex products of often astonishingly small dimensions, such as cell phones or PDAs.

Yet besides these consumer-oriented gadgets, there is a class of devices of similar functionality and complexity: the class of so-called *smart wireless communication devices*, short *SWCDs*.

As the name implies, a common, defining aspect of these devices consists in their ability to communicate with their environment by wireless means. Furthermore, they expose a certain “intelligence,” i.e., they are characterized by a certain complexity on the one hand and autonomy with respect to their functionality on the other hand.

Areas of application of SWCDs are many and diverse, ranging from pure telematics to security applications to remote diagnosis and monitoring. A typical application example consists in fleet management, as will be dealt with in more detail later on.

Despite the vastly different fields of application, all SWCDs share a number of specific characteristics and resulting design challenges: their ability to communicate with their environment, their autonomous functionality, their

inherent internal concurrency, a low cost threshold, a high degree of expected robustness, and continuous innovation.

Coping with all of these demands represents an enormous challenge for manufacturers of SWCDs. Consequently, a trend promising a better handling of most of the above requirements has established itself: a continuing shift of complex functionality from hardware to software [1].

However, continuous technological progress often requires switching from one hardware platform to another, possibly even during the life time of a single product. Induced by changing platforms, the operating system or even the programming language underlying the development of a specific application may change.

A flexible software architecture should anticipate such changes in its design and provide means of abstraction from underlying specifics while allowing for even complex applications to be realized efficiently.

A facilitating factor in conjunction with this is that despite the generally limited resources, technological progress causes the resources available to applications running on SWCD platforms to increase continually. In fact, the use of full-featured real-time operating systems in SWCDs has only recently become viable, which is one of the reasons why the state of the art of software development for this class of devices is still commonly sub-standard compared to, e.g., business software development.

These considerations formed the basis of the project summarized in this chapter, the subject of which consisted in developing a software architecture for an existing hardware platform, a first member of a hardware product family, using object-oriented design methods. More specifically, the software architecture and application to be developed were meant to replace an existing C library and application. In order to account for variabilities of the specific SWCD platform family, adequate approaches to modeling frameworks and software product lines were evaluated and applied.

By applying state-of-the-art object-oriented design techniques – specifically, UML-based modeling – we hoped to achieve benefits regarding the quality of the overall design and documentation, leading to improved maintainability, flexibility, and productivity. Moreover, by employing UML modeling tools supporting code generation, we expected a further increase in productivity and adaptability [2].

In contrast to mere case studies, the project aimed to apply design techniques feasible with tools that were available at the time to accomplish a functional design, thus taking different approaches' practicability at that time into consideration.

The sections following this introduction will introduce the SWCD platform in more detail and then discuss the choice and realization of a split overall design of a middleware framework and application product line development in the course of the project.<sup>1</sup> In a final section, conclusions will be drawn.

## 2. PLATFORM AND APPLICATION SPECIFICS

The development of the software architecture and application were to be based on an initial member of a product family of SWCDs, requiring support of that particular platform while being open to future extensions.

The resource-constrained, cost-optimized SWCD hardware platform (for details, see [3]) was resembled by a core module comprising a NEC-type processor operating at 20 MHz, a variable amount of on-board RAM (up to 2 MB), and a number of optional components, such as a GPS module, a GSM/GPRS modem, UARTs, a real-time clock (RTC), Flash memory, etc. This core module was meant to be supplemented by a power supply, additional interface circuitry, and any of a number of additional components, such as USB, Ethernet, or Bluetooth interfaces.

As can be concluded from this brief description, the hardware platform was characterized by a high degree of variability with respect to incorporated components. This variability is shown in more detail in the feature diagram of Figure 9-1. Furthermore, rather than assuming just a single specific component per component type, components could vary as well with respect to their exact model or even manufacturer.

The basis for software development on this SWCD platform was formed by eCos™, a real-time operating system (RTOS) originally developed by Red Hat, Inc. eCos™ is open-source and license-cost-free, contributing to a reduction in overall costs. It offers both a native and a number of standardized (though limitedly-implemented) APIs.

Building upon both the hardware's capabilities and the RTOS, there is a prototypical application for this kind of platform which is able to make use of the entire range of (optional and mandatory) components the hardware

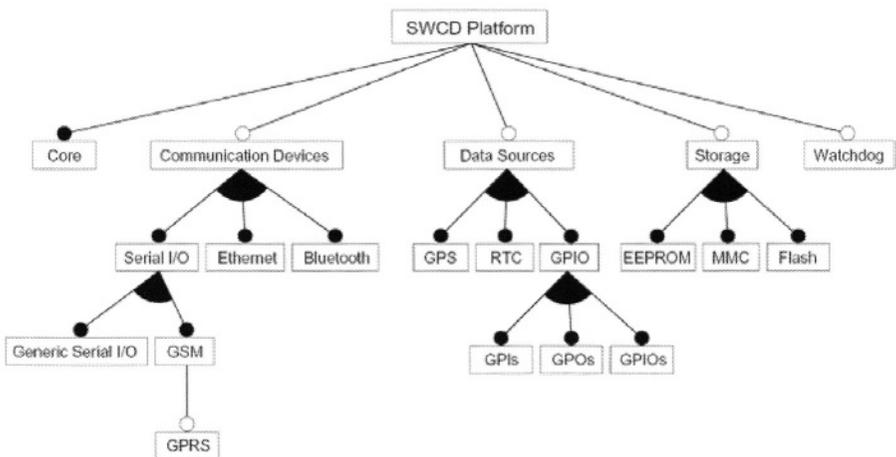


Figure 9-1. Feature diagram of the SWCD platform.

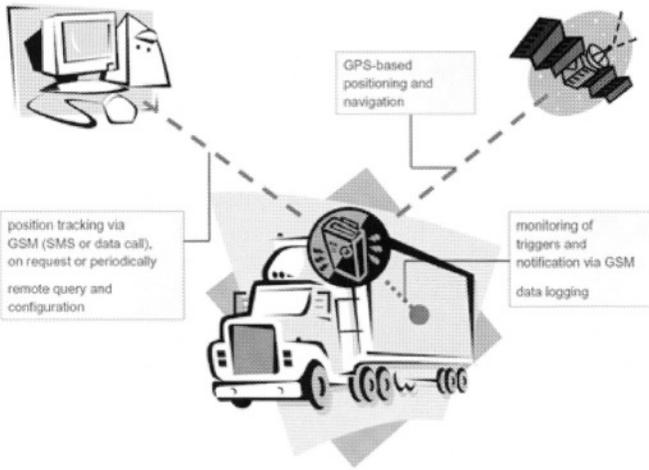


Figure 9-2. Application scenario of tracking devices.

platform consists of: the so-called tracker application (for an illustration, see Figure 9-2; a use case diagram is given in Figure 9-5 on page 10).

The SWCD hardware platform in a suitable configuration, combined with the tracking application, is meant to resemble a tracking device to be deployed in fleets of trucks or other vehicles. There, they gather and relay (via, e.g., GSM) positional (GPS-determined) and auxiliary (from general-purpose I/Os, GPIOs) data to a remote administrative center, allowing it to optimize operations.

Considered in conjunction with the fact that the underlying hardware platform was engineered to be variable with respect to its assortment of components, this implied the same degree of variability for the tracker application. Other than offering a full-fledged tracking functionality for the purposes of fleet management, the tracker application had to be easily extensible or otherwise customizable to fit varying areas of application. If distributed as open-source software, customers would be able to adapt the device to their specific requirements, relieving them of the effort of developing a completely custom application from scratch.

### 3. THE OVERALL SOFTWARE ARCHITECTURE

To maximize flexibility, an overall software architecture consisting of a middleware framework and application product line development was found to be the most promising approach.

Examining this layered architecture, depicted in Figure 9-3, from bottom to top, different hardware platforms as a whole form the basis for a product family of SWCDs.

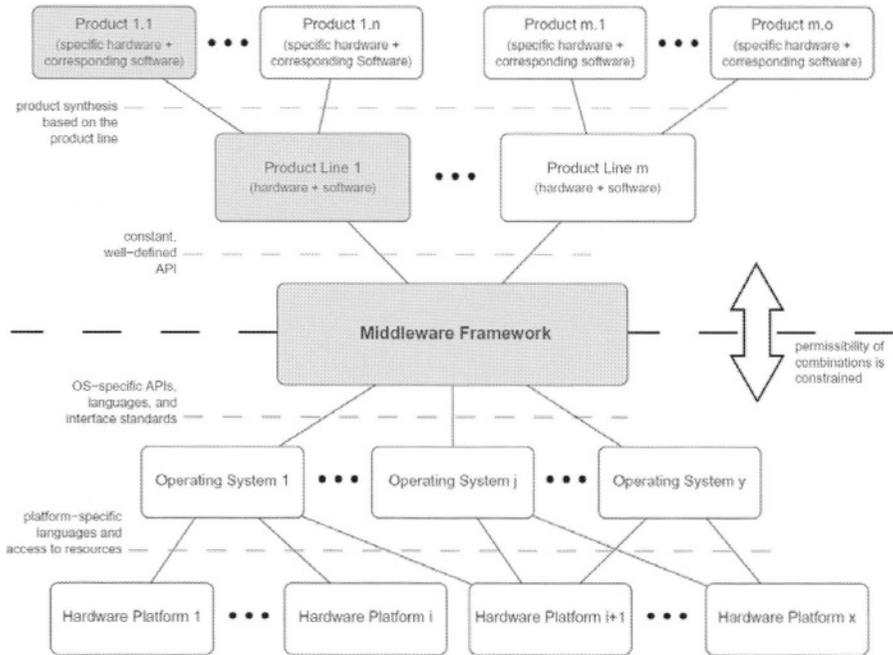


Figure 9-3. Layer model of the overall SWCD architecture.

Building upon these hardware platforms, a set of operating systems resembles a first abstraction layer. This abstraction is meant to provide flexibility with respect to the OS: while currently, eCos™ is used, future platforms of the family may promote a switch to a different operating system (for reasons of support, efficiency, etc.). By considering this possibility right from the start on, the flexibility of the entire design is increased.

The central element is a middleware platform, implemented as a framework architecture. Correspondingly, it resembles the glue between (variable) operating systems on different hardware platforms on the one hand and a number of product lines on the other hand. The purpose of this middleware is to abstract from the underlying hardware and operating systems and offer a well-defined (UML-modeled) API for application development.

Consequently, product line modeling builds upon the middleware. Assuming an efficient, well-defined API, product lines can be designed without the need of individual adaptations to specifics of the hardware and OS, which reduces development time should further product lines have to be added. On the other hand, modeling product lines instead of concrete applications reduces the development time required if further product variants are defined.

As the UML modeling tool of choice, we selected ARTiSAN Real-Time Studio™, primarily because of its UML 1.4 support and its customizable, template-based code generator.

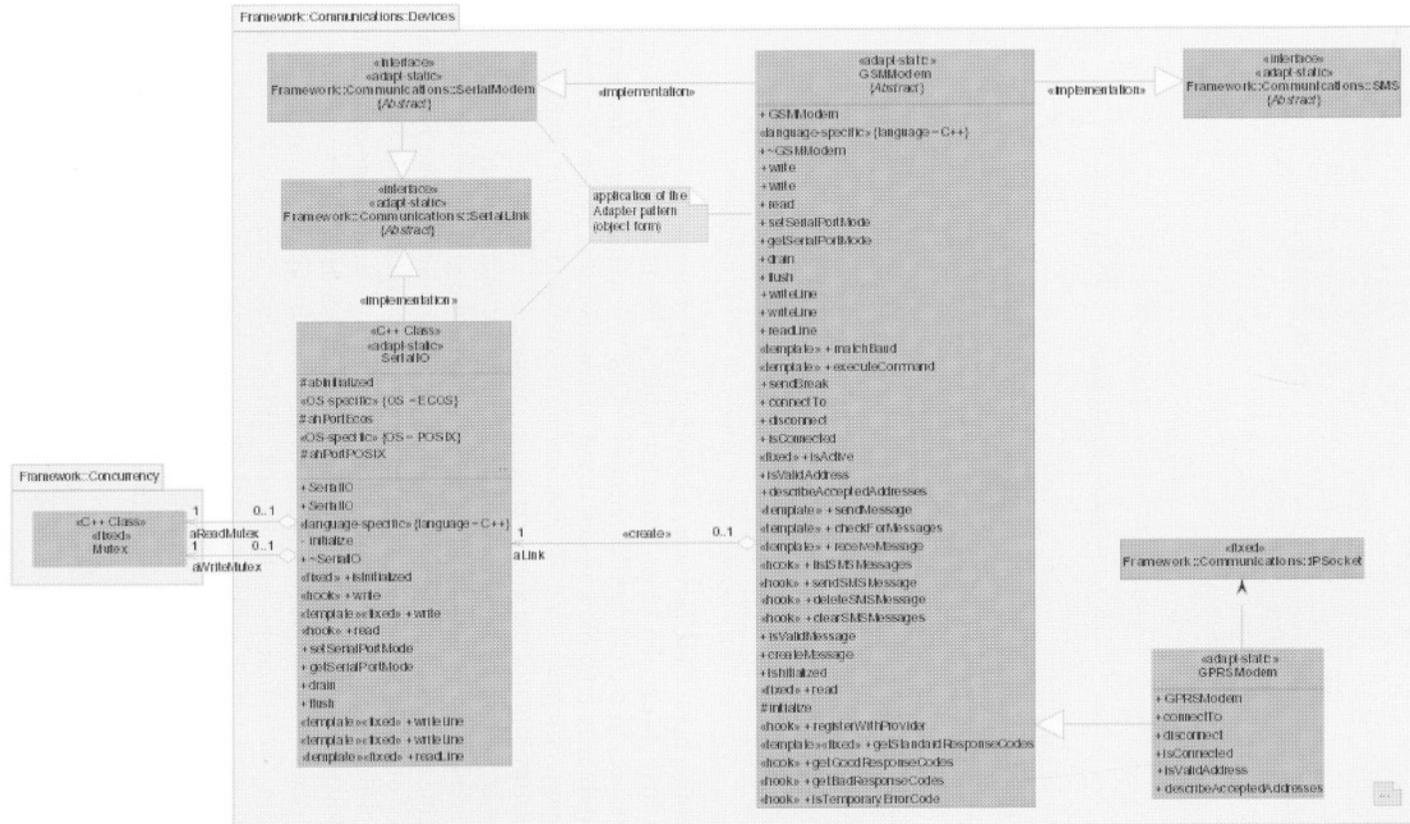


Figure 9-4. Portion of the middleware's communications package in which generic implementations of various interfaces are defined.

The following two sub-sections will discuss the design of the middleware framework and the product line architecture in detail. These elements of the overall architecture, the actual subject of the project, are designated by gray boxes in the diagram of Figure 9-3.

#### **4. THE MIDDLEWARE FRAMEWORK**

As stated in the previous section, the primary design goal of the middleware framework was an abstraction from specifics of the underlying OS, hardware, and – if possible – implementation language. To achieve this, a standardized, object-oriented API needed to be designed, encapsulating OS and hardware specifics. This API had to be sufficiently generic to support different product lines and yet sufficiently specific to significantly facilitate application development.

The most reasonable-seeming approach to the design of the middleware framework consisted in mapping elements of the SWCD platform's feature model (see Figure 9-1) to classes (a small portion of the model is shown in Figure 9-4) and packages. Generally, atomic features were turned into classes, composite features into packages. Additional packages and classes were created for elements not covered by the hardware feature model but for which abstractions had to be provided, such as means for handling concurrency. Classes were interrelated using refinement/abstraction, implementation, and generic dependency relations. Relations were defined in such a fashion that hardware/OS dependencies were kept within as few classes as possible, reducing the effort posed by adding support for a new platform later on. Common base and dedicated interface classes were used to capture commonalities among components of a kind, such as stream- or message-based communications components/classes.

We furthermore tried to follow a number of framework (and software architecture in general) design guidelines [5, 6], reduced dependencies among packages to improve the ability to develop packages separately from each other, and applied various design patterns [7, 8].

As for abstraction from hardware and OS specifics, besides differences in the way, e.g., components are accessed, the OS-level support for specific components might vary or even be missing. Furthermore, there were components (such as GSM modems or GPS modules) which are only rudimentarily supported at OS level (via generic drivers) – for these, the middleware framework should provide a uniform, higher-level API, also abstracting from specific models of a given type of component.

To this end, the framework design had to accommodate three dimensions of variability:

- varying operating systems;

- varying hardware platforms; and
- varying implementation languages.

The approach chosen to reflect and provide support for these variabilities at the UML level consisted in defining stereotypes and associated tagged values – the UML’s primary means of extension. Using «OS-specific», «platform-specific», and «language-specific» stereotypes, model elements were marked as specific with respect to one (or more) of these dimensions (for an example of their usage, see class `SerialIO` in the class diagram of Figure 9-4).

Even though adding a different implementation language to the framework obviously represents an immense effort, and C++ has been the only *intended* language, the framework was designed so as to reduce dependencies on a specific language’s capabilities. More precisely, the set of OO features used was limited to that of the Java programming language wherever viable because the latter’s set of supported OO notions was considered a common denominator among current OO languages. Inevitable C++ specifics were accounted for by using dedicated Real-Time-Studio™-provided stereotypes, avoiding pollution of the actual model with language specifics.

In order to provide for the extensibility demanded from a framework, some of the mechanisms suggested in the UML-F profile [9] were applied. These consisted in the use of specific stereotypes to designate hot spots of (explicitly) expected extensions. These stereotypes, in contrast to the ones introduced above, however, are of a mere documentary nature.

Since the framework resembles, in first regard, a mapping from a constant high-level API to a set of native APIs of different operating systems and hardware platforms, only few interactions were sufficiently complex to warrant dynamic modeling. Methods were implemented manually within the UML tool, and sequence diagrams were used to illustrate the behavior of more complex implementations. The manually-implemented dynamic code was complemented by Real-Time Studio™’s code generator, effectively realizing the vision of “single-click” code generation.

A mapping from modeled to code-level variabilities with respect to OS and platform was achieved by adapting Real-Time Studio™’s code generation templates to surround the respective model elements’ source code artifacts by pre-processor statements, reducing the problem to conditional implementation. Likewise, dependent fragments of methods’ bodies implemented within the UML tool accounted for such variabilities through the explicit use of conditional pre-processor statements. The resulting code can be configured and compiled for a given hardware platform by defining appropriate pre-processor constants.

This static management of variabilities (as opposed to run-time dependencies) is sufficient since OS, platform, and language dependencies can all be resolved at build time. This still allows for later variability with respect to the features used in an application since the latter will only include those

portions of the framework that are actually required and included in a given hardware configuration.

### 5. THE TRACKER PRODUCT LINE

The functional scope of the application as modeled and implemented in the course of the project was limited to basic requirements, aiming to prove the flexibility of both the overall architectural concept and the middleware framework rather than create a full-fledged alternative model and implementation at that point in time. The modeled and implemented functionality is described by the use case diagram of Figure 9-5.

As has been detailed before, the tracker application product line was meant to (finally) be both a full-fledged tracking application and a basis for customizations by customers. As such, modeling it in UML offered immediate benefits regarding documentation and thus maintainability on the manufacturer's as well as adaptability on the customer's side.

A product line approach was chosen because the actual platform on which

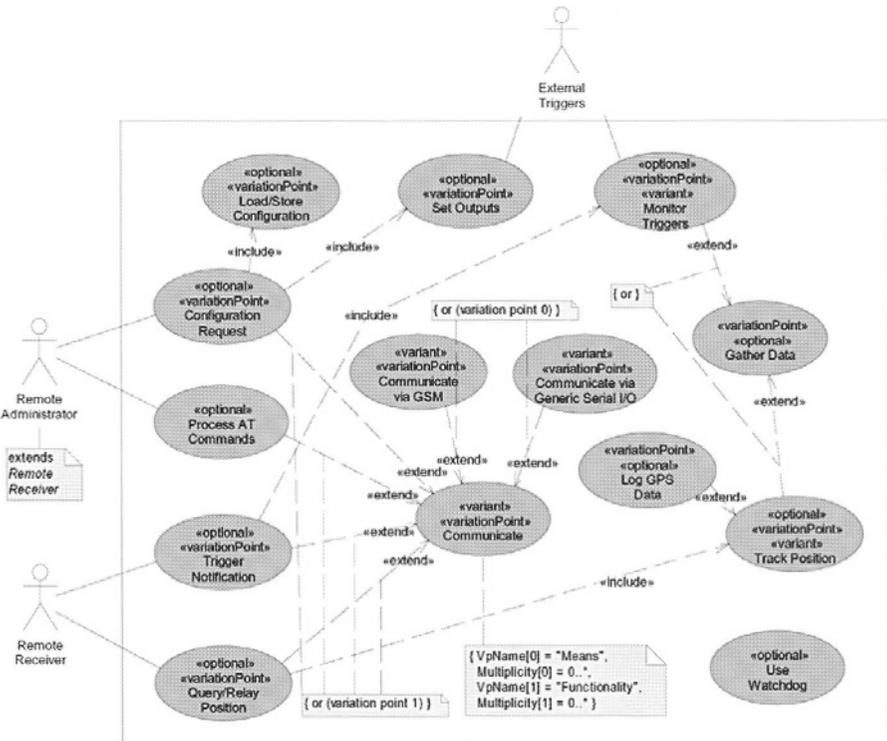


Figure 9-5. Use case diagram of the tracker application.

the tracker application was to be deployed should be allowed to vary significantly, both with respect to core components (processor, OS) and dependency on specific auxiliary components. This means that the product line had to accommodate products ranging from, e.g., multiply-connected SWCDs (via GSM/GPRS and further means) to the extreme of isolated devices that only perform data logging.

Since the handling of hardware specifics is delegated to the middleware, product line modeling can focus on merely using the components supported by the middleware framework and accounting for configurability rather than also having to cope with hardware and OS specifics.

While UML lacks the means to natively deal with variability and configurability, an intuitive, stereotype-based notation was suggested by Clauß [10], which became the approach chosen to represent the product line's variability in UML (for an example, see the use case diagram of Figure 9-5).

A number of advanced techniques for achieving implementation-level configurability were considered and evaluated, including composition filters, aspect-oriented programming, subject-oriented programming, and generative programming. However, all of these approaches require either a particular implementation language or dedicated tools to be put into practice. The resulting lack in flexibility and the run-time overhead of some of these concepts made us take a simpler yet language- and tool-independent approach, exploiting specifics of the variabilities encountered in the tracker application's specification.

Compared to software product lines sold as stand-alone products, the tracker application's variability was special in that available hardware features determined the software features to be provided – further arbitrary restrictions were neither planned nor required. This fact effectively reduced the complexity of the variability requirements by binding software features to hardware features – which were, for the most part, modeled as classes in the middleware framework.

Based on these considerations, a number of techniques were selected and combined in our approach:

- specialization through inheritance (based on interfaces and base classes provided by the middleware) and the Factory pattern [6]
- instantiation and passing of objects conforming to certain interfaces and base classes – or lack thereof – were used to determine whether or not certain software features are to be present
- instantiation of thread classes associated with particular types of components with strict encapsulation of component-specific functionality, dependent on whether or not instances of the respective component classes were created and passed to the constructor of a central, coordinating thread class
- optimization by the compiler/linker, preventing unused classes' code from being included in the application's binary
- use of a configuration object [5] to handle parametric variabilities (such as timeouts, queue lengths, etc.)

While lacking in flexibility compared to more sophisticated techniques (whose application was ruled out for reasons discussed above), this approach was feasible since the configuration of tracker products would happen entirely at build time. Furthermore, it represented an efficient choice in the context of embedded applications as it does not incur any (noteworthy) run-time overhead.

The static model of the tracker product line comprises a number of concurrent threads, each encapsulated by a class, for handling GPS-based position tracking, the GSM modem, generic external serial ports, the watchdog, etc. An additional, central element is resembled by a dedicated thread responsible for instantiating and coordinating the other threads in correspondence with the hardware configuration. Figure 9-6 gives an overview of the basic application structure.

As for the dynamic model, behavior was modeled using activity, state, and sequence diagrams. Since the basic nature of the tracker application is data-processing rather than state-driven, state diagrams did not play a major role, ruling out the use of state-diagram-based code generation. Instead, implementation was done manually in C++. Since methods' implementations were entered within the UML tool, a run of the code generator nevertheless yielded complete application code.

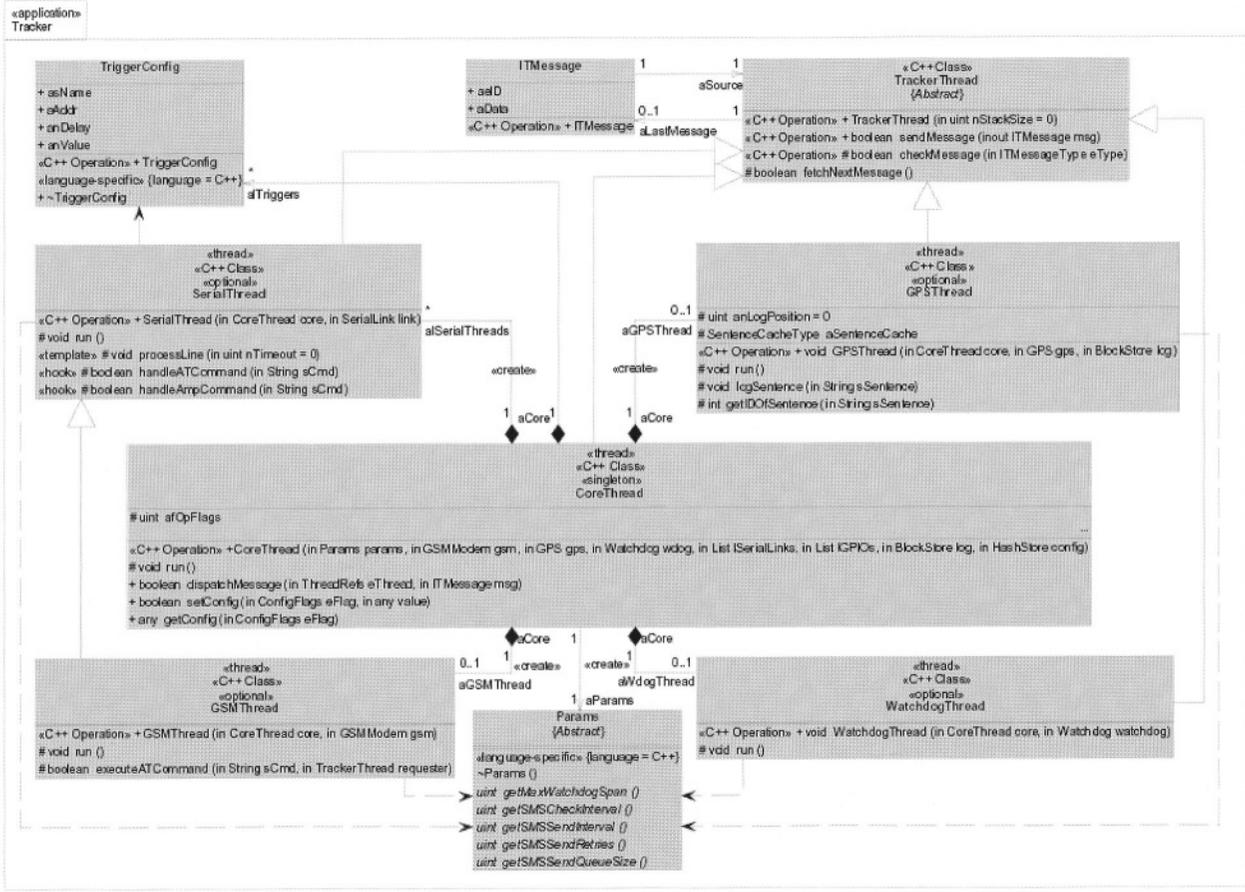
The configuration of a concrete application of the tracker product line is achieved by adapting a source code file taking care of the instantiations described above. By commenting out instantiations or substituting other sub-classes, an application can be configured in correspondence with a given hardware configuration. Alternatively, to make things more comfortable, a GUI-driven configuration tool generating the instantiation code could be implemented easily.

## **6. CONCLUSIONS AND OUTLOOK**

Thanks to the combination of framework and product line development, the flexibility of the software designed in the course of the project clearly surpasses that of the previous, conventional implementation.

The approaches chosen to model variabilities in UML have proven suitable to accommodate the requirements of the software architecture. Thanks to the template-based code generation facility provided by Real-Time Studio™, the stereotypes used could be mapped to customizations of code generation templates, thus linking the modeled variability to the source code level. Combined with methods' bodies implemented within the UML tool, "single-click" code generation has been shown to be feasible. This automation avoids the danger of introducing inconsistencies among model and code which are likely to result from the need to manually adapt characteristics at both levels while offering a high degree of flexibility with respect to variabilities at both the OS/hardware platform and application end of the spectrum.

Figure 9-6. Portion of the class diagram of the tracker application.



Moreover, the transition from structured (C) to object-oriented (C++) software development on the specific target platform has proven feasible in terms of both code size and run-time performance, although definite comparisons to the previous, C-based implementation will only be possible once the modeled tracker application's functional scope has reached that of the previous implementation.

Discussions have furthermore indicated significant enhancements in terms of documentation and comprehensibility achieved through modeling the software in UML, suggesting that secondary benefits, such as improved maintainability and adaptability of the tracker application by customers, have in fact been achieved as well.

Of course, the precise degree to which a UML-based middleware framework architecture and a product-line-based application development approach help further maintainability and extensibility will only become evident once the architecture is being employed in practice – and so will the overall quality and flexibility of the design in comparison to the previous, conventional approach.

## NOTE

<sup>1</sup> Due to the limited scope of this chapter, the included diagrams offer only a superficial glance at selected aspects of the model. A much more detailed discussion of the project is provided in German in the author's diploma thesis [4].

## REFERENCES

1. T. Jones and S. Salzman. "Opening Platforms to Hardware/Software Co-Development." *Communications System Design Magazine*, December 2001.
2. M. Götze and W. Kattaneck. "Experiences with the UML in the Design of Automotive ECUs." In *Design, Automation and Test in Europe (DATE) Conference 2001*, volume Designers' Forum, 2001.
3. W. Kattaneck, A. Schreiber, and M. Götze. "A Flexible and Cost-Effective Open System Platform for Smart Wireless Communication Devices." In *International Symposium on Consumer Electronics (ISCE) 2002*, 2002.
4. M. Götze. *Entwurf einer Softwarearchitektur für Smart Wireless Communication Devices und darauf basierend Realisierung einer prototypischen Applikation*. Diploma thesis, Ilmenau University of Technology, Thuringia, Germany, 2003.
5. S. Demeyer, T. D. Meijler, O. Nierstrasz, and P. Steyaert. "Design Guidelines for Tailorable Frameworks." *Communications of the ACM*, Vol. 40, No. 10, pp. 60–64, October 1997.
6. R. C. Martin. [various articles]. *C++ Report*, January-December 1996. <<http://www.objectmentor.com/resources/articleIndex>>.
7. E. Gamma, R. Helm, R. Johnson, and J. Vlissedes. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
8. W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1995.

9. M. Fontoura, W. Pree, and B. Rumpe. *The UML Profile for Framework Architectures*. Addison-Wesley, 2002.
10. M. Clauß. “Generic modeling using UML extensions for variability.” In *16th Annual ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume Workshop on Domain Specific Visual Languages, 2001.

## Chapter 10

# SCHEDULING AND TIMING ANALYSIS OF HW/SW ON-CHIP COMMUNICATION IN MP SOC DESIGN

Youngchul Choi, Ganghee Lee<sup>1</sup>, Kiyoun Choi<sup>1</sup>, Sungjoo Yoo<sup>2</sup> and Nacer-Eddine Zergainoh<sup>2</sup>

<sup>1</sup> *Seoul National University, Seoul, Korea;* <sup>2</sup> *TIMA Laboratory, Grenoble, France*

**Abstract.** On-chip communication design includes designing software parts (operating system, device drivers, interrupt service routines, etc.) as well as hardware parts (on-chip communication network, communication interfaces of processor/IP/memory, etc.). For an efficient exploration of its design space, we need fast scheduling and timing analysis. In this work, we explore two problems. One is to incorporate the dynamic behavior of software (interrupt processing and context switching) into on-chip communication scheduling. The other is to reduce on-chip data storage required for on-chip communication, by making different communications to share a physical communication buffer. To solve the problems, we present both integer linear programming formulation and heuristic algorithm.

**Key words:** MP SoC, on-chip communication, design space exploration, communication scheduling

## 1. INTRODUCTION

In multiprocessor system on chip (MP-SoC) design, on-chip communication design is one of crucial design steps. By on-chip communication design, we mean (1) mapping and scheduling of on-chip communications and (2) the design of both the hardware (HW) part of communication architecture (communication network, communication interfaces, etc.) and the software (SW) part (operating system, device drivers, interrupt service routines (ISRs), etc.). We call the two parts HW communication architecture and SW communication architecture, respectively.

In our work, we tackle two problems (one for SW and the other for HW) in on-chip communication design. First, we present a method of incorporating, into on-chip communication scheduling, the dynamic behavior of SW (interrupt processing and context switching) related to on-chip communication. Second, to reduce on-chip data storage (in our terms, physical communication buffer) required for on-chip communication, we tackle the problem of making different communications to share a physical communication buffer in on-chip communication scheduling.

*On-chip communication design considering both SW and HW communication architectures*

Most of previous on-chip communication design methods focus on HW communication architecture design, such as bus topology design, determining bus priorities, and DMA size optimization [3–7]. A few studies consider the SW communication architecture in on-chip communication design [5, 8]. In [5], Ortega and Borriello presented a method of SW architecture implementation. In [8], Knudsen and Madsen considered device driver runtime (which depends statically on the size of transferred data) to estimate on-chip communication runtime. However, the behavior of SW communication architecture is dynamic. The dynamism includes interrupt processing, context switching, etc. Since the overhead of interrupt and that of context switching can often dominate embedded SW runtime, they can significantly affect the total performance of HW/SW on-chip communication. In this work, *we take into account the dynamic behavior of SW communication architecture in scheduling on-chip communication (Problem 1).*

*Physical communication data storage: a physical communication buffer sharing problem*

In MP SoCs, multiple processors, other IPs and memory components communicate with each other requiring a data storage, i.e. physical communication buffer, to enable the communication. It is often the case that the physical communication buffer can take a significant portion of chip area. Especially, in the case of multimedia systems such as MPEG 2 and 4, the overhead of physical communication buffer can be significant due to the requirement of large-size data communication between components on the chip. Therefore, to reduce the chip area, we need to reduce the size of the physical communication buffers of on-chip communication architecture. In our work, for the reduction, *we take into account the sharing of the physical communication buffers by different communications in scheduling on-chip communication (Problem 2).*

## 2. MOTIVATION

Figure 10-1 shows a simple task graph and its mapping on a target architecture. The three tasks in the graph are mapped on a microprocessor and a DSP. The two edges (communication buffers) are mapped on a shared memory in the target architecture. Figure 10-1(b), (c), and (d) shows Gantt charts for three cases of scheduling task execution and communication.

In Figure 10-1(b), we assume that the two edges mapped on the shared memory do not share their physical buffers with each other. Thus, two physical buffers on the shared memory occupy separate regions of the shared memory. Figure 10-1(b) exemplifies the execution of the system in this case. First, task T1 executes and writes its output on its output physical buffer on the

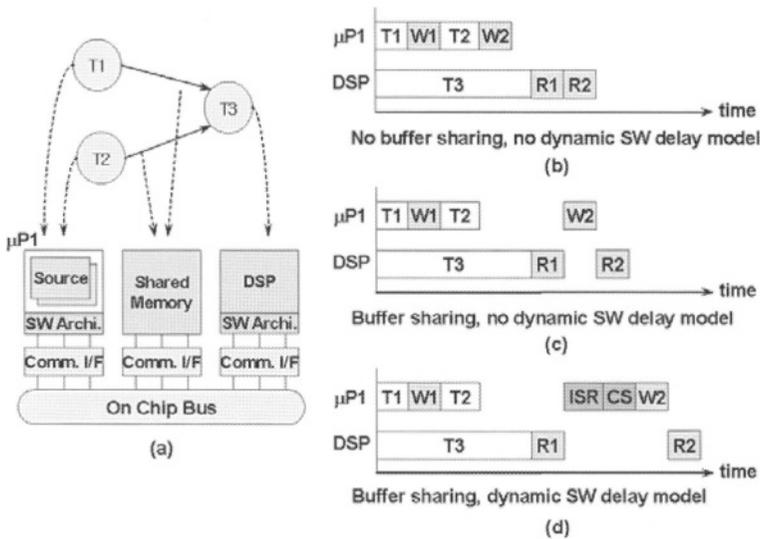


Figure 10-1. A motivational example.

shared memory by a write operation (W1). Task T2 runs and writes its output on its output physical buffer (W2). After task T3 completes its previous computation at time  $t_1$ , it reads from two input buffers (R1 and R2).

In Figure 10-1(c), we assume that the two edges of the task graph share the same physical buffer to reduce the physical buffer overhead. In this case, compared to the case of Figure 10-1(b), after the execution of task T2, the task cannot write its output data on the shared physical buffer, since the shared physical buffer is filled by task T1. Thus, the write operation of task T2 waits until task T3 reads from its input buffer. When the shared physical buffer becomes empty after the read of task T3, we assume that there is an interrupt to task T2 to invoke its write operation (W2). Note that, in this case, the dynamic behavior of SW architecture is not modeled. That is, the execution delay of interrupt service routine and that of context switching are not modeled in this case.

In Figure 10-1(d), the dynamic behavior of SW architecture is modeled. Compared to the case of Figure 10-1(c), when the interrupt occurs to invoke the write operation of task T2, first, it accounts for the execution delay of ISR. Then, it accounts for the delay of context switching (CS). As this example shows, compared to the case of Figure 10-1(c), when we model the dynamic behavior of SW, we can obtain more accurate system runtime in scheduling on-chip communication and task execution.

### 3. ON-CHIP COMMUNICATION SCHEDULING AND TIMING ANALYSIS

For an accurate timing estimation of on-chip communication for design space exploration, we actually perform optimal scheduling of the communication. For the accuracy, we consider the dynamic behavior of SW. We also consider the effect of buffer sharing on the communication scheduling.

#### 3.1. Problem definition and notations

##### *Problem*

Given a task graph, a target architecture (with its parameters, e.g. maximum physical buffer sizes on shared memory components), and a mapping of tasks and communications on the target architecture (including physical buffer sharing), the problem is to find a scheduling of task execution and communication which yields the minimum execution time of the task graph on the target architecture.

To solve the problem, we extend the task graph to an extended task graph that contains detailed communication operations (in our terms, communication nodes). Then we apply an ILP formulation or heuristic algorithm to schedule tasks and communications. For the on-chip communication, we assume on-chip buses and/or point-to-point interconnections.

##### *Extended task graph*

To solve the problem, we extend the input task graph (TG) to an extended task graph (ETG). Figure 10-2(a) shows a TG of H.263 encoder system and Figure 10-2(b) shows an ETG extended from the TG. As shown in Figure 10-2(b), we replace an edge of the TG into two communication nodes representing write and read data transactions. For instance, the edge between Source and MB\_Enc of Figure 10-2(a) is transformed to one communication node for the write operation,  $C_{Source}^{WI}$  and another communication node for read operation,  $C_{MB\_Enc}^{RI}$  in Figure 10-2(b).

#### 3.2. ILP formulation

Before explaining our ILP formulation, we explain our notations used in it.

##### 3.2.1. ILP for dynamic software behavior

The delay of the communication nodes in the ETG can be modeled in the ILP formulation as follows:

$C_i(n)$ : communication time of communication node  $v_i$ , where  $n$  is the size of communication data

$T_s(v_i)$ : start time of node  $v_i$

$T_f(v_i)$ : finish time of node  $v_i$

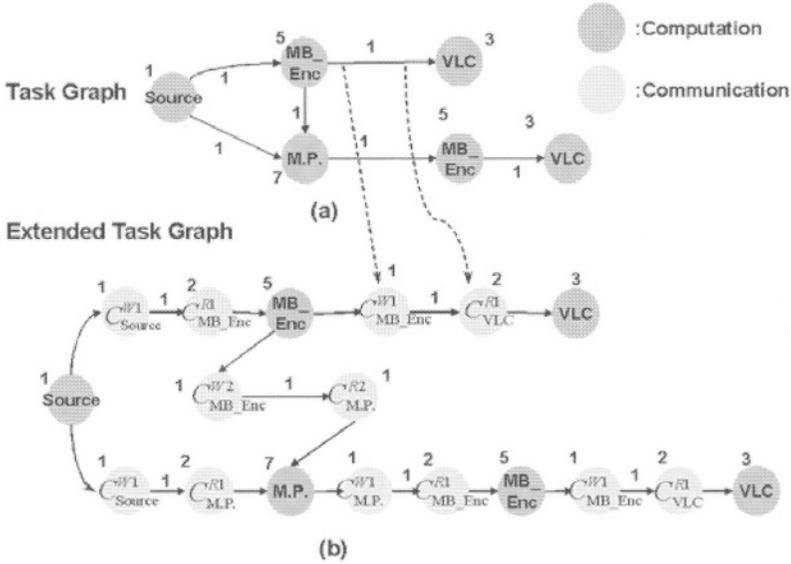


Figure 10-2. Extended task graph.

$R(v_i)$ : boolean variable representing the state of node  $v_i$

$$= \begin{cases} 0, & \text{if the task } v_i \text{ is waiting for the processor.} \\ 1, & \text{if the task } v_i \text{ is running on the processor.} \end{cases}$$

$WR(v_i)$ : constant boolean value representing the access type of communication node  $v_i$

$$= \begin{cases} 0, & \text{if the access type of } v_i \text{ is WRITE.} \\ 1, & \text{if the access type of } v_i \text{ is READ.} \end{cases}$$

Communication delay is caused by both SW and HW communication architectures. We decompose the delay into three parts as follows.

$$C_i(n) = C_i^{SW}(n) + C_i^{HW}(n) + C_i^{OCN}(n),$$

where  $n$  is the communication data size and  $C_i^{SW}(n)$ ,  $C_i^{HW}(n)$  and  $C_i^{PCN}(n)$  are the delays of SW communication architecture, HW communication interface, and on-chip communication network, respectively. The delay of software communication architecture is given by

$$C_i^{SW}(n) = R(v_i) \times [C_{ISR}(n) + C_{CS}(n)] + C_{DD}(n),$$

where  $C_{ISR}(n)$  is ISR delay,  $C_{CS}(n)$  is CS delay and  $C_{DD}(n)$  is device driver delay. If the software task is running on a processor, ISR delay and CS delay are zero. Else, the task is blocked and is in a wait state and the communication interface interrupts the processor to wake up the task. Thus, the ISR and CS delay should be counted into the communication time.

The delay of communication interface ( $C_i^{HW}$ ) is given as a function of data size [14].

The delay of on-chip communication network is

$$C_i^{OCN}(n) = D_{trans} \times n + D_{BLOCKED},$$

where  $D_{trans}$  is a constant delay of transferring a single data item and  $D_{BLOCKED}$  is a delay caused by the contention of access to the communication network (e.g. bus contention).

$R(v_i)$  and  $D_{BLOCKED}$  can be calculated during task/communication scheduling, which is explained in the next subsection.

### 3.2.2. ILP for scheduling communication nodes and tasks

In scheduling tasks and communication transactions, we need to respect data dependency between tasks and to resolve resource contention on the target architecture. In the target architecture, there are three types of resource that are shared by tasks and communications: processors, communication networks (e.g. on-chip buses), and physical buffers. Therefore, we model the scheduling problem with (1) data dependency constraints between tasks, (2) constraints that represent resource contention and (3) an objective function that aims to yield the minimum execution time of the task graph.

#### 3.2.2.1. Data dependency constraints

In the ETG, data and communication dependency is represented by directed edges. For every edge from node  $v_i$  to node  $v_j$ , the following inequalities are satisfied.

$$T_S(v_j) \geq T_F(v_i)$$

#### 3.2.2.2. Resource contention constraints

*Processor and on-chip communication network.* Processor contention and on-chip communication network contention are similar in that the contentions occur when two or more components are trying to access the same resource (processor or on-chip communication network) at the same time. To formulate the processor contention constraints in the ILP, first we find a set of task nodes that are mapped on the same processor. For the task set, the processor contention occurs when two tasks,  $v_i$  and  $v_j$  try to use the same processor simultaneously. To prevent such a processor contention, we need to satisfy  $T_S(v_j) \geq T_F(v_i)$  or  $T_S(v_i) \geq T_F(v_j)$ . The same concept applies to the on-chip communication networks such as on-chip buses and point-to-point interconnections.

*Physical buffer contention constraints.* Physical buffer contention constraints are different from those of processor and on-chip communication network

since simultaneous accesses to the physical buffer are possible. For instance, if the physical buffer size is 2, two communication transactions with data size of one can use the same physical buffer. The number of communication transactions is not limited unless the buffer is full. In terms of physical buffer implementation, we assume the number of memory ports of physical buffer as the maximum number of simultaneous accesses to the physical buffer. Our future work includes ILP formulation for the case where the number of memory ports of physical buffer is less than the maximum number of simultaneous accesses to the physical buffer.

For the ILP formulation of this constraint, we use a Boolean variable  $BU(v_i)$ .  $BU(v_i)$  is 1 if the physical buffer is used by a communication node  $v_i$  and 0 otherwise. In our physical buffer model, we assume for safety that the data is pushed to a buffer at the *start time* of write operation and popped at the *finish time* of read operation. Thus, if the access type of a node  $v_i$  is *read*, the value of  $BU(v_i, t)$  is determined as follows:

$$BU(v_i, t) = \begin{cases} 0, & \text{if } t < T_r(v_i). \\ 1, & \text{if } t \geq T_r(v_i). \end{cases}$$

or if the access type of a communication node  $v_i$  is *write*,

$$BU(v_i, t) = \begin{cases} 0, & \text{if } t < T_s(v_i). \\ 1, & \text{if } t \leq T_s(v_i). \end{cases}$$

However, since  $BU(v_i, t)$  is a non-linear function of another variable  $T_s(v_i)$  or  $T_r(v_i)$ , it cannot be a variable in the ILP formulation as it is. To resolve this, the condition for  $BU(v_i, t)$  is formulated as follows (for the case of write access type):

$$\begin{aligned} 1 \leq k_i &\leq \text{TimeMax} \\ \text{TimeMax} \times BU(v_i, t) - k_i &= t - T_s(v_i) \end{aligned}$$

where  $\text{TimeMax}$  is a big integer constant and  $k_i$  is a dummy variable. If time  $t < T_s(v_i)$ , the right hand side of the equation is negative. For the left hand side of the equation to be negative,  $BU(v_i)$  is 0 because  $k_i$  is equal to or smaller than  $\text{TimeMax}$ . If  $t \geq T_s(v_i)$ , the right hand side is zero or positive.  $BU(v_i)$  should be 1 because  $k_i$  is bigger than 1.

The physical buffer contention constraints are modeled as follows.

for all physical buffer  $B_k$   
for all communication nodes  $v_j$  using  $B_k$

$$\sum_{i=1}^n \{s(v_i) \times [WR(v_i) - 1] \times BU[v_i, T_r(v_i)] + s(v_i) \times WR(v_i) \times BU[v_i, T_s(v_i)]\} \leq B_{MAX}(B_k),$$

where  $v_i$  is a communication node using  $B_k$ ,  $n$  is the number of communication nodes that use physical buffer  $B_k$ ,  $B_{MAX}(B_k)$  is the maximum size of physical buffer  $B_k$ , and  $s(v_i)$  is the data size of communication node  $v_i$ .

We need to check every time step to see if the physical buffer contention constraints are satisfied. But the physical buffer status changes only at the start or finish time of communication. Thus, we can reduce the number of inequalities by checking buffer status only at the start or finish time of communication.

### 3.3. Heuristic algorithm

Considering that ILP belongs to the class of NP-complete problems, we devise a heuristic algorithm based on list scheduling, which is one of the most popular scheduling algorithms. The scheduling is a function of  $ETG = G(V, E)$  and  $\mathbf{a}$ , where  $\mathbf{a}$  is a set of resource constraints. Just like the ILP formulation, it considers physical buffers as one of the shared resources. The algorithm consists of four steps. 1) For each shared resource, a set of the candidate nodes (that can run on the resource) and a set of unfinished nodes (that are using the resource) are determined. 2) For each shared resource, if there is available resource capacity (e.g. idle processor or when the shared physical buffer is not full), the algorithm selects a node with the highest priority among the candidate nodes, where the priority of a node is determined by the longest path from the candidate node to the sink node. 3) It schedules the nodes selected in step 2. 4) It repeats steps 1 to 3 until all the nodes are scheduled. Figure 10-3 shows the scheduling algorithm.

```

LIST (G(V, E),  $\mathbf{a}$ ) {
  I = 1;
  Repeat {
    For each resource type  $k = 1, 2, \dots, n_{res}$  {
      Determine candidate tasks  $U_{I,k}$ ;
      Determine unfinished tasks  $T_{I,k}$ ;
      Select  $S_k \subseteq U_{I,k}$  nodes, such that  $|S_k| + |T_{I,k}| \leq a_k$ ;
      Schedule the  $S_k$  tasks at time I
    }
    by setting  $t_i = I, (\forall i: v_i \in S_k)$ 
  }
  } until ( $v_n$  is scheduled);
  return ( $\mathbf{t}$ );
}

```

Figure 10-3. List scheduling.

## 4. EXPERIMENTAL RESULTS

This section consists of two parts. In the first part we show the effectiveness of the proposed approach in terms of runtime and accuracy for the H.263 example by comparing it with the simulation approach. In the second part, we show the effect of buffer size on the execution cycles and the accuracy of

the timing analysis using the JPEG and IS-95 CDMA examples. Since these examples require prohibitively long simulation runtime, we exclude simulation.

First we performed experiments with an H.263 video encoder system as depicted in Figure 10-4(a). It has seven tasks including Source, DCT, Quantizer (Q), De-Quantizer (Q<sup>-1</sup>), IDCT, Motion Predictor, and Variable Length Coder (VLC). Input stream is a sequence of qcif (176 × 144 pixels). We clustered four tasks (DCT, Q, Q<sup>-1</sup>, and IDCT) into a single unit, which we call *Macro Block (MB) Encoder* (see Figure 10-4(b)). In the experiment we used two target architectures: one with point-to-point interconnection (Figure 10-4(c)) and the other with an on-chip shared bus (Figure 10-4(d)). In the target architecture, Source and VLC were mapped on ARM7TDMI microprocessor and MB Encoder and Motion Predictor were mapped on their own dedicated hardware components. On the ARM7TDMI processor, we ran an embedded configurable operating system (eCos) [15] of Redhat Inc.

The execution delay of SW for, operating systems, device driver, ISR, etc. was measured by instruction set simulator (ISS) execution. That of HW tasks was obtained by VHDL simulator execution. And the software communication architecture delay is measured by ISS.

Table 10-1 shows the advantage of considering dynamic SW behavior in the scheduling of task execution and communication. The table presents the execution cycles of the system obtained by cosimulation, ILP without considering dynamic SW behavior, ILP with dynamic SW behavior, and heuristic with dynamic SW behavior, for the two architectures: point-to-point interconnection and on-chip shared-bus. By considering dynamic SW behavior, we

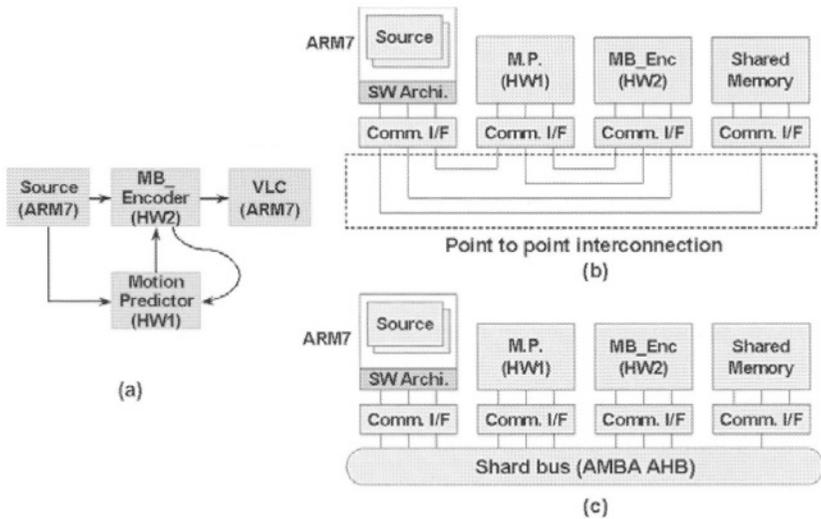


Figure 10-4. (a) Block diagram of h.263 encoder, (b) point-to-point interconnection, and (c) shared bus architecture.

Table 10-1. Execution time of H.263 encoder system.

Architecture	Point-to-point interconnection			Shared bus Execution cycle	Runtime (sec)	Accuracy (%)
	Execution cycle	Runtime (sec)	Accuracy (%)			
Cosimulation	5,031,708	4348	100	5,151,564	29412	100
ILP						
w/o dyn. SW	4,684,812	1.71	93.10	4,804,668	10.17	93.27
w/ dyn. SW	4,979,535	2.23	98.96	5,099,391	14.37	98.98
Heuristic	4,979,535	4.11	98.96	5,099,391	4.23	98.98

could enhance the accuracy from 93.10% to 98.96% for the point-to-point interconnection architecture. Similar enhancement was obtained for the other architecture. We could not obtain 100% accuracy because we did not consider some dynamic behaviors of task VLC and OS scheduler. In this example, the schedule obtained by the heuristic algorithm was the same as that obtained by ILP. That is why we obtained the same accuracy. Table 11-1 shows also the runtimes of cosimulation, ILP, and heuristic algorithm in seconds. In case of point-to-point interconnection, heuristic runtime is larger than that of ILP. That is because the number of nodes in ETG is so small that the complexity depends on constants rather than the number of nodes. Note also that the cosimulation was performed for the schedule obtained by ILP and the runtime does not include the scheduling time.

Then we experimented with larger examples including JPEG and IS-95 CDMA. Table 10-2 shows a comparison of system execution times for different sizes of physical buffers. The third column shows the numbers of nodes in ETG and the last column shows the errors obtained by comparing with ILP optimal scheduling.

As Table 10-2 shows, as the size of physical buffer gets smaller, the system execution time increases. It is due to the synchronization delay caused by the conflicts in accessing the shared buffer. As the table shows, the ILP solutions without dynamic SW behavior give the same number of system execution cycles for three different sizes of shared buffer. This implies that without considering the dynamic SW behavior, such conflicts are not accounted for during the scheduling. As shown in the table, the ILP without the dynamic SW behavior yields up to 15.13% error compared to the ILP with the dynamic SW behavior. Thus, to achieve accurate scheduling of communication and task execution, the dynamic SW behavior needs to be incorporated into the scheduling.

Table 10-2. Execution time and errors for JPEG and IS-95 CDMA modem example.

Examples	Methods	Number of nodes	Buffer size	Execution cycles	Runtime (sec)	Errors (%)
JPEG	ILP w/o dyn. SW	30	64	141,249	200.29	-4.15
	ILP w/ dyn. SW	30	64	147,203	11.43	0
	Heuristic	30	64	147,203	0.47	0
	ILP w/o dyn. SW	30	32	141,249	194.30	-7.78
	ILP w/ dyn. SW	30	32	153,157	70.04	0
	Heuristic	30	32	153,157	0.51	0
	ILP w/o dyn. SW	30	16	141,249	196.16	-14.43
	ILP w/ dyn. SW	30	16	165,065	15.02	0
	Heuristic	30	16	165,065	0.56	0
IS-95 CDMA modem	ILP w/o dyn. SW	49	64	1,288,079	156.26	-1.37
	ILP w/ dyn. SW	49	64	1,305,941	471.20	0
	Heuristic	49	64	1,305,941	4.68	0
	ILP w/o dyn. SW	49	32	1,333,226	161.93	-15.13
	ILP w/ dyn. SW	49	32	1,570,771	482.70	0
	Heuristic	49	32	1,570,771	5.29	0

## 5. CONCLUSION

On-chip communication design needs SW part design (operating system, device driver, interrupt service routine, etc.) as well as HW part design (on-chip communication network, communication interface of processor/IP/memory, shared memory, etc.). In this work, we tackle the problems of on-chip communication scheduling that includes SW dynamic behavior (interrupt processing and context switching) and HW buffer sharing. To resolve the problems, we present an ILP formulation that schedules task execution and communication on HW/SW on-chip communication architectures as well as a heuristic scheduling algorithm.

We applied our method to H.263 encoder, JPEG encoder, and IS-95 CDMA systems. Our experiments show that by considering the dynamic SW behavior, we can obtain the accuracy of scheduling of about 99%, which is more than 5%~15% improvement over naive analysis without considering the dynamic SW behavior.

## REFERENCES

1. J. Brunel, W. Kruijtzter, H. Kenter, F. Petrot, L. Pasquier, and E. Kock. "COSY Communication IP's." In *Proceedings of Design Automation Conference*, pp. 406-409, 2000.
2. W. Cesario, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. Jerayya, and M. Diaz-Nava. "Component-based Design Approach for Multicore SoCs." In *Proceedings of Design Automation Conference*, June 2002.

3. Jon Kleinsmith and Daniel D. Gajski. "Communication Synthesis for Reuse." UC Irvine, Technical Report ICS-TR-98-06, February 1998.
4. T. Yen and W. Wolf. "Communication Synthesis for Distributed Embedded Systems." In *Proceedings of ICCAD*, 1995.
5. R. B. Ortega and G. Borriello. "Communication Synthesis for Distributed Embedded Systems." In *Proceedings of ICCAD*, 1998.
6. M. Gasteier and M. Glesner. "Bus-Based Communication Synthesis on System-Level." *ACM Transactions on Design Automation of Electronic Systems*, Vol. 4, No. 1, 1999.
7. K. Lahiri, A. Raghunathan, and S. Dey. "System-Level Performance Analysis for Designing On-Chip Communication Architecture." In *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, June 2001.
8. P. Knudsen and J. Madsen. "Integrating Communication Protocol Selection with Hardware/Software Codesign." In *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, August 1999.
9. ARM, Inc. AMBA™ Specification (Rev 2.0), available in <http://www.arm.com/>.
10. Sonics Inc., "Sonics Integration Architecture," available in <http://www.sonicsinc.com/>.
11. W. J. Dally and B. Towles. "Route Packet, Not Wires: On-Chip Interconnection Networks." In *Proceedings of Design Automation Conference*, June 2001.
12. A. Siebenborn, O. Bringmann, and W. Rosenstiel. "Worst-Case Performance of Parallel, Communicating Software Processes." In *Proceedings of CODES 2002*, 2002.
13. Cadence Inc., VCC available at <http://www.cadence.com>.
14. Amer Baghdadi, Nacer-Eddine Zergainoh, Wander O. Cesário, and Ahmed Amine Jerraya. "Combining a Performance Estimation Methodology with a Hardware/Software Codesign Flow Supporting Multiprocessor Systems." *IEEE Transactions on Software Engineering*, Vol. 28, No. 9, pp. 822–831, September 2002 .
15. Redhat Inc., eCos, available in <http://www.redhat.com/embedded/technologies/ecos/>.

## Chapter 11

# EVALUATION OF APPLYING SPECC TO THE INTEGRATED DESIGN METHOD OF DEVICE DRIVER AND DEVICE

Shinya Honda and Hiroaki Takada

*Information and Computing Sciences, Toyohashi University of Technology, 1-1 Hibarigaoka, Tempaku-cho, Toyohashi City, Aichi pref., 441-8580 Japan*

**Abstract.** We are investigating an integrated design method for a device driver and a device in order to efficiently develop device drivers used in embedded systems. This paper evaluates whether SpecC, which is proposed as a system level description language, is applicable to integrated description language for the integrated design method. We use an SIO system to confirm the feasibility of using SpecC for integrating a device and description device driver. We manually convert the SpecC description to the device, the device driver and the interface in between and confirm that the conversion can be automated. We also confirm the feasibility of conversion when the partition point between the software and the hardware is changed. As a result, we show that SpecC could apply as a integrated design language of the design method.

**Key words:** co-design, RTOS, device driver

### 1. INTRODUCTION

Embedded systems are becoming complicated and largely scaled. In addition to the increase in development time and costs, the degrading design quality and system reliability have been a major problem for developers. Since embedded systems are designed for specific target hardware mechanism, each system has different hardware configuration and peripheral devices. Therefore, a device driver that directly control devices needs to be developed for each system. Development of device drivers occupy a high proportion of software development in embedded system [1].

Developing a device driver is difficult and takes a long time regardless of the small code size. This can be mainly attributed to the lack of communication between hardware designers and software designers. Device manuals mainly focus on explanations of device registers and they lack explanations from a software viewpoint regarding control methods that are necessary for device driver development. The information provided by the device driver manual is not enough to design a device driver and in most cases, the device interface is designed without considering the device driver development situation, so the device driver development gets more difficult.

Another cause is that verification and debugging of a device driver is difficult since a device driver must operate under the timing determined by the device. In general, verification of time-dependent software takes a long time since the operation must be verified for every possible timing. Moreover, even though an error occurred in a software, finding the bug is difficult since the repetition of a bug is low.

To solve this problem, we have conducted a research on an integrated design method of a device driver (Software) and a device (Hardware) for the efficient development of device drivers for embedded systems. This method uses system level design and co-design technologies to describe the device and the device driver as a unit in one language and generate the implementation descriptions from the integrated description. This method is expected to improve the communication between device designers and device driver designers and to increase the efficiency of device driver development. Also another merit that can be expected by applying co-design technology is that the border between the device and the device driver can be flexibly changed.

In this paper, we evaluate whether SpecC [2] is applicable to the integrated description in the proposed design method. Applicability means the ability of SpecC to describe the device and the device driver as a unit and automatically generate implementation descriptions from the integrated description. SpecC is C language-based with additional concepts and statements for enabling hardware descriptions. It is proposed as a language to cover the range from system level description to implementation design. However, the semantics of the added concept and statements were mainly discussed from the viewpoint of hardware description and hardware generation and the method to generate the software using a real-time kernel was not considered. Since a real-time kernel is essential to construct software on a large scale system LSI, the device driver generated by the proposed design method is assumed to use a real-time kernel. Thus, evaluating whether SpecC is applicable to the proposed design method is necessary.

To evaluate the applicability of SpecC, we use a serial I/O system (SIO system) that includes the Point-to-Point Protocol (PPP) function used in sending a TCP/IP packet on a serial line. We choose the SIO system in our evaluation because it has a simple but typical architecture used in communication systems which is important application of system LSI.

The rest of the paper is organized as follows: Section 2 presents the proposed integrated design method of a device driver and a device. Section 3 describes evaluation points and methods to evaluate the applicability of SpecC to the integrated design method. Section 4 describes description of the SIO system in SpecC and SpecC description guidelines. Section 5 discusses the conversion of SpecC description to implementation description. In Section 6, we shows the conversion when partition point is changed.

## 2. INTEGRATED DESIGN METHOD

There are works being conducted for efficient development of device drivers for embedded systems, such as design guidelines for device driver design and research on generating device drivers automatically [3, 4]. However, design guidelines only solve the problem partially and in automatic generation, the description method of the device driver and device specifications is a problem. Moreover, these approaches only deal with existing device and since in most cases in embedded systems, a device is designed for each system, these approaches are ineffective.

As a more effective approach for increasing the efficiency of the device driver development, we conduct a research on integrated design method that describes the device and the device driver as a unit and generate implementation descriptions from the integrated description. By describing the device and the device driver as a unit, the communication between device designers and device driver designers can be improved, and the device design in consideration of the device driver development situation is enhanced. Moreover if the integrated description is executable, it can be useful to verify the model at an early development stage.

The design flow of the proposed method is shown in Figure 11-1.<sup>1</sup> The first describes the device driver and the device using SpecC. This description is divided into the portion that is implemented in software and the portion that

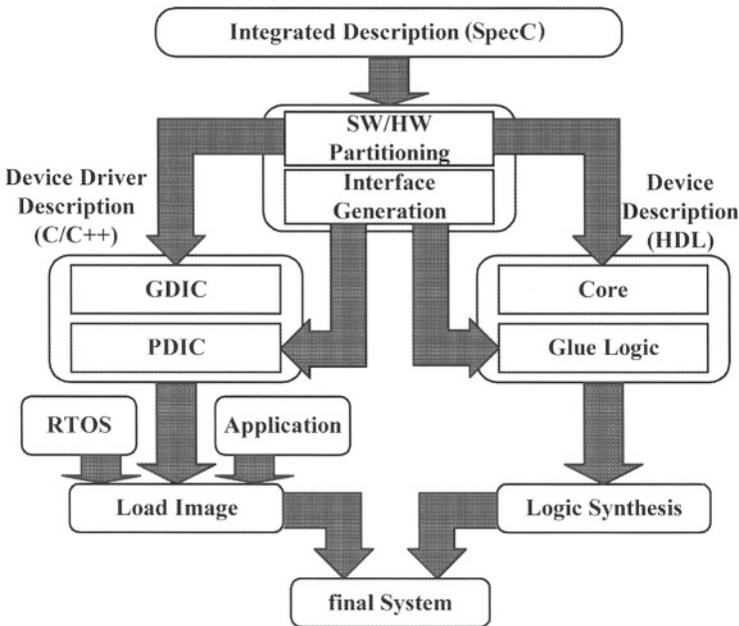


Figure 11-1. Design flow of the integrated design method.

is implemented in hardware and generate interface between them. In the present condition, although the designer have to specify the partition point, the automation could be possible by development of co-design technology in the future.

The part implemented through software is converted to C/C++ description. The generated description and the generated software interface (PDIC) become the description of the device driver. The statement for parallelism, synchronization and communication offered by SpecC are converted to tasks and system calls in real-time kernel. The part implemented through hardware are converted to HDL. The generated HDL and the generated hardware interface (the glue logic) become the description of the device. In this research, the conversion technique of SpecC to HDL is not considered as one of the main objectives and it is taken from the work of other researchers [5].

### 3. EVALUATION POINTS AND EVALUATION METHODS

The points for evaluating the applicability of SpecC as the integrated description language for the proposed design method are shown below.

1. The feasibility of describing the device and the driver as a unit.
2. The feasibility of mechanically converting the description of parallelism, synchronization and communication offered by SpecC to a device driver or a software-hardware interface.
3. The feasibility of converting the integrated description to implement description when the partition point between the software and the hardware is changed.

This research only evaluates the conversion technique from a SpecC description to a device driver and not the conversion technique from a SpecC description to a device.

To evaluate point (1), we describe the device and the device driver that constitute the SIO system as a unit using SpecC. A conceptual figure of the SIO system is shown in Figure 11-2. In addition to the basic serial transmit and receive functions, the system also has generation and analysis functions

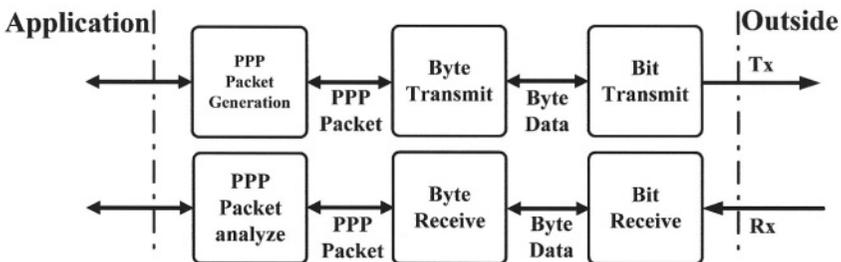


Figure 11-2. Overview of the SIO system.

for PPP packet. The serial communication method, baud rate, byte size and stop bit are: asynchronous, 19200 bps, 8 bits and 1, respectively. Parity is not used.

To evaluate point (2), we manually convert the SpecC description to the device, the device driver and the interface according to the proposed design method. The real-time kernel used by the converted software is a  $\mu$ ITRON4.0 specification kernel [6].

To evaluate point (3), we use the same method described in point (2) but we change the partition point between the software and the hardware of the SIO system described in SpecC.

#### 4. SPECC DESCRIPTION

Figure 11-3 shows the transmitter part and receiver part of SIO system in SpecC. The rounded squares represent the behaviors while the ellipses represent the channels. The description of the behaviors and the functions of both the receiver and the transmitter, and the channel consist of 589 lines (17.2 kbyte).

The transmitter consists of the PPP generation behavior (tx\_ppp), the byte transmit behavior (tx\_byte), the bit transmit behavior (tx\_bit), the baud rate clock generator (tx\_clk), and the 16-time baud rate clock generator (gen\_clk16). These behaviors are executed parallel. The application, tx\_ppp, tx\_byte and tx\_bit byte are connected by channels as shown in Figure 11-4.

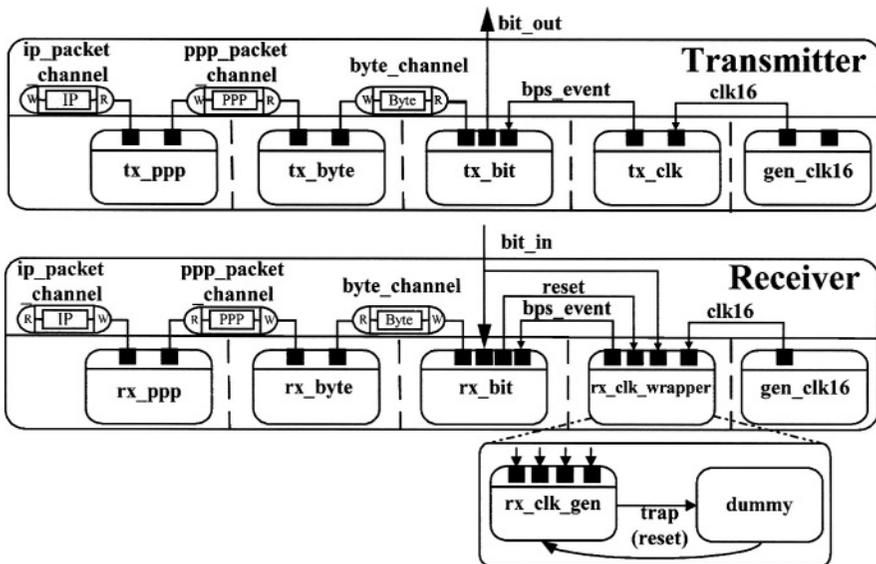


Figure 11-3. SIO system in SpecC.

```

channel byte_channel(void) implements byte_interface{
    unsigned char data_buffer; /* Buffer */
    bool data_valid;          /* Buffer Status */
    evnet sync_write, sync_read /* Event for writeing/reading */
    void write(unsigned char data){
        while(data_valid)
            wait(sync_read);
        data_buffer = data;
        data_valid = true;
        notify(sync_write);
    }
    unsigned char read(void){
        while(!data_valid)
            wait(sync_write);
        data_valid = false;
        notify(sync_read);
        return(data_buffer);
    }
}

```

Figure 11-4. Byte channel (byte\_channel).

At the start of the execution, the three behaviors (tx\_ppp, tx\_byte and tx\_bit) waits for data from the upper layer. When tx\_ppp receives an IP packet from the application, it generates and sends a PPP packet to tx\_byte. Tx\_byte sends the PPP packet one byte at a time to tx\_bit. tx\_bit then transmit the data bit by bit to a serial line(bit\_out). The transmission to a serial line is synchronized with the event from tx\_clk (bps\_event). The above mentioned process is repeated infinitely.

The receiver is also divided into behaviors which are executed parallel. The PPP analysis behavior(rx\_ppp) and the byte receive behavior(rx\_byte) wait for data from the lower layer. Detection of the falling edge of the serial line for start bit detection is performed by the receive clock generation behavior(rx\_clk\_gen). When rx\_clk\_gen detects the falling edge, it sends the event(bps\_event) to the bit receive behavior(rx\_bit) after a half cycle of the baud rate. The rx\_clk\_gen then continues sending the event at a cycle equal to the baud rate. When rx\_bit receives the first event from rx\_clk\_gen, rx\_bit, it determines whether the data on the serial line is a start bit. If the data is a start bit, rx\_bit fetches the data on the serial line synchronized with the event from rx\_clk\_gen. If the data is not a start bit, tx\_bit resets rx\_clk\_gen for further detection of a start bit. rx\_bit also resets rx\_clk\_gen after receiving 1 byte of data. The reset functionality of rx\_clk\_gen was realized using *try-trap-interrupt* statement, wrapper behavior and dummy behavior. In the statement

block in *trap*, a behavior needs to be described so the dummy behavior which does nothing is used.

Three kinds of channels are used for each description of the transmitter part and the receiver part. Since data is passed one byte at a time between `tx_byte` and `tx_bit`, and `rx_byte` and `rx_bit`, raw data is directly copied between these behaviors. However, pointers are used to pass IP packet and PPP packet between the application and `tx_ppp`, `tx_ppp` and `tx_byte` to avoid the data copy. The SpecC code of a byte channel which delivers 1-byte data is shown in Figure 11-4. PPP packet channels (`ppp_packet_channel`) and IP packet channels (`ip_packet_channel`) also use same structure as a byte channel except for the type of data to be delivered.

To confirm the SIO system description, the serial lines of the transmitter and the receiver are connected (loop back) and simulation is performed using the test bench where `tx_ppp` transmits an IP packet to `rx_ppp`. SpecC reference compiler (SCRC 1.1) is used for the simulation environment. The results of the simulation show that the SIO system description operates correctly.

Despite the redundancy in the description of the reset function, the SIO system has been described using parallelism and synchronization and communication statements in SpecC. Thus, SpecC satisfies evaluation point (1) described in section 3.

#### 4.1. SpecC description guidelines

In the SIO system description, system functions with equal execution timing are included in the same behavior. The system functions are grouped into behaviors according to their activation frequencies. The point where the execution frequency changes is considered as a suitable point for software and hardware partitioning so the behaviors in Figure 11-3 are arranged according their activation frequency, with the right most behavior having the highest frequency.

The main purpose of introducing channels to SpecC is to make the functional descriptions independent from the description of synchronization and communication by collecting synchronization and communication mechanism to a channel. In the SpecC description of the SIO system, the descriptiveness and readability of the behavior has improved since the behavior contains only its functionality. Describing the behaviors that needs to communicate with the upper and the lower layer, like `tx_ppp` and `tx_byte`, without using channels is difficult. Thus, synchronization and communication mechanism should be collected and described in a channel where possible. Collecting synchronization and communication functionalities to a channel also makes the conversion of a software from a SpecC description efficient. This is further described in Section 5.

## 5. CONVERSION OF SPECC DESCRIPTION

This section discusses the conversion technique from a SpecC description to a device, a device driver and the interface in between. For the conversion of a device driver and interface, we use evaluation point (2) described in section 3. The partition of the software and the hardware is the byte channel. The description of the layer above the byte channel is converted into a device driver description in C-language program and the description of the layer below the byte channel is converted into a device description in VHDL and the byte channel is converted into an interface in between. The conversions are made mechanically where possible. The converted software part (device driver) in C-language is linked with  $\mu$ ITRON4.0 specification real-time kernel. The converted hardware part (device) in VHDL is synthesized for FPGA. The behavior of the implementations are verified using a prototype system with processor and FPGA. The results show that the behaviors of the implementations are correct.

### 5.1. Device driver

In the SIO system, the object of device driver conversion is the description ranging from the PPP generation and analysis behavior to the byte transmit and receive behavior.

The SpecC behaviors can be classified into two: parallel execution and used functionally. Behaviors executed in parallel are converted into tasks and the other behaviors are converted into functions. Specifically, behaviors that are executed in parallel according to the *par* statement, are converted to tasks. In order to suspend the behavior until all the other behaviors executed through the *par* statement have been completed, the parent task executing the *par* statement is moved to the waiting state until all child tasks have completed execution.

The *try-trap-interrupt* statement is converted into a description that uses the task exception-handling function of  $\mu$ ITRON. The task corresponding to the behavior described in the *try* statement block is generated and an exception demand to a task realizes the event. The exception handling routine calls functions that are converted from the behavior described in the *trap* statement block and the *interrupt* statement block. Other behaviors such as behaviors executed sequentially and behaviors within the *fsm* statement used to describe FSM are converted into functions. In the SIO system, the PPP generation and analysis behavior and the byte transmit and receive behavior are each converted to one task since they are executed in parallel and do not call any other behavior.

A channel can be mechanically converted into a software by realizing the atomic executions of methods encapsulated in channels, event triggers (*notify* statement) and suspension of execution (*wait* statement) using semaphores and eventflags provided in  $\mu$ ITRON. However, since the  $\mu$ ITRON provides

high-level synchronization and communication mechanisms, such as data queues, the channels in the SIO system should be converted into data queues in order to improve the execution speed. Hence, the PPP packet channel and the IP packet channel are realized as data queues.

## **5.2. Device and interface**

The descriptions of layers lower than the bit transmit and receive behaviors are converted into the Finite State Machine in VHDL. The behaviors, together with the SpecC descriptions, in the layer lower than the bit transmit and receive behavior are clock-synchronized. As a result, a large part of the SpecC description can be mechanically converted into a VHDL description of the target device. However, the conversion of SpecC descriptions that are not synchronized with the clock is a difficult job.

Since the SpecC description of the SIO system are written according to the description guidelines as described above, the partition point between hardware and software should be a channel. The channel between the behavior converted into software and behavior converted into hardware is considered as the interface between hardware and software. The details of interface conversion are described in [7].

## **5.3. Conversion result and discussion**

Figure 11-5 shows the relation between the transmitter of the SIO system in SpecC and the software and hardware obtained from the above mentioned conversion. Since the receiver has almost the same structure as the transmitter in the SpecC description, the resulting structures after conversion are similar with each other. The transmitter and the receiver are sharing a bus interface, an address decoder and an interrupt generator. The hardware part with the receiver uses 309 cells in FPGA and a maximum frequency of 44 MHz.

The mechanical conversion of a behavior to a software consists of finding out whether the behavior should be converted to a task or a function. The conversion rule described in above can be realized by an analysis of SpecC description, therefore the mechanical conversion is feasible.

A channel conversion is possible using synchronization and communication functions provided by  $\mu$ ITRON. However, judging that the code shown in Figure 11-4 can be realized using data queues is not possible mechanically. Hence, softwares generated by mechanical conversions are not always efficient. One efficient way of converting frequently used channels with similar functions to efficient softwares is to prepare a channel library. An efficient mechanical conversion is possible by preparing the conversion method which uses the synchronization and communication functions provided by  $\mu$ ITRON for the channel in the library. The channels included in library should be able to handle arbitrary data types like C++ STL, since the data types that are passed through the channel differ with each situation.

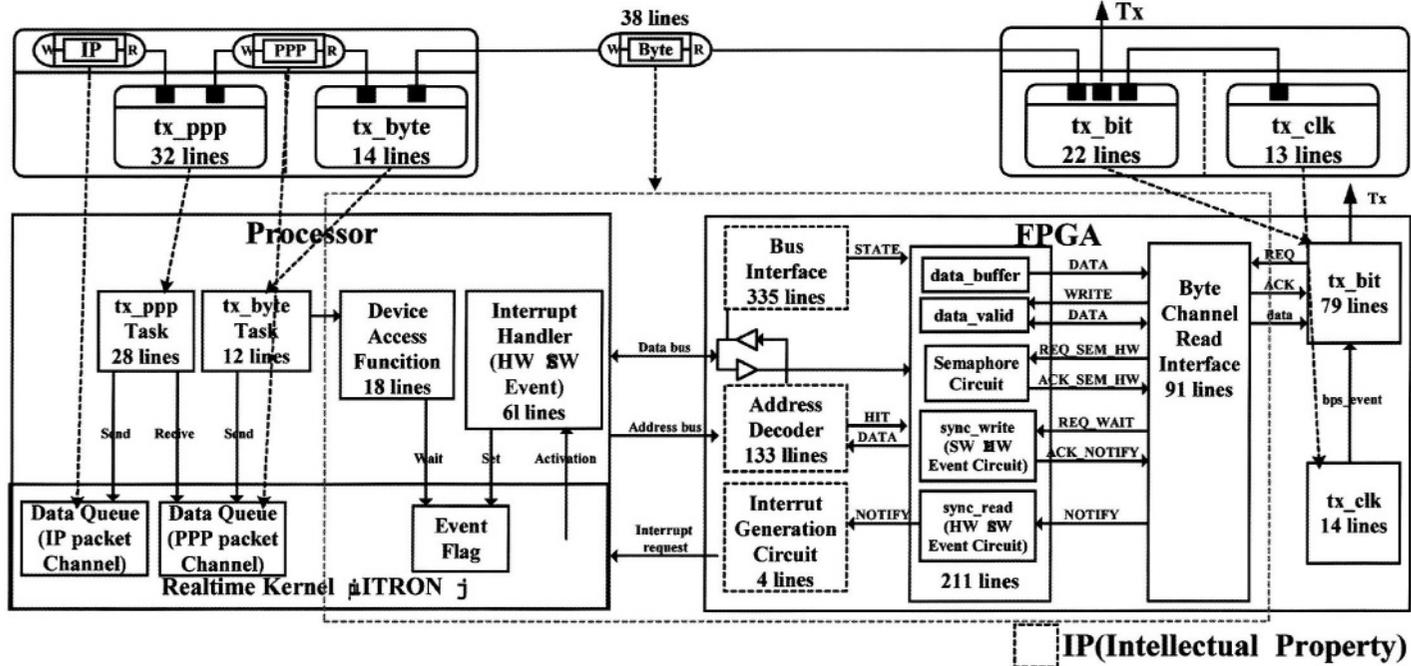


Figure 11-5. The relation between the transmitter in SpecC and implementation model partitioned at byte channel.

As described in [7], the interface on the software side can be converted mechanically. For the interface on the hardware side, mechanical conversion is possible by preparing a bus interface circuit for bus models. By preparing a channel library and an interface description for every channel included in library, the mechanical conversion can be made more efficient. Since we were able to show the mechanical conversion methods of SpecC description to softwares and interface, we conclude that evaluation point (2) is satisfied.

## 6. CHANGE OF PARTITION POINT

The optimal partition point between a software and a hardware changes with the demands and restrictions of a system. The advantage of the conversion from integrated description is that the partition point of the device and the device driver can be changed according to the need of the system. To be able to change the partition point, a single behavior description should be converted into both software and hardware.

To evaluate point (3) described in section 3, the partition point of the software and the hardware is changed from a byte channel to a PPP packet channel previous conversion. The difference between the byte channel conversion and the PPP packet channel conversion is that the device that receives the pointer has to take out the data from the memory using direct memory access (DMA) because the PPP packet channel only passes the pointer to the PPP packet.

### 6.1. Device driver and interface

For the SIO system, the objects of the device driver conversion are descriptions of IP packet channels and PPP generation and analysis behaviors. As described in section 5, the conversion can be done mechanically.

The PPP packet channels can be mechanically converted to an interface using the method done with a byte channel. The functions of the channel for PPP packet are equivalent to the functions of the byte channel. The only difference is that in the conversion of PPP packet channels, the type *data\_buffer* is used so after the conversion the size of the device register used to hold the a *data\_buffer* is different from that of the byte channel.

### 6.2. Conversion of the device

This section describes the conversion of the byte channel and the byte transmit behavior to the device. For description on the layer below the bit transmit/receive behavior, conversion is the same as described in section 5. Since the PPP packet channel transmits and receives pointers to PPP packet, the byte transmit/receive behavior includes functions to access memory. To convert memory access with pointers, the memory access is done using DMA. The

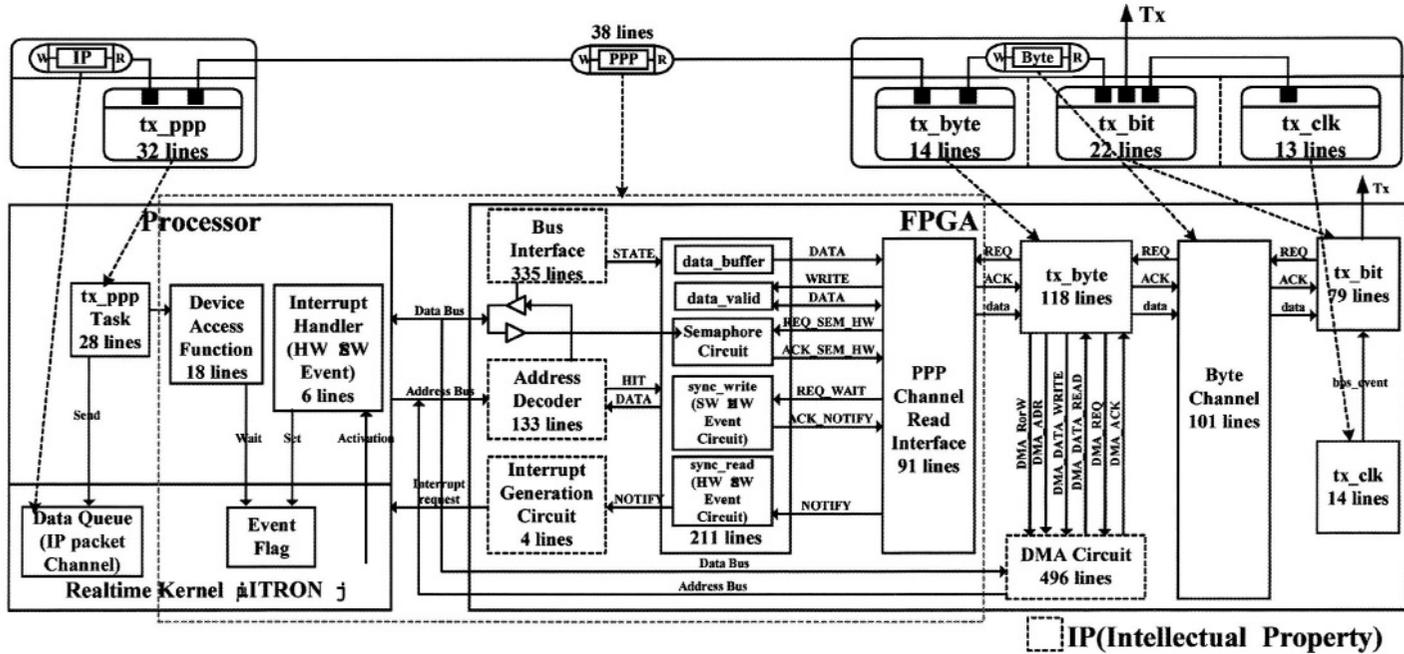


Figure 11-6. The relation between the transmitter in SpecC and implementation model partitioned at PPP packet channel.

Colour picture

DMA circuit is realized by using intellectual property (IP). However, there are systems where the device cannot be the bus master or it cannot access a part of memory used by the processor. In these kinds of systems the DMA cannot be realized.

Since the partition point is changed, the byte channel is converted into a hardware. The channel conversion rule to the hardware is described in [7].

### **6.3. Conversion result and discussion**

Figure 11-6 shows the relation between the transmitter part of SIO system in SpecC and the software and hardware obtained from the above conversion. The implementation behaves correctly on the prototype system. The hardware part with the receiver uses 951 cells in FPGA and a maximum frequency of 40 MHz.

By using a DMA circuit, changing the partition point of the SIO system into a PPP packet channel is possible. The results show that the number of cells in the FPGA implementation of the conversion with PPP packet channel is 3 times that of the conversion with the byte channel. This is because a DMA is included and the functions realized as hardware increased. Furthermore, since the interface converted from PPP packet channel delivers 32 bit data while the interface converted from the byte channel delivers only 8 bit data, the data size handled in the glue logics increased 4 times compared with the previous conversion. In this conversion, the behavior and the channel are converted to hardware module one to one. However, the behavior and the channel can be merged into one module for optimization.

One problem of realizing the hardware using DMA is that the head address of the structure may vary when it is accessed from the processor or from the device. This is because the connections of the memory, the processor and the DMA circuit vary or the processor performs a memory translation. This problem can be solved by translating the address passed by the software to the address seen by the device.

## **7. CONCLUSION**

In this paper, we have presented an integrated design method of a device and a device driver and we have evaluated the applicability of SpecC to the proposed method using the SIO system.

Evaluations have shown that the SIO system can be mechanically converted by following the description guidelines and preparing a library for channels. Since the SIO system can be described using parallelism, synchronization and communication statements in SpecC, we have verified that SpecC can be used as the description language for the proposed integrated design method of a device driver and a device.

Furthermore, we have verified that the conversion by changing the parti-

tion point of the software and the hardware from a single SpecC description is also possible. The conversion is possible by using a DMA circuit for realizing memory access with pointers.

## NOTE

<sup>1</sup> PDIC (Primitive Device Interface Component) is minimum interface software handling a device. GDIC(General Device Interface Component) is upper layer software which handles a device depending on a real-time kernel [3].

## REFERENCES

1. Hiroaki Takada. "The Recent Status and Future Trends of Embedded System Development Technology." *Journal of IPSJ*, Vol. 42, No. 4, pp. 930–938, 2001.
2. D. Gajski, J. Zhu, D. Dömer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*, Kluwer Academic Publishers, 2000.
3. K. Muranaka, H. Takada, et al. "Device Driver Portability in the ITRON Device Driver Design Guidelines and Its Evaluation." In *Proceedings of RTP 2001, a Workshop on Real-Time Processing*, Vol. 2001, No. 21, pp. 23–30, 2001.
4. T. Katayama, K. Saisho, and A. Fukuda. "Abstraction of Device Drivers and Inputs of the Device Driver Generation System for UNIX-like Operating Systems." In *Proceedings of 19th IASTED International Conference on Applied Informatics (AI2001)*, pp. 489–495, 2001.
5. D. Gajski, N. Dutt, C. Wu, and S. Lin. *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers, 1994.
6. TRON Association.  $\mu$ ITRON 4.0 Specification (Ver 4.00.00), Tokyo, Japan, 2002. <http://www.assoc.tron.org/eng/>
7. Shinya Honda. *Applicability Evaluation of SpecC to the Integrated Design Method of a Device Driver and a Device*, Master Thesis, Information and Computing Sciences, Toyohashi University of Technology, 2002.

## Chapter 12

# INTERACTIVE RAY TRACING ON RECONFIGURABLE SIMD MORPHOSYS

H. Du<sup>1</sup>, M. Sanchez-Elez<sup>2</sup>, N. Tabrizi<sup>1</sup>, N. Bagherzadeh<sup>1</sup>, M. L. Anido<sup>3</sup>, and M. Fernandez<sup>1</sup>

<sup>1</sup> *Electrical Engineering and Computer Science, University of California, Irvine, CA, USA;*

<sup>2</sup> *Dept. Arquitectura de Computadores y Automatica, Universidad Complutense de Madrid, Spain;*

<sup>3</sup> *Nucleo do Computation, Federal University of Rio de Janeiro, NCE, Brazil;*

*E-mail: {hdu, ntabrizi, nader}@ece.uci.edu, {marcos, mila45}@fis.ucm.es, mlois@nce.ufrj.br*

**Abstract.** MorphoSys is a reconfigurable SIMD architecture. In this paper, a BSP-based ray tracing is gracefully mapped onto MorphoSys. The mapping highly exploits ray-tracing parallelism. A straightforward mechanism is used to handle irregularity among parallel rays in BSP. To support this mechanism, a special data structure is established, in which no intermediate data has to be saved. Moreover, optimizations such as object reordering and merging are facilitated. Data starvation is avoided by overlapping data transfer with intensive computation so that applications with different complexity can be managed efficiently. Since MorphoSys is small in size and power efficient, we demonstrate that MorphoSys is an economic platform for 3D animation applications on portable devices.

**Key words:** SIMD reconfigurable architecture, interactive ray tracing

## 1. INTRODUCTION

MorphoSys [1] is a reconfigurable SIMD processor targeted at portable devices, such as Cellular phone and PDAs. It combines coarse grain reconfigurable hardware with one general-purpose processor. Applications with a heterogeneous nature and different sub-tasks, such as MPEG, DVB-T, and CDMA, can be efficiently implemented on it. In this paper a 3D graphics algorithm, ray tracing, is mapped onto MorphoSys to achieve realistic illumination. We show that SIMD ray-tracing on MorphoSys is more efficient in power consumption and has a lower hardware cost than both multiprocessors and the single CPU approaches.

Ray tracing [2] is a global illumination model. It is well known for its highly computation characteristic due to its recursive behavioral. Recent fast advancement of VLSI technology has helped achieving interactive ray tracing on a multiprocessor [3] and a cluster system [9, 10] for large scenes, and on a single PC with SIMD extensions [4] for small scenes. In [3], Parker achieves 15 frames/second for a  $512 \times 512$  image by running ray tracing on a 60-node (MIPS R12000) SGI origin 2000 system. Each node has a clock faster than

250 MHz, 64-bit data paths, floating-point units, 64 K L1 cache, 8 MB L2 cache, and at least 64 MB main memory. Muuss [9, 10] worked on parallel and distributed ray tracing for over a decade. By using a cluster of SGI Power Challenge machines [10], a similar performance as Parker's is reached. Their work is different in their task granularity, load balancing and synchronization mechanisms. The disadvantage of their work is that there are extra costs such as high clock frequency, floating-point support, large memory bandwidth, efficient communication and scheduling mechanisms. Usually, the sub-division structure (such as BSP tree *Binary Space Partitioning* [7, 8]) is replicated in each processor during traversal. As will be seen, only one copy is saved in our implementation.

Wald [4] used a single PC (Dual Pentium-III, 800 Mhz, 256 MB) to render images of  $512 \times 512$ , and got 3.6 frames/second. 4-way Ray coherence is exploited by using SIMD instructions. The hardware support for floating-point, as well as the advanced branch prediction and speculation mechanisms helps speed up ray tracing. The ray incoherence is handled using a scheme similar to multi-pass scheme [14], which requires saving intermediate data, thus causing some processors to idle.

The migration from fixed-function pipeline to programmable processors also makes ray tracing feasible on graphics hardware [5, 6]. Purcell [6] proposes a ray-tracing mapping scheme on a pipelined graphics system with fragment stage programmable. The proposed processor requires floating-point support, and intends to exploit large parallelism. Multi-pass scheme is used to handle ray incoherence. As a result, the utilization of SIMD fragment processors is very low (less than 10%). Carr [5] mapped ray-object intersection onto a programmable shading hardware: Ray Engine. The Ray Engine is organized as a matrix, with vertical lines indexed by triangles, and horizontal lines by rays represented as pixels. Data is represented as 16-bit fixed-point value. Ray cache [5] is used to reorder the distribution of rays into collections of coherent rays. They got 4 frame/sec for  $256 \times 256$  images but only for static scenes. Although still on its way to achieve interactivity, all these schemes represent one trend towards extending the generality of previous rasterization-oriented graphics processors [10].

In this paper, a novel scheme that maps ray tracing with BSP onto MorphoSys is discussed. Large parallelism is exploited such that 64 rays are traced together. To handle the problem that the traversal paths of these rays are not always the same, a simple and straightforward scheme is implemented. Two optimizations, object test reordering and merging, are utilized. The overhead caused by them is very small after being amortized in highly paralleled ray tracing. Compared with the other schemes, MorphoSys achieves 13.3 frames/second for  $256 \times 256$  scenes under 300 MHz with a smaller chip size, with a simple architecture (SIMD, fixed-point, shared and small memory), with power efficient performance (low frequency), and highly parallel execution (64-way SIMD). This justifies MorphoSys as an economic platform for 3D game applications targeted at portable devices.

The paper begins with an architecture overview of MorphoSys. Section 3 describes ray tracing pipeline process, followed by the SIMD BSP traversal in Section 4. Some design issues, such as data structure, emulated local stack, and memory utilization are detailed in Sections 5, 6, 7. Section 8 describes the global centralized control. We give the implementation and results in Section 9, followed by the conclusion in Section 10.

## 2. MORPHOSYS ARCHITECTURE

MorphoSys [1] is a reconfigurable SIMD architecture targeted for portable devices. It has been designed and intended for a better trade-off between generality and efficiency in today’s general-purpose processors and ASICs, respectively. It combines an array of 64 Reconfigurable Cells (RCs) and a central RISC processor (TinyRISC) so that applications with a mix of sequential tasks and coarse-grain parallelism, requiring computation intensive work and high throughput can be efficiently implemented on it. MorphoSys is more power efficient than general-purpose processors because it was designed with small size and low frequency yet to achieve higher performance through highly paralleled execution and minimum overhead in data and instruction transfers and reconfiguration time. The detailed design and implementation information of the MorphoSys chip are described in [16]. The general architecture is illustrated in Figure 12-1.

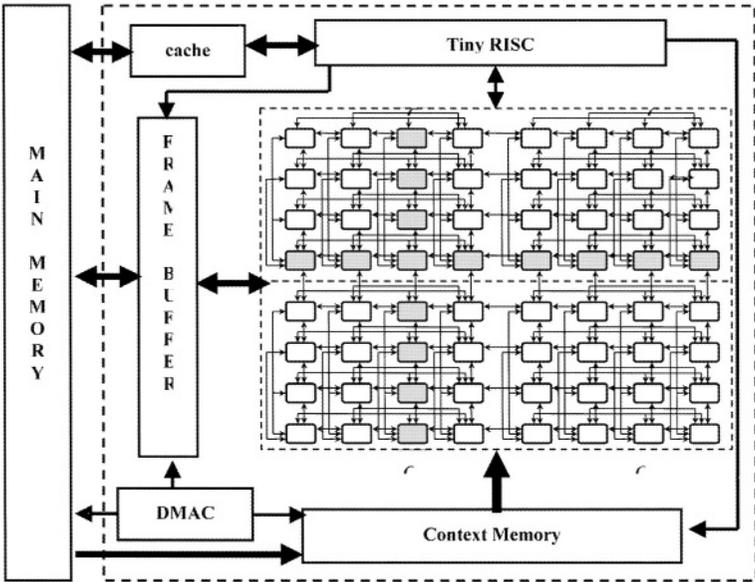


Figure 12-1. MorphoSys SIMD architecture.

The kernel of MorphoSys is an array of 8 by 8 RCs. The RC array executes the parallel part of an application, and is organized as SIMD style. 64 different sets of data are processed in parallel when one instruction is executed. Each RC is a 16-bit fixed-point processor, mainly consisting of an ALU, a 16-bit multiplier, a shifter, a register file with 16 registers, and a 1 KB internal RAM for storing intermediate values and local data.

MorphoSys has very powerful interconnection between RCs to support communication between parallel sub-tasks. Each RC can communicate directly with its upper, below, left and right neighbors peer to peer. One horizontal and one vertical short lane extend one RC's connection to the other 6 RCs in row wise and column wise in the same quadrant (4 by 4 RCs). And one horizontal and one vertical global express lane connect one RC to all the other 14 RCs along its row and column in the 8 by 8 RC array.

The reconfiguration is done through loading and executing different instruction streams, called "contexts", in 64 RCs. A stream of contexts accomplishes one task and is stored in Context Memory. Context Memory consists of two banks, which flexibly supports pre-fetch: while the context stream in one bank flows through RCs, the other bank can be loaded with a new context stream through DMA. Thus the task execution and loading are pipelined and context switch overhead is reduced.

A specialized data cache memory, Frame Buffer (FB), lies between the external memory and the RC array. It broadcasts global and static data to 64 RCs in one clock cycle. The FB is organized as two sets, each with double banks. Two sets can supply up to two data in one clock cycle. While the contexts are executed over one bank, the DMA controller transfers new data to the other bank.

A centralized RISC processor, called TinyRISC, is responsible for controlling RC array execution and DMA transfers. It also takes sequential portion of one application into its execution process.

The first version of MorphoSys, called *M1*, was designed and fabricated in 1999 using a 0.35  $\mu\text{m}$  CMOS technology [16].

### 3. RAY TRACING PIPELINE

Ray tracing [2] works by simulating how photons travel in real world. One *eye ray* is shot from the viewpoint (eye) backward through image plane into the scene. The objects that might intersect with the ray are tested. The closest intersection point is selected to spawn several types of rays. *Shadow rays* are generated by shooting rays from the point to all the light sources. When the point is in shadow relative to all of them, only the ambient portion of the color is counted. The *reflection ray* is also generated if the surface of the object is reflective (and *refraction ray* as well if the object is transparent). This reflection ray traverses some other objects, and more shadow and reflection rays may be spawned. Thus the ray tracing works in a recursive way. This

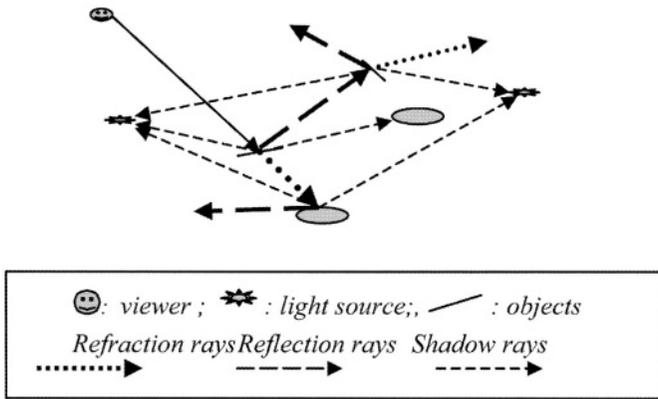


Figure 12-2. Illustration of ray tracing.

recursive process terminates when the ray does not intersect with any object, and only background color is returned. This process is illustrated in Figure 12-2.

Ray tracing is basically a pipelined process. The algorithm is separated into four steps. First, rays are generated. Then each ray traverses BSP tree to search for the object with the closest intersection point. This is an iterative process in term of programming model, where BSP tree nodes are checked in depth-first-order [11]. Once a leaf is reached, the objects in this leaf are scanned. If no intersection in this leaf or the intersection point is not in the boundary of this leaf, BSP tree is traversed again. Finally, when the closest point is found, the shading is applied, which recursively generates more rays. This pipeline process is illustrated in Figure 12-3.

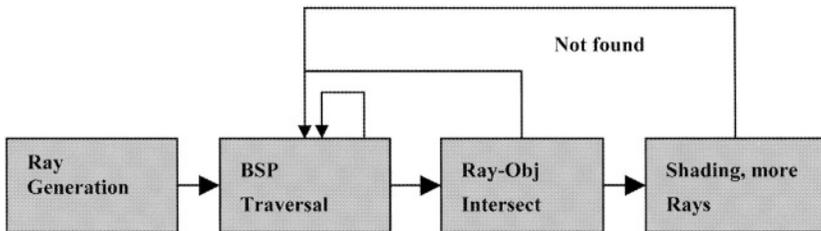


Figure 12-3. Ray tracing pipeline.

#### 4. SIMD BSP TRAVERSAL

The ray object intersection occupies more than 95% of ray tracing time. Some techniques have been developed to accelerate it. BSP [7, 8] is one of them. It tries to prevent those objects lying far away from being tested. BSP recur-

sively partitions a 3D cube into 2 sub-cubes, defined as left and right children. BSP works like a Binary-Search Tree [15]. When one ray intersects with one cube, it tests whether it intersects with only left, right, or both children. Then the ray continues to test these sub-cubes recursively. The traversal algorithm stops when the intersection is found or when the tree is fully traversed.

The efficiency may be sacrificed in SIMD ray tracing. Each BSP-tree ray traversal involves many conditional branches, such as *if-then-else* structure. *Program autonomous* [12] is thus introduced to facilitate them in a SIMD. We implemented *pseudo branch* and applied *guarded execution* to support conditional branch execution [13].

During parallel BSP traversal different rays may traverse along the same BSP tree path (“*ray coherence* [15]”) or along different BSP tree paths (“*ray incoherence*”). The coherence case is easily handled in a SIMD architecture.

However, the incoherence case demands a high memory bandwidth because different object data and also different object contexts are required concurrently. It can be further observed that two types of ray incoherence exist:

- First type, the ray incoherence occurs in an internal tree node, and not all rays intersect with the same child of the current node. In this case, BSP traversals of all rays are stopped and all objects under this node are tested.
- Second type, the ray incoherence occurs in a leaf, where not all rays find the intersection points in that leaf. In this case the RCs that find the intersections are programmed to enter the sleep mode (no more computation), while the others continue the BSP traversal. Power is saved in such a way.

Sometimes, some rays may terminate traversal earlier and may start ray-object intersection while others are still traversing. In this case, different context streams are required for different rays. In [6], the extended *multi-pass* [14] scheme is used to address this problem. In their scheme, if any ray takes a different path from the others, all the possible paths are traversed in turn. The disadvantage of this scheme is that the intermediate node information has to be saved for future use.

In our design, SIMD BSP traversal is implemented in a simple and straightforward way. Whenever there is a ray taking a different path than the others (“*ray incoherence*”), the traversal is stopped and all the objects descended from the current node are tested. Thus, all the RCs process the same data and over the same contexts at the same time. This process is illustrated in Figure 12-4.

The advantages of this scheme is that no intermediate node information needs to be saved, thus simplifying control and reducing memory accesses since fewer address pointers are followed.

In this scheme, each ray may test more objects than necessary. Thus some overhead is introduced. However, simulations show that the amortized overhead is very small when 64 rays are processed in parallel, although each ray itself may take more time to finish than those with 4, 8, 16, or 32 parallel rays. To further remove this overhead, we applied *object test reordering* and

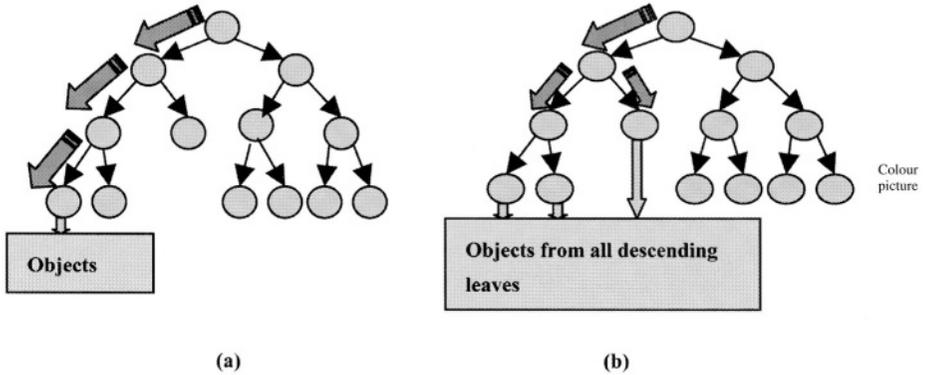


Figure 12-4. Novel BSP incoherence handling. (a) All the 64 rays have ray-coherence. Objects in one leaf are tested. (b) Not all the 64 rays have ray-coherence. All the leaf objects under the node are tested.

*object merging.* We will discuss these optimizations in more details in the next section.

## 5. DATA STRUCTURE AND OPTIMIZATIONS

We have developed a special data structure to support BSP mapping. This structure reduces data and contexts reload, as we describe in this section.

Our data structure is created so that for each node the objects under it are known immediately. The data structure is illustrated in Figure 12-5. In this figure, the item bit-length (e.g., 16 bits) is specified in parenthesis after each item. *Child Address* stands for the address of each child node in FB. The objects descending from this node are grouped by object type. *Sphere*, *cylinder*, *rectangle* and *other objects* stand for vectors where each bit indicates whether or not the object of this type specified by its positions belongs to this node. Figure 12-5 gives one example for sphere case. We order all spheres as *sphere 0*, *sphere 1*, *sphere 2*, and so on. If any of these spheres belongs to the node, the bit indexed by its order is set to ‘1’ in the sphere vector. In this example, sphere 1, 3, 4 belong to the current node. TinyRISC tests these vectors to know which geometrical data to operate on when incoherence happens. All the identified objects under this node are tested. In all the cases, the objects are tested only once. This is called “object merging”, or *mailbox* in [6]. If one object is tested, it is labeled as tested. The calculated value is retained in the FB. If one object to be tested has been labeled, this testing is cancelled, and the value is fetched from FB.

This data structure automatically does “object test reordering”, which tests the same objects without context reload. For example, we check all the spheres before we continue with other object types. Moreover this data structure allows

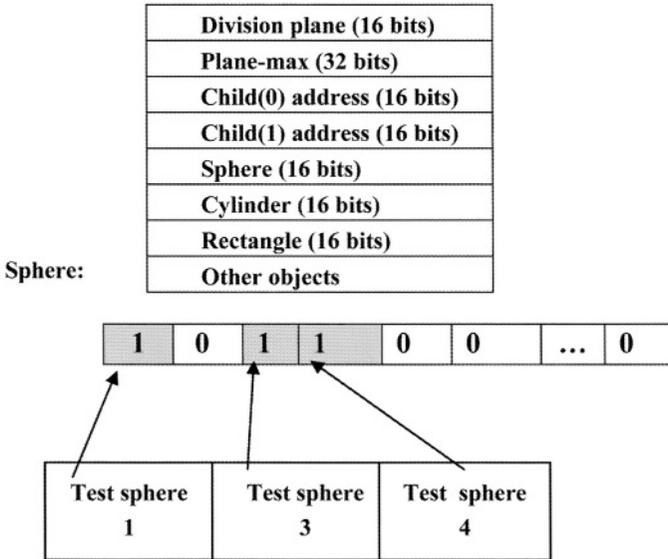


Figure 12-5. Data structure for our BSP algorithm.

the object addresses in the FB to be easily calculated, as is shown in the algorithm in Figure 12-6. It also facilitates pre-fetching since objects are known in advance.

```

r12:= BSP sphere data;
r13:= First sphere address;
r14:= Sphere size;
r0:= 0;
while r0 ≤ number of sphere in the image
do {
    if r12 < 0 then compute_sphere(r13);
    r13 := r14 + r13;
    r12 << 1;
    r0 = r0 + 1;
}

```

Figure 12-6. Calculation of FB address from data structure in each node.

## 6. LOCAL STACK FOR PARALLEL SHADING

After all rays find the closest intersection points and intersected object, the ray-tracing algorithm calculates the color (using *Phong-Shading* model [2, 3]). The shadow and reflection rays are generated and again traverse the BSP tree, as described in Section 3.

However, during this process, the intersection points and intersected objects can be different for different rays. This data cannot be saved to and later fetched from the FB. The reason is that they would have to be fetched one by one for different RCs due to limited bandwidth, which means all but one RC are idle and cycles are wasted. Local RC RAM is used to emulate a stack to address this problem. Besides stack, each RC has one vector to store the current intersection point and intersected object. During ray-object intersection process, when one object is found to be closer to the eye than the one already in the vector, the corresponding data replaces the data in the vector. Otherwise, the vector is kept unchanged. When new recursion starts, the vector is pushed into the stack. When recursion returns, the data is popped from the stack into the vector. In this way, the object data and intersection point required for shading are always available for different rays. The overhead due to these data saving and restoring is very small compared with the whole shading process. This process is illustrated in Figure 12-7.

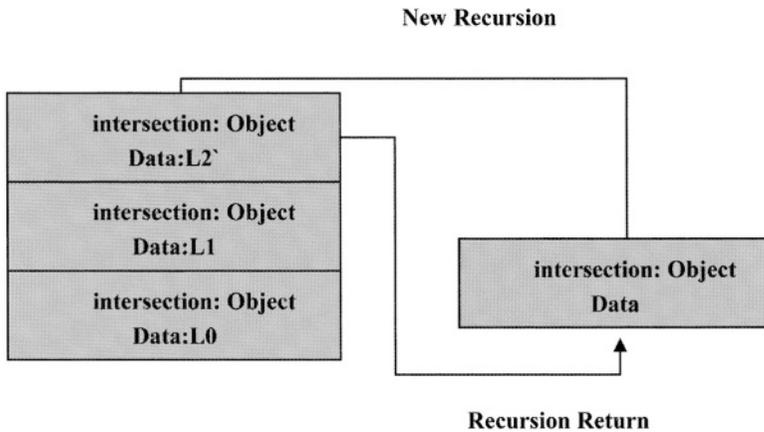


Figure 12-7. Local stack and vector to keep track of the current closest intersection point and the intersect object.

## 7. MEMORY UTILIZATION

SIMD processing of 64 RCs demands high memory bandwidth. For example, up to 64 different data may be concurrently required in MorphoSys. Fortunately, this is not a problem in our design. Our implementation guaran-

tees that all RCs always require the same global data for BSP traversal and intersection, such as BSP tree structure, static object data, etc. Thus, only one copy of this data is needed. The implementation of object merging is also simplified. Since all rays are always required to test the same set of objects, whether they are coherent or not, one copy of the object test history is kept for all RCs.

One parameter that affects all ray tracing mapping schemes is the memory size. For a very large scene, the size of BSP tree structure or object data can be so large that not all of them can fit in the main memory or cache. The result is that processors have to wait for the data to be fetched from the external memory. Also, the context stream may be very large so that not all of them are available for execution. However, this can be easily solved by our double-bank organizations of the FB and Context Memory, as was described in Section 2.

## 8. CENTRALIZED TINYRISC CONTROL

Using our ray tracing mapping scheme, all the rays traverse along the same BSP tree path and find the intersection with the same set of objects. Thus one central controller is enough to broadcast data and contexts to all 64 RCs. TinyRISC in MorphoSys plays this role. The same context is loaded and broadcast to all 64 RCs. Traversal starts from the BSP tree root downward toward the leaves. At each internal node, the status of all rays is sent to TinyRISC. TinyRISC loads the corresponding tree node and also contexts and broadcast them to all 64 RCs. This interaction continues until leaves are reached, where the object data are broadcast. In case that the status information indicates incoherence, TinyRISC loads all the objects data descended from the current node and broadcast them for calculation.

## 9. SIMULATION AND EXPERIMENTAL RESULTS

MorphoSys is designed to be running at 300MHz. It has  $512 \times 16$  internal RC RAM,  $4 \times 16 \text{ K} \times 16 \text{ FB}$ ,  $8 \times 1 \text{ K} \times 32$  Context Memory, and 16 internal registers in each RC. The chip size is expected to be less than  $30 \text{ mm}^2$  using  $0.13 \mu\text{m}$  CMOS technology. Thus MorphoSys is more power efficient than general-purpose processors.

Our targeted applications are those running on portable devices with small images and small number of objects. In our experiments, applications are  $256 \times 256$  in size. The BSP tree is constructed with maximum depth of 15, maximum 5 objects in each leaf. The recursive level is 2.

Different from the other ray tracing mapping schemes [4–6], whose primitives are only triangles, the primitives mapped in our implementation can be any type, such as sphere, cylinder, box, rectangle, triangle, etc. The

advantages are: (1) the object data size is small compared with pure triangle scheme. Much more data is needed to represent scenes using only triangles, and very small triangles are used to get good images. (2) No preprocessing time is needed to transform original models into triangles. However, using pure triangles can simplify BSP traversal since some conditional branches are removed, and also only one ray-object intersection code is needed. Thus, we decided to use a mix of different objects, to attain a better trade-off between algorithm complexity and memory performance.

The algorithm was translated into MorphoSys Assembly and then into machine code. The Simulation is run on MorphoSys processor simulator “Mulate” [16]. We did simulation to get the frame rates for 4, 8, 16, 32, and 64 parallel rays, so that a view of ray tracing performance under different levels of parallel processing is seen. The result is shown in Figure 12-8.

This figure shows that frame rates increase with more paralleled rays. However, the performance of 64-way ray tracing is not twice that of 32-way, but less than that. The reason is that overhead increases as well, although the amortized overhead is actually decreased. This can be formulated and explained as follows. Suppose the processing time for one ray without overhead is  $T$ , total number of rays is  $N$ , and number of rays processed in parallel is  $n$ , and the overhead in processing one ray is  $OV$ , the total processing time is:

$$N \times \frac{T + OV}{n}$$

Thus the frame rate is  $C * n / (N(T + OV))$ , where  $C$  is a constant. If overhead is constant, the frame rate is  $O(n)$ . However,  $OV$  increases as  $n$  increases. Thus frame rate is sub-linear with  $n$ , the number of parallel rays.

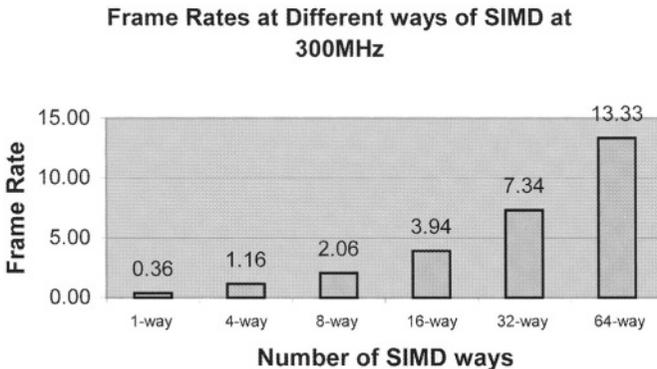


Figure 12-8. Frame rates under different SIMD ways.

## 10. CONCLUSION

This paper gives a complete view of how to utilize the simple SIMD MorphoSys to achieve real-time ray tracing with an efficient use of hardware resources. BSP traversal is mapped in a straightforward way such that no complicated decision and intermediate data saving are necessary. Optimizations, such as object reordering and merging, help simplify the SIMD mapping. The resulted overhead is amortized and is very small when large number of rays are traced in parallel, thus the performance can be very good. Memory is also flexibly utilized. Due to its small size and potential power efficiency, MorphoSys can be used as an economic platform for 3D games on portable devices. Right now, we are further optimizing the architecture so that better hardware supports, such as 32-bit data paths, more registers, etc, are included. Based on what we have achieved (more than 13 frames/second in 300 MHz), it is believed that obtaining real-time ray tracing on portable devices is practical soon.

## ACKNOWLEDGEMENTS

We would like to thank everyone in MorphoSys group, University of California, Irvine, for their suggestions and help in our architectural and hardware modifications. We thank Maria-Cruz Villa-Uriol and Miguel Sainz in Image-Based-Modeling-Rendering Lab for giving sincere help in our geometry modeling and estimation. This work was sponsored by DARPA (DoD) under contract F-33615-97-C-1126 and the National Science Foundation (NSF) under grant CCR-0083080.

## REFERENCES

1. G. Lu, H. Singh, M. H. Lee, N. Bagherzadeh, F. Kurdahi, and E. M. C. Filho. "The MorphoSys Parallel Reconfigurable System." In *Proceedings of Euro-Par*, 1999.
2. A. Glassner. *An Introduction to Ray Tracing*. Academic Press, 1989.
3. S. Parker, W. Martin, P. P. J. Sloan, P. Shirley, B. Smits, and C. Hansen. "Interactive Ray Tracing." In *Proceedings of ACM Symposium on Interactive 3D Graphics*, ACM, 1999.
4. I. Wald, P. Slusallek, C. Benthin, and M. Wagner. "Interactive Rendering with Coherent Ray Tracing." *Computer Graphics Forum*, Vol. 20, pp. 153–164, 2001.
5. N. A. Carr, J. D. Hall, and J. C. Hart. *The Ray Engine*. Tech. Rep. UIUCDCS-R-2002-2269, Department of Computer Science, University of Illinois.
6. T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. "Ray Tracing on Programmable Graphics Hardware." *SIGGraphics 2002 Proc.*, 2002.
7. K. Sung and P. Shirley. "Ray Tracing with the BSP-Tree." *Graphics Gem*, Vol. III, pp. 271–274, Academic Press, 1992.
8. A. Watt. *3D Computer Graphics*, 2nd Edition. Addison-Wesley Press.
9. M. J. Muuss. "Rt and Remrt-Shared Memory Parallel and Network Distributed Ray-Tracing Programs." In *USENIX: Proceedings of the Fourth Computer Graphics Workshop*, October 1987.

10. M. J. Muuss. "Toward Real-Time Ray-Tracing of Combinational Solid Geometric Models." In *Proceedings of BRL-CAD Symposium*, June 1995.
11. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, 2nd Edition. McGraw-Hill and MIT Press, 2001.
12. P. J. Narayanan. "Processor Autonomy on SIMD Architectures." *ICS-7*, pp. 127–136. Tokyo 1993.
13. M. L. Anido, A. Paar, and N. Bagherzadeh. "Improving the Operation Autonomy of SIMD Processing Elements by Using Guarded Instructions and Pseudo Branches." *DSD'2002, Proceedings of EUROMICRO Symposium on Digital System Design*, North Holland, Dortmund, Germany, September, 2002.
14. M. S. Peercy, M. Olano, J. Airey, and P. J. Ungar. "Interactive Multi-Pass Programmable Shading." *ACM SIGGRAPH*, New Orleans, USA, July 2000.
15. L. R. Speer, T. D. DeRose, and B. A. Barsky. "A Theoretical and Empirical Analysis of Coherent Ray Tracing." *Computer-Generated Images (Proceedings of Graphics Interface '85)*, May 1985, 11-25.
16. H. Singh, M. H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. C. Filho. "MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications." *IEEE Transactions on Computers*, Vol. 49, No. 5, pp. 465–481, 2000.

*This page intentionally left blank*

## Chapter 13

# PORTING A NETWORK CRYPTOGRAPHIC SERVICE TO THE RMC2000

## *A Case Study in Embedded Software Development*

Stephen Jan, Paolo de Dios, and Stephen A. Edwards

*Department of Computer Science, Columbia University*

**Abstract.** This chapter describes our experience porting a transport-layer cryptography service to an embedded microcontroller. We describe some key development issues and techniques involved in porting networked software to a connected, limited resource device such as the Rabbit RMC2000 we chose for this case study. We examine the effectiveness of a few proposed porting strategies by examining important program and run-time characteristics.

**Key words:** embedded systems, case study, microcontroller, TCP/IP, network, ethernet, C, assembly, porting

### 1. INTRODUCTION

Embedded systems present a different software engineering problem. These systems are unique in that the hardware and the software are tightly integrated. The limited nature of an embedded system's operating environment requires a different approach to developing and porting software. In this paper, we discuss the key issues in developing and porting a Unix system-level transport-level security (TLS) service to an embedded microcontroller. We discuss our design decisions and experience porting this service using Dynamic C, a C variant, on the RMC2000 microcontroller from Rabbit Semiconductor. The main challenges came when APIs for operating-system services such as networking were either substantially different or simply absent.

Porting software across platforms is such a common and varied software engineering exercise that much commercial and academic research has been dedicated to identifying pitfalls, techniques, and component analogues for it. Porting software has been addressed by high level languages [2, 12], modular programming [11], and component based abstraction, analysis and design techniques [17]. Despite the popularity of these techniques, they are of limited use when dealing with the limited and rather raw resources of a typical embedded system. In fact, these abstraction mechanisms tend to consume more resources, especially memory, making them impractical for microcontrollers. Though some have tried to migrate some of these abstractions to the world

of embedded systems [9], porting applications in a resource-constrained system still requires much reengineering.

This paper presents our experiences porting a small networking service to an embedded microcontroller with an eye toward illustrating what the main problems actually are. Section 2 introduces the network cryptographic service we ported. Section 3 describes some relevant related work, and Section 4 describes the target of our porting efforts, the RMC 2000 development board.

Section 5 describes issues we encountered while porting the cryptographic network service to the development board, Section 6 describes the performance experiments we conducted; we summarize our findings in Section 7.

## 2. NETWORK CRYPTOGRAPHIC SERVICES

For our case study, we ported iSSL, a public-domain implementation of the Secure Sockets Layer (SSL) protocol [6], a Transport-Layer Security (TLS) standard proposed by the IETF [5]. SSL is a protocol that layers on top of TCP/IP to provide secure communications, e.g., to encrypt web pages with sensitive information.

Security, sadly, is not cheap. Establishing and maintaining a secure connection is a computationally-intensive task; negotiating an SSL session can degrade server performance. Goldberg et al. [10] observed SSL reducing throughput by an order of magnitude.

iSSL is a cryptographic library that layers on top of the Unix sockets layer to provide secure point-to-point communications. After a normal unencrypted socket is created, the iSSL API allows a user to bind to the socket and then do secure read/writes on it.

To gain experience using the library, we first implemented a simple Unix service that used the iSSL library to establish a secure redirector. Later, we ported to the RMC2000.

Because SSL forms a layer above TCP, it is easily moved from the server to other hardware. For performance, many commercial systems use coprocessor cards that perform SSL functions. Our case study implements such a service. The iSSL package uses the RSA and AES cipher algorithms and can generate session keys and exchange public keys. Because the RSA algorithm uses a difficult-to-port bignum package, we only ported the AES cipher, which uses the Rijndael algorithm [3]. By default, iSSL supports key lengths of 128, 192, or 256 bits and block lengths of 128, 192, and 256 bits, but to keep our implementation simple, we only implemented 128-bit keys and blocks. During porting, we also referred to the AESCrypt implementation developed by Eric Green and Randy Kaelber.

### **3. RELATED WORK**

Cryptographic services for transport layer security (TLS) have long been available as operating system and application server services [15]. The concept of an embedded TLS service or custom ASIC for stream ciphering are commercially available as SSL/TLS accelerator products from vendors such as Sun Microsystems and Cisco. They operate as black boxes and the development issues to make these services available to embedded devices have been rarely discussed. Though the performance of various cryptographic algorithms such as AES and DES have been examined on many systems [16], including embedded devices [18], a discussion on the challenges of porting complete services to a device have not received such a treatment.

The scope of embedded systems development has been covered in a number of books and articles [7, 8]. Optimization techniques at the hardware design level and at the pre-processor and compiler level are well-researched and benchmarked topics [8, 14, 19]. Guidelines for optimizing and improving the style and robustness of embedded programs have been proposed for specific languages such as ANSI C [1], Design patterns have also been proposed to increase portability and leverage reuse among device configurations for embedded software [4]. Overall, we found the issues involved in porting software to the embedded world have not been written about extensively, and are largely considered “just engineering” doomed to be periodically reinvented. Our hope is that this paper will help engineers be more prepared in the future.

### **4. THE RMC2000 ENVIRONMENT**

Typical for a small embedded system, the RMC2000 TCP/IP Development Kit includes 512 K of flash RAM, 128 k SRAM, and runs a 30 MHz, 8-bit Z80-based microcontroller (a Rabbit 2000). While the Rabbit 2000, like the Z80, manipulates 16-bit addresses, it can access up to 1 MB through bank switching.

The kit also includes a 10 Base-T network interface and comes with software implementing TCP/IP, UDP and ICMP. The development environment includes compilers and diagnostic tools, and the board has a 10-pin programming port to interface with the development environment.

#### **4.1. Dynamic C**

The Dynamic C language, developed along with the Rabbit microcontrollers, is an ANSI C variant with extensions that support the Rabbit 2000 in embedded system applications. For example, the language supports cooperative and preemptive multitasking, battery-backed variables, and atomicity guarantees for shared multibyte variables. Unlike ANSI C, local variables in

Dynamic C are `static` by default. This can dramatically change program behavior, although it can be overridden by a directive. Dynamic C does not support the `#include` directive, using instead `#use`, which gathers precompiled function prototypes from libraries. Deciding which `#use` directives should replace the many `#include` directives in the source files took some effort.

Dynamic C omits and modifies some ANSI C behavior. Bit fields and enumerated types are not supported. There are also minor differences in the `extern` and `register` keywords. As mentioned earlier, the default storage class for variables is `static`, not `auto`, which can dramatically change the behavior of recursively-called functions. Variables initialized in a declaration are stored in flash memory and cannot be changed.

Dynamic C's support for inline assembly is more comprehensive than most C implementations, and it can also integrate C into assembly code, as in the following:

```
#asm nodebug
InitValues::
    ld hl,0xa0;
c   start_time = 0;    // Inline C
c   counter = 256;    // Inline C
    ret
#endasm
```

We used the inline assembly feature in the error handling routines that caught exceptions thrown by the hardware or libraries, such as divide-by-zero. We could not rely on an operating system to handle these errors, so instead we specified an error handler using the `defineErrorHandler(void*errfcn)` system call. Whenever the system encounters an error, the hardware passes information about the source and type of error on the stack and calls this user-defined error handler. In our implementation, we used (simple) inline assembly statements to retrieve this information. Because our application was not designed for high reliability, we simply ignored most errors.

## 4.2. Multitasking in Dynamic C

Dynamic C provides both cooperative multitasking, through `costatements` and `cofunctions`, and preemptive multitasking through either the `slice` statement or a port of Labrosse's  $\mu$ C/OS-II real-time operating system [13]. Dynamic C's `costatements` provide multiple threads of control through independent program counters that may be switched among explicitly, such as in this example:

```
for (;;) {
    costate {
        waitfor( tcp_packet_port_21() );
        // handle FTP connection
        yield; // Force context switch
    }
    costate {
        waitfor(tcp_packet_port_23() );
        // handle telnet connection
    }
}
```

The `yield` statement immediately passes control to another costatement. When control returns to the costatement that has yielded, it resumes at the statement following the `yield`. The statement `waitfor(expr)`, which provides a convenient mechanism for waiting for a condition to hold, is equivalent to `while (!expr) yield;`

Cofunctions are similar, but also take arguments and may return a result. In our port, we used costatements to handle multiple connections with multiple processes. We did not use  $\mu\text{C}/\text{OS-II}$ .

### 4.3. Storage class specifiers

To avoid certain race conditions, Dynamic C generates code that disables interrupts while multibyte variables marked `shared` are being changed, guaranteeing atomic updates.

For variables marked `protected`, Dynamic C generates extra code that copies their value to battery-backed RAM before every modification. Backup values are copied to main memory when when system is restarted or when `_sysIsSoftReset()` is called. We did not need this feature in this port.

The Rabbit 2000 microcontroller has a 64 K address space but uses bank-switching to access 1 M of total memory. The lower 50 K is fixed, root memory, the middle 6 K is I/O, and the top 8 K is bank-switched access to the remaining memory. A user can explicitly request a function to be located in either root or extended memory using the storage class specifiers `root` and `xmem` (Figure 13-1). We explicitly located certain functions, such as the error handler, in root memory, but we let the compiler locate the others.

### 4.4. Function chaining

Dynamic C provides function chaining, which allows segments of code to be embedded within one or more functions. Invoking a named function chain causes all the segments belonging to that chain to execute. Such chains enable initialization, data recovery, or other kinds of tasks on request. Our port did not use this feature.

```

// Interrupts disabled during changes to a, b, and c
// Updates guaranteed atomic
shared float a, b, c;

main() {
    protected int state1; // Battery-backed
    ...
    // restore protected variable
    _sysIfSoftReset()
}

// Place func1 in root memory
root int func1() { ... }

// Place following assembly code in root memory
#memmap root
#asm root
...
#endasm

// Place func2 in extended memory
xmem int func2() { ... }

```

Figure 13-1. Fragment illustrating various Dynamic-C-specific storage class specifiers.

```

// Create a chain named "recover" and add three functions
#makechain recover
#funcchain recover free_memory
#funcchain recover declare_memory
#funcchain recover initialize

// Invoke all three functions in the chain in some sequence
recover();

```

## 5. PORTING AND DEVELOPMENT ISSUES

A program rarely runs unchanged on a dramatically different platform; something always has to change. The fundamental question is, then, how much must be changed or rewritten, and how difficult these rewrites will be.

We encountered three broad classes of porting problems that demanded code rewrites. The first, and most common, was the absence of certain libraries and operating system facilities. This ranged from fairly simple (e.g., Dynamic C does not provide the standard `random` function), to fairly difficult (e.g., the

protocols include timeouts, but Dynamic C does not have a timer), to virtually impossible (e.g., the iSSL library makes some use of a filesystem, something not provided by the RMC2000 environment). Our solutions to these ranged from creating a new implementation of the library function (e.g., writing a `random` function) to working around the problem (e.g., changing the program logic so it no longer read a hash value from a file) to abandoning functionality altogether (e.g., our final port did not implement the RSA cipher because it relied on a fairly complex bignum library that we considered too complicated to rework).

A second class of problem stemmed from differing APIs with similar functionality. For example, the protocol for accessing the RMC2000's TCP/IP stack differs quite a bit from the BSD sockets used within iSSL. Figure 13-2 illustrates some of these differences. While solving such problems is generally much easier than, say, porting a whole library, reworking the code is tedious.

A third class of problem required the most thought. Often, fundamental assumptions made in code designed to run on workstations or servers, such as the existence of a filesystem with nearly unlimited capacity (e.g., for keeping a log), are impractical in an embedded systems. Logging and somewhat sloppy memory management that assumes the program will be restarted occasionally to cure memory leaks are examples of this. The solutions to such problems are either to remove the offending functionality at the expense of features (e.g., remove logging altogether), or a serious reworking of the code (e.g., to make logging write to a circular buffer rather than a file).

## 5.1. Interrupts

We used the serial port on the RMC2000 board for debugging. We configured the serial interface to interrupt the processor when a character arrived. In response, the system either replied with a status messages or reset the application, possibly maintaining program state. A Unix environment provides a high-level mechanism for handling software interrupts:

```
main() {
    signal(SIGINT, sigproc); // Register signal handler
}
void sigproc() { /* Handle the signal */ }
```

In Dynamic C, we had to handle the details ourselves. For example, to set up the interrupt from the serial port, we had to enable interrupts from the serial port, register the interrupt routine, and enable the interrupt receiver.

```

int echo_server() {
    int sock, newsock, len;
    struct sockaddr_in addr;
    char buf[LEN];

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        return -1;

    memset(&addr, 0, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = htonl(INADDR_ANY);
    addr.sin_port = htons(MYPORT);
    if ( bind(sock, (struct sockaddr *) &addr,
              sizeof(struct sockaddr_in)) < 0 ) return -1;
    if ( listen(sock, LISTENQ) < 0 ) return -1;
    for (;;) {
        if ((newsock = accept(sock, NULL, NULL) ) < 0 )
            return -1;
        if ((len = recv(newsock, buf, LEN, 0)) < 0)
            return -1;
        if (send(newsock, buf, len, 0) < 0) return -1;
        close(conn_s);
    }
}

```

(a)

```

int echo_server()
{
    tcp_socket sock;
    int status;
    char buf[LEN];

    sock_init();
    for (;;) {
        tcp_listen(&sock, PORT, 0, 0, NULL, 0);
        sock_wait_established(&sock, 0, NULL, &status);
        sock_mode(&sock, TCP_MODE_ASCII);
        while (tcp_tick(&sock)) {
            sock_wait_input(&sock, 0, NULL, &status);
            if (sock_gets(&sock, buf, LEN))
                sock_puts(&sock, buf);
        }
    }
}

```

(b)

Figure 13-2. A comparison of (a) traditional BSD sockets-based code and (b) equivalent code in the Dynamic C environment illustrating the significant differences in API.

```
main() {
    // Set serial port A as input interrupt
    WrPortI(SADR, &SADRShadow, 0x00);
    // Register interrupt service routine
    SetVectExtern2000(1, my_isr);
    // Enable external INTO on SA4, rising edge
    WrPortI(IOCR, NULL, 0x2B);
    ...
    // Disable interrupt 0
    WrPortI(IOCR, NULL, 0x00);
}

nodebug root interrup void my_isr() { ... }
```

We could have avoided interrupts had we used another network connection for debugging, but this would have made it impossible to debug a system having network communication problems.

## 5.2. Memory

A significant difference between general platform development and embedded system development is memory. Most embedded devices have little memory compared to a typical modern workstation. Expecting to run into memory issues, we used a well-defined taxonomy [20] to plan out memory requirements. This proved unnecessary, however, because our application had very modest memory requirements.

Dynamic C does not support the standard library functions `malloc` and `free`. Instead, it provides the `xalloc` function that allocates extended memory only (arithmetic, therefore, cannot be performed on the returned pointer). More seriously, there is no analogue to `free`; allocated memory cannot be returned to a pool.

Instead of implementing our own memory management system (which would have been awkward given the Rabbit's bank-switched memory map), we chose to remove all references to `malloc` and statically allocate all variables. This prompted us to drop support of multiple key and block sizes in the iSSL library.

## 5.3. Program structure

As we often found during the porting process, the original implementation made use of high-level operating system functions such as `fork` that were not provided by the RMC2000 environment. This forced us to restructure the program significantly.

The original TLS implementation handles an arbitrary number of connections using the typical BSD sockets approach shown below. It first calls

`listen` to begin listening for incoming connections, then calls `accept` to wait for a new incoming connection. Each request returns a new file descriptor passed to a newly forked process that handles the request. Meanwhile, the main loop immediately calls `accept` to get the next request.

```
listen(listen_fd)
for (;;) {
    accept_fd = accept(listen_fd);
    if ((childpid = fork()) == 0) {
        // process request on accept_fd
        exit(0); // terminate process
    }
}
```

The Dynamic C environment provides neither the standard Unix `fork` nor an equivalent of `accept`. In the RMC 2000's TCP implementation, the socket bound to the port also handles the request, so each connection is required to have a corresponding call to `tcp_listen`. Furthermore, Dynamic C effectively limits the number of simultaneous connections by limiting the number of `costatements`.

```
for (;;) {
    costate {
        tcp_listen(socket1,TLS_PORT, ...);
        while (sock_established(socket1) == 0) yield;
        // handle request
    }
    costate {
        tcp_listen(socket2,TLS_PORT, ...);
        while((0 == sock_established(socket2))) yield;
        // handle request
    }
    costate {
        tcp_listen(socket2,TLS_PORT, ...);
        while((0 == sock_established(socket2))) yield;
        // handle request
    }
    costate {
        // drive TCP stack
        tcp_tick(NULL);
    }
}
```

*Figure 13-3.* The structure of the main loop of the TLS server, which can handle a most three requests because it is limited to four processes.

Thus, to handle multiple connections and processes, we split the application into four processes: three processes to handle requests (allowing a maximum of three connections), and one to drive the TCP stack (Figure 13-3). We could easily increase the number of processes (and hence simultaneous connections) by adding more costatements, but the program would have to be re-compiled.

## **6. EXPERIMENTAL RESULTS**

To gauge which optimization techniques were worthwhile, we compared the C implementation of the AES algorithm (Rijndael) included with the iSSL library with a hand-coded assembly version supplied by Rabbit Semiconductor. A testbench that pumped keys through the two implementations of the AES cipher showed the assembly implementation ran faster than the C port by a factor of 15-20.

We tried a variety of optimizations on the C code, including moving data to root memory, unrolling loops, disabling debugging, and enabling compiler optimization, but this only improved run time by perhaps 20%. Code size appeared uncorrelated to execution speed. The assembly implementation was 9% smaller than the C, but ran more than an order of magnitude faster. Debugging and testing consumed the majority of the development time. Many of these problems came from our lack of experience with Dynamic C and the RMC2000 platform, but unexpected, undocumented, or simply contradictory behavior of the hardware or software and its specifications also presented challenges.

## **7. CONCLUSIONS**

We described our experiences porting a library and server for transport-level security protocol – iSSL – onto a small embedded development board: the RMC 2000, based on the Z80-inspired Rabbit 2000 microcontroller. While the Dynamic C development environment supplied with the board gave useful, necessary support for some hardware idiosyncrasies (e.g., its bank-switched memory architecture) its concurrent programming model (cooperative multi-tasking with language-level support for costatements and cofunctions) and its API for TCP/IP both differed substantially from the Unix-like behavior the service originally used, making porting difficult. Different or missing APIs proved to be the biggest challenge, such as the substantial difference between BSD-like sockets and the provided TCP/IP implementation or the simple absence of a filesystem. Our solutions to these problems involved either writing substantial amounts of additional code to implement the missing library functions or reworking the original code to use or simply avoid the API.

We compared the speed of our direct port of a C implementation of the RSA (Rijndael) cipher with a hand-optimized assembly version and found a disturbing factor of 15 20 in performance in favor of the assembly. From all of this, we conclude that there must be a better way. Understanding and dealing with differences in operating environment (effectively, the API) is a tedious, error-prone task that should be automated, yet we know of no work beyond high-level language compilers that confront this problem directly.

## REFERENCES

1. M. Barr. *Programming Embedded Systems in C and C++*. O Reilly & Associates, Inc., Sebastopol, California, 1999.
2. P. J. Brown. "Levels of Language for Portable Software." *Communications of the ACM*, Vol. 15, No. 12, pp. 1059–1062, December 1972.
3. J. Daemen and V. Rijmen. "The Block Cipher Rijndael." In *Proceedings of the Third Smart Card Research and Advanced Applications Conference*, 1998.
4. M. de Champlain. "Patterns to Ease the Port of Micro-Kernels in Embedded Systems." In *Proceedings of the 3rd Annual Conference on Pattern Languages of Programs (PLoP 96)*, Allerton Park, Illinois, June 1996.
5. T. Dierks and C. Allen. *The TLS Protocol*. Internet draft, Transport Layer Security Working Group, May 1997.
6. A. O. Freier, P. Karlton, and P. C. Kocher. *The SSL Protocol*. Internet draft, Transport Layer Security Working Group, Nov. 1996.
7. J. Gassle. "Dumb Mistakes." *The Embedded Muse Newsletter*, August 7, 1997.
8. J. G. Gassle. *The Art of Programming Embedded Systems*. Academic Press, 1992.
9. A. Gokhale and D. C. Schmidt. "Techniques for Optimizing CORBA Middleware for Distributed Embedded Systems." In *Proceedings of INFOCOM 99*, March 1999.
10. A. Goldberg, R. Buff, and A. Schmitt. "Secure Web Server Performance Using SSL Session Keys." In *Workshop on Internet Server Performance, held in conjunction with SIGMETRICS 98*, June 1998.
11. D. R. Hanson. *C Interfaces and Implementations-Techniques for Creating Reusable Software*. Addison-Wesley, Reading, Massachusetts, 1997.
12. B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, New Jersey, second edition, 1988.
13. J. Labrosse. *MicroC/OS-II*. CMP Books, Lawrence, Kansas, 1998.
14. R. Leupers. *Code Optimization Techniques for Embedded Processors: Methods, Algorithms, and Tools*. Kluwer Academic Publishers, 2000.
15. mod ssl. Documentation at <http://www.modssl.org>, 2000. Better-documented derivative of the Apache SSL secure web server.
16. B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson. "Performance Comparison of the AES Submissions." In *Proceedings of the Second AES Candidate Conference*, pp. 15–34, NIST, March 1999.
17. S. Vinoski. "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments." *IEEE Communications Magazine*, Vol. 14, No. 2, February 1997.
18. C. Yang. "Performance Evaluation of AES/DES/Camellia on the 6805 and H8/300 CPUs." In *Proceedings of the 2001 Symposium on Cryptography and Information Security*, pp. 727–730, Oiso, Japan, January 2001.
19. V. Zivojnovic, C. Schlager, and H. Meyr. "DSPStone: A DSP-oriented Benchmarking Methodology." In *International Conference on Signal Processing*, 1995.
20. K. Zurell. *C Programming for Embedded Systems*. CMP Books, 2000.

PART IV:

EMBEDDED OPERATING SYSTEMS FOR SOC

*This page intentionally left blank*

## Chapter 14

# INTRODUCTION TO HARDWARE ABSTRACTION LAYERS FOR SOC

Sungjoo Yoo and Ahmed A. Jerraya

*TIMA Laboratory, Grenoble, France*

**Abstract.** In this paper, we explain hardware abstraction layer (HAL) and related issues in the context of SoC design. First, we give a HAL definition and examples of HAL function. HAL gives an abstraction of HW architecture to upper layer software (SW). It hides the implementation details of HW architecture, such as processor, memory management unit (MMU), cache, memory, DMA controller, timer, interrupt controller, bus/bus bridge/network interface, I/O devices, etc. HAL has been used in the conventional area of operating system to ease porting OSs on different boards. In the context of SoC design, HAL keeps still the original role of enabling the portability of upper layer SW. However, in SoC design, the portability impacts on the design productivity in two ways: SW reuse and concurrent HW and SW design. As in the case of HW interface standards, e.g. VCI, OCP-IP, etc., the HAL API needs also a standard. However, contrary to the case of HW interface, the standard of HAL API needs to be generic not only to support the common functionality of HAL, but also to support new HW architectures in application-specific SoC design with a guideline for HAL API extension. We present also three important issues of HAL for SoC design: HAL modelling, application-specific and automatic HAL design.<sup>1</sup>

**Key words:** SoC, hardware abstraction layer, hardware dependent software, software reuse, HAL standard, simulation model of HAL, automatic and application-specific design of HAL

## 1. INTRODUCTION

Standard on-chip bus interfaces have been developed to enable hardware (HW) component reuse and integration [1, 2]. Recently, VSIA is investigating the same analogy in software (SW) component reuse and integration with its hardware dependent software (HdS) application programming interface (API). It is one that is conventionally considered as hardware abstraction layer (HAL) or board support package (BSP).

In this paper, we investigate the following questions related to HAL, especially, for system-on-chip (SoC).

- (1) What is HAL?
- (2) What is the role of HAL for SoC design?
- (3) What does the HAL standard for SoC design need to look like?
- (4) What are the issues of HAL for SoC design?

## 2. WHAT IS HAL?

In this paper, we define HAL as *all the software that is directly dependent on the underlying HW*. The examples of HAL include boot code, context switch code, codes for configuration and access to HW resources, e.g. MMU, on-chip bus, bus bridge, timer, etc. In real HAL usage, a practical definition of HAL can be done by the designer (for his/her HW architecture), by OS vendors, or by a standardization organization like VSIA.

As Figure 14-1 shows, a HAL API gives to the upper layer SW, i.e. operating system (OS) and application SW an abstraction of underlying HW architecture, i.e. processor local architecture. Figure 14-2 shows an example of processor local architecture. It contains processor, memory management unit (MMU), cache, memory, DMA controller, timer, interrupt controller, bus/bus bridge/network interface, I/O devices, etc.

To be more specific, as the abstraction of processor, HAL gives an abstraction of:

- data types, in the form of data structure, e.g. bit sizes of boolean, integer, float, double, etc.
- boot code
- context, in the form of data structure
  - register save format (e.g. R0–R14 in ARM7 processor)
- context switch functions, e.g. context\_switch, setjmp/longjmp
- processor mode change
- enable kernel/user\_mode
  - (un)mask processor interrupt
  - interrupt\_enable/disable
  - get\_interrupt\_enabled
- etc.

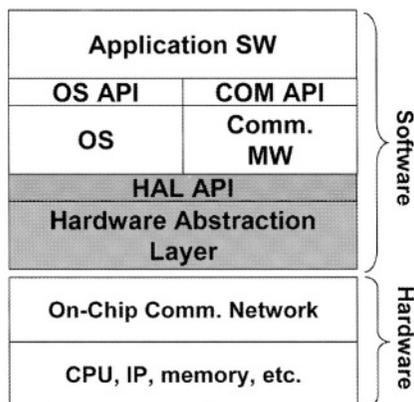


Figure 14-1. Hardware abstraction layer.

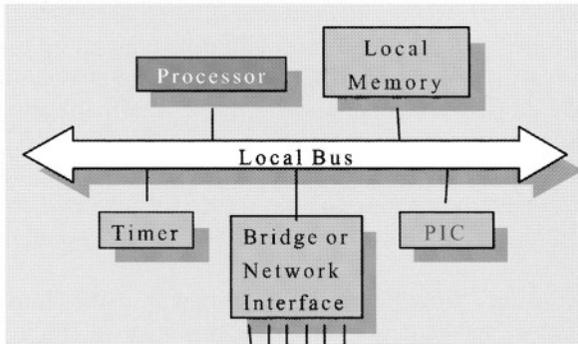


Figure 14-2. A processor local architecture.

Figure 14-3 shows an example of HAL API function for context switch, `_cxt_switch(cxt_type oldcxt, cxt_type newcxt)`. The function can be used for context switch on any processors. For a specific processor, we need to implement the body of the function. Figure 14-3 shows also the real code of the HAL API function for ARM7 processor.

Since the HAL API gives an abstraction of underlying HW architecture, when the OS or application SW is designed using the HAL API, the SW code is portable as far as the HAL API can be implemented on the underlying HW architecture. Thus, conventionally, the HAL has been used to ease OS porting on new HW architectures.

```
// HAL API function for context switch
typedef int cxt_type[15];
void __cxt_switch(cxt_type oldcxt, cxt_type newcxt);
```

```
// ARM7-specific implementation of __cxt_switch
__cxt_switch ;r0, old stack pointer, r1, new stack pointer
STMIA r0!,{r0-r14} ; save the registers of current task
LDMIA r1!,{r0-r14} ; restore the registers of new task
SUB pc,lr,#0 ; return
END
```

Figure 14-3. An example of HAL API function.

### 3. RELATED WORK

There are similar concepts to HAL: nano-kernel and device driver. Nano-kernel is usually defined to be an ensemble of interrupt service routines and task stacks [4]. It serves as a foundation where a micro-kernel can be built. In this definition, nano-kernel can be considered to be a part of HAL since it

does not handle I/O. However, nano-kernel is often used exactly to represent HAL. In the case of  $\mu$ Choice OS, nano-kernel is equivalent to HAL [5].

A device driver gives an abstraction of I/O device. Compared to HAL, it is limited to I/O, not covers context switch, interrupt management, etc. To be more exact, the entire device driver does not belong to HAL. In the case of device driver, to identify the portion that depends on the underlying HW architecture, we need to separate the device driver into two parts: HW independent and HW dependent parts [6]. Then, the HW dependent part can belong to HAL.

Though HAL is an abstraction of HW architecture, since it has been mostly used by OS vendors and each OS vendor defines its own HAL, most of HALs are also OS dependent. In the case of OS dependent HAL, it is often called board support package (BSP).

Window CE provides for BSPs for many standard development boards (SDBs) [3]. The BSP consists of boot loader, OEM abstraction layer (OAL), device drivers, and configuration files. To enable to meet the designer's HW architecture, the BSP can be configured with a configuration tool called Platform Builder. The device drivers are provided in a library which has three types of functions: chip support package (CSP) drivers, BSP or platform-specific drivers, and other common peripheral drivers. In other commercial OSs, we can find similar configuration tools and device driver libraries.

## **4. HAL FOR SOC DESIGN**

### **4.1. HAL usage for SW reuse and concurrent HW and SW design**

In the context of SoC design, HAL keeps still the original role of enabling the portability of upper layer SW. However, in SoC design, the portability impacts on the design productivity in two ways: SW reuse and concurrent HW and SW design.

Portability enables to port the SW on different HW architectures. In terms of design reuse, the portability enables to reuse the SW from one SoC design to another. Easy porting of OS and application SW means easy reuse of OS and application SW over different HW architectures that support the same HAL API. Thus, it can reduce the design efforts otherwise necessary to adapt the SW on the new HW architecture.

In many SoC designs, complete SW reuse may be infeasible (e.g. due to a new functionality or performance optimisation). Thus, in such cases, both SW and HW need to be designed. The conventional design flow is that the HW architecture is designed first, then the SW design is performed based on the designed HW architecture. In terms of design cycle, this practice takes a long design cycle since SW and HW design steps are sequential.

HAL serves to enable the SW design early before finishing the HW architecture design. After fixing a HAL API, we can perform SW and HW design

concurrently considering that it is a contract between SW and HW designers. The SW is designed using the HAL API without considering the details of HAL API implementation. Since the SW design can start as soon as the HAL API is defined, SW and HW design can be performed concurrently thereby reducing the design cycle.

To understand better SW reuse by HAL usage, we will explain the correspondence between SW and HW reuse. Figure 14-4 exemplifies HW reuse. As shown in the figure, to reuse a HW IP that has been used before in a SoC design, we make it ready for reuse by designing its interface conforming to a HW interface standard (in Figure 14-4(a), i.e. on-chip bus interface standards (e.g. VCI, OCP-IP, AMBA, STBus, CoreConnect, etc.). Then, we reuse it in a new SoC design (Figure 14-4(b)).

Figure 14-5 shows the case of SW reuse. Figure 14-5(a) shows that a SW IP, MPEG4 code, is designed for a SoC design #1. The SoC design uses a HAL API (shaded rectangle in the figure). The SW IP is designed using the

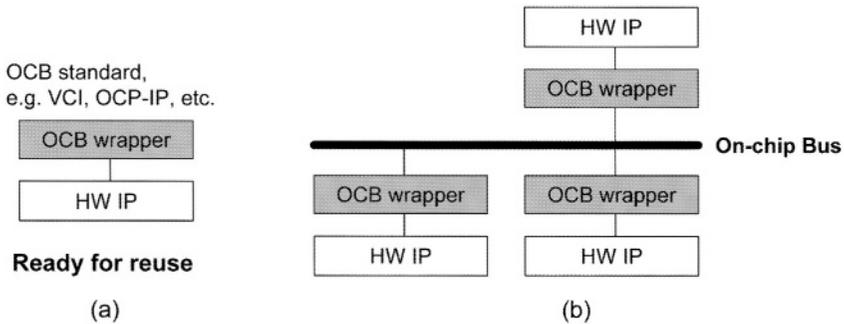


Figure 14-4. HW reuse using a HW interface standard.

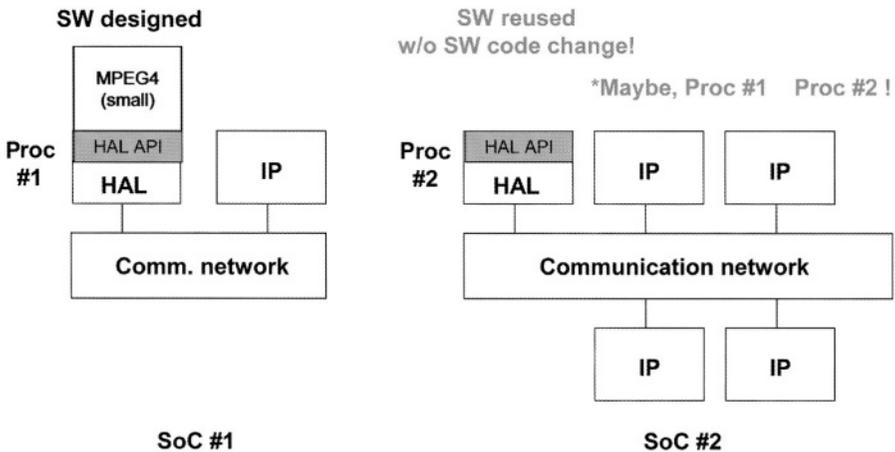


Figure 14-5. SW reuse using a HAL API as a SW interface standard.

HAL API on a processor, Proc #1. When we design a new SoC #2 shown in Figure 14-5(b), if the new design uses the same HAL API, then, the same SW IP can be reused without changing the code. In this case, the target processor on the new SoC design (Proc #2) may be different from the one (Proc #1) for which the SW IP was developed initially.

## 4.2 HAL standard for SoC design

As in the case of HW interface standard, HAL needs also standards to enable easy SW reuse across industries as well as across in-house design groups. In VSIA, a development working group (DWG) for HdS (hardware dependent software) works to establish a standard of HAL API since September 2001 [1].

When we imagine a standard of HAL API, we may have a question: how can such a HAL standard support various and new hardware architectures? Compared with conventional board designs, one of characteristics in SoC design is application-specific HW architecture design. To design an optimal HW architecture, the designer can invent any new HW architectures that an existing, maybe, fixed HAL may not support. To support such new architectures, a fixed standard of HAL API will not be sufficient. Instead, the standard HAL API needs to be a generic HAL API that consists of common HAL API functions and a design guideline to extend HAL API functions according to new HW architectures. It will be also possible to prepare a set of common HAL APIs suited to several application domains (e.g. a HAL API for multimedia applications) together with the design guideline.

The guideline to develop extensible HAL API functions needs to be a component-based construction of HAL, e.g. as in [5]. In this case, HAL consists of components, i.e. basic functional components. Components communicate via clear interface, e.g. C++ abstract class for the interface. To implement the standard of HAL API, their internal implementations may depend on HW architectures.

## 5. HAL ISSUES

In this section, we present the following three issues related with HAL for SoC design: HAL modelling, application-specific and automatic HAL design.

### 5.1. HAL modelling

When using a HAL API in SW reuse or in concurrent HW and SW design, the first problem that the designer encounters is how to validate the upper layer SW with the HW architecture that may not be designed yet. Figure 14-6 shows a case where the designer wants to reuse the upper layer SW including application SW, OS and a communication middleware, e.g. message passing interface library. At the first step of SoC design, the designer needs

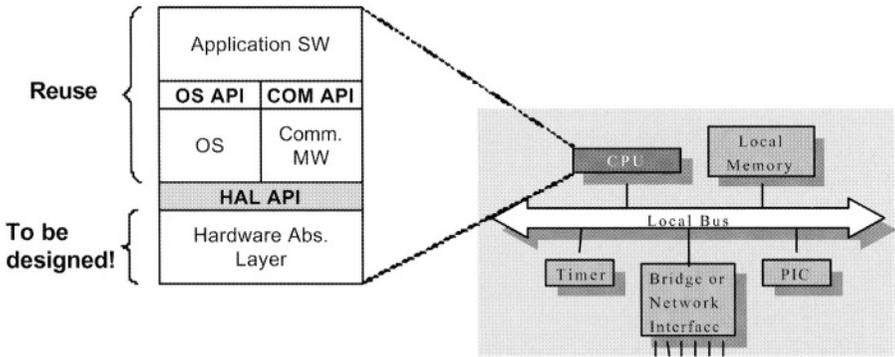


Figure 14-6. HAL modelling.

to validate the correct operation of reused SW in the HW architecture. However, since the design of HW architecture may not yet be finished, the HAL may not be designed, either.

In such a case, to enable the validation via simulation, we need a simulation model of HAL. The simulation model needs to simulate the functionality of HAL, i.e. context switch, interrupt processing, and processor I/O. The simulation model of HAL can be used together with transaction level or RTL models of on-chip bus and other HW modules. The simulation model needs also to support timing simulation of upper layer SW. More details of HAL simulation model can be found in [7].

## 5.2. Application-specific and automatic HAL design

When a (standard) HAL API is generic (to support most of HW architectures) or platform-specific, it will provide for good portability to upper layer SW. However, we can have a heavy HAL implementation (up to ~10 k lines of code in the code library [3]). In such a case, the HAL implementation may cause an overhead of system resource (in terms of code size) and performance (execution delay of HAL).

To reduce such an overhead of HAL implementation, application-specific HAL design is needed. In this case, HAL needs to be tailored to the upper layer SW and HW architecture. To design such a HAL, we need to be able to implement only the HAL API functions (and their dependent functions) used by the upper layer SW. To the best of our knowledge, there has been no research work to enable such an application-specific HAL design.

Conventional HAL design is manually done for a give board, e.g. using a configuration tool such as Platform Builder for WindowCE. In the case of SoC design, we perform design space exploration of HW architectures to obtain optimal HW architecture(s). For each of HW architecture candidates, HAL needs to be designed. Due to the large number of HW architecture candidates, manual HAL design will be too time-consuming to enable fast

evaluation of HW architecture candidates. Thus, in this case, automatic HAL design is needed to reduce the HAL design efforts. In terms of automatic design of device drivers, there have been presented some work [6]. However, general solutions to automatically generate the entire HAL need to be developed. The general solutions need also to support both application-specific and automatic design of HAL.

## 6. CONCLUSION

In this paper, we presented HAL definition, examples of HAL function, the role of HAL in SoC design, and related issues. HAL gives an abstraction of HW architecture to upper layer SW. In SoC design, the usage of HAL enables SW reuse and concurrent HW and SW design. The SW code designed using a HAL API can be reused, without code change, on different HW architectures that support the same HAL API. The HAL API works also as a contract between SW and HW designers to enable them to work concurrently without bothering themselves with the implementation details of the other parts (HW or SW).

As in the case of HW interface standards, the HAL API needs also a standard. However, contrary to the case of HW interface standards, the standard of HAL API needs to support new HW architectures in application-specific SoC design as well as the common functionality of HAL. Thus, the standard needs to have a common HAL API(s) and a guideline to expand the generic one to support new HW architectures. To facilitate the early validation of reused SW, we need also the simulation model of HAL. To reduce the overhead of implementing HAL (i.e. design cycle) and that of HAL implementation itself (i.e. code size, execution time overhead), we need methods of automatic and application-specific HAL design.

## REFERENCES

1. Virtual Socket Interface Alliance, <http://www.vsi.org/>
2. Open Core Protocol, <http://www.ocpip.org/home>
3. Windows CE, <http://www.microsoft.com/windows/embedded/>
4. D. Probert, et al. "SPACE: A New Approach to Operating System Abstraction." *Proceedings of International Workshop on Object Orientation in Operating Systems*, pp. 133–137, October 1991.
5. S. M. Tan, D. K. Raila, and R. H. Campbell. "An Object-Oriented Nano-Kernel for Operating System Hardware Support." In *Fourth International Workshop on Object-Oriented in Operating Systems*, Lund, Sweden, August 1995.
6. S. Wang, S. Malik, and R. A. Bergamaschi, "Modeling and Integration of Peripheral Devices in Embedded Systems." *Proceedings of DATE (Design, Automation, and Test in Europe)*, March 2003.
7. S. Yoo, A. Bouchhima, I. Bacivarov, and A. A. Jerraya. "Building Fast and Accurate SW Simulation Models based on SoC Hardware Abstraction Layer and Simulation Environment Abstraction Layer." *Proceedings of DATE*, March 2003.

## Chapter 15

# HARDWARE/SOFTWARE PARTITIONING OF OPERATING SYSTEMS

## *The $\delta$ Hardware/Software RTOS Generation Framework for SoC*

Vincent J. Mooney III

*School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA, USA*

**Abstract.** We present a few specific hardware/software partitions for real-time operating systems and a framework able to automatically generate a large variety of such partitioned RTOSes. Starting from the traditional view of an operating system, we explore novel ways to partition the OS functionality between hardware and software. We show how such partitioning can result in large performance gains in specific cases involving multiprocessor System-on-a-Chip scenarios.

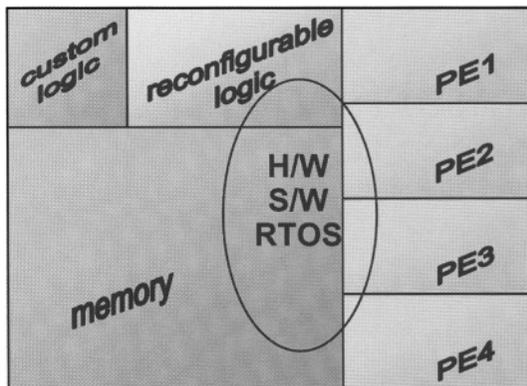
**Key words:** hardware/software partitioning, operating systems

### 1. INTRODUCTION

Traditionally, an Operating System (OS) implements in software basic system functions such as task/process management and I/O. Furthermore, a Real-Time Operating Systems (RTOS) traditionally implements in software functionality to manage tasks in a predictable, real-time manner. However, with System-on-a-Chip (SoC) architectures similar to Figure 15-1 becoming more and more common, OS and RTOS functionality need not be implemented solely in software. Thus, partitioning the interface between hardware and software for an OS is a new idea that can have a significant impact.

We present the  $\delta$  hardware/software RTOS generation framework for System-on-a-Chip (SoC). We claim that current SoC designs tend to ignore the RTOS until late in the SoC design phase. In contrast, we propose RTOS/SoC codesign where both the multiprocessor SoC architecture and a custom RTOS (with part potentially in hardware) are designed together.

In short, this paper introduces a hardware/software RTOS generation framework for customized design of an RTOS within specific predefined RTOS services and capabilities available in software and/or hardware (depending on the service or capability).



Colour picture

Figure 15-1. Target SoC architecture (PE: Processing element).

## 2. RELATED WORK

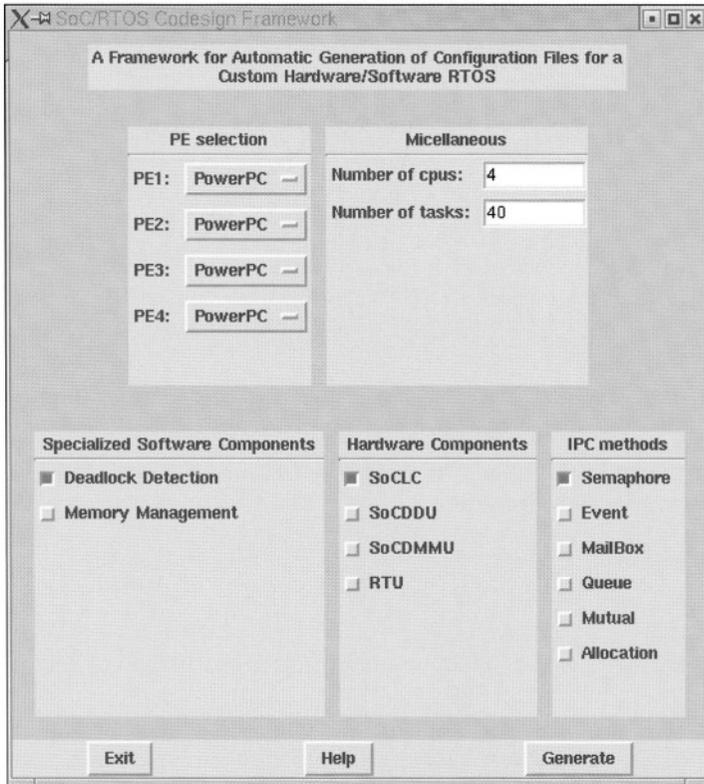
In general, commercial RTOSes available for popular embedded processors provide significant reduction in design time. However, they have to be general and might not be efficient enough for specific applications. To overcome this disadvantage, some previous work has been done in the area of automatic RTOS generation [1, 2]. Using these proposed methodologies, the user can take advantage of many benefits such as a smaller RTOS for embedded systems, rapid generation of the RTOS, easy configuration of the RTOS and a more efficient and faster RTOS due to smaller size than commercial RTOSes. Also, some previous work about automated design of SoC architectures has been done [3, 4]. However, this previous work mainly focuses on one side or the other of automatic generation: either software or hardware. In the methodology proposed in this paper, we focus on the configuration of an RTOS which may include parts of the RTOS in hardware as well as software.

Previous work in hardware/software partitioning focused on the more general problem of partitioning an application and typically assumed either a custom execution paradigm such as co-routines or else assumed a particular software RTOS [5–7]. In contrast, our work focuses exclusively on RTOS partitioning among a few pre-defined partitions. The approach presented here could fit into the prior approaches; specifically, our approach could partition the RTOS component of the system under consideration.

## 3. APPROACH

Our framework is designed to provide automatic hardware/software configurability to support user-directed hardware/software partitioning.

A Graphical User Interface (GUI) (see Figure 15-2) allows the user to select



Colour picture

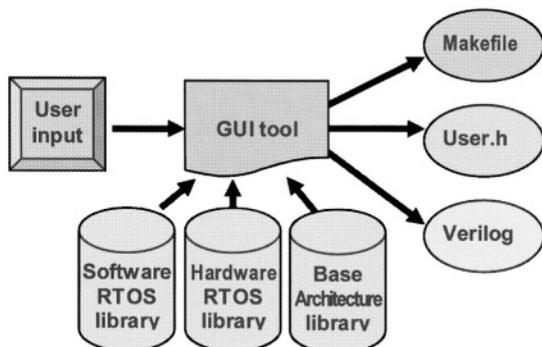
Figure 15-2. Graphic user interface for the  $\delta$  framework.

desired RTOS features most suitable for the user's needs [8–10]. Some RTOS features have both hardware and software versions available. Figure 15-2 is a snapshot of the “GUI tool” shown in the center of Figure 15-3.

### 3.1. Methodology

Figure 15-3 shows a novel approach to automating the partitioning of a hardware/software RTOS between a few pre-designed partitions. The  $\delta$  hardware/software RTOS generation framework takes as input the following four items:

- **Hardware RTOS Library**  
This hardware RTOS library currently consists of SoCLC, SoCDDU and SoCDMMU [11–16].
- **Base System Library**  
The base system library consists of basic elements such as bus arbiters and memory elements such as various caches (L1, L2, etc.). Furthermore,



Colour picture

Figure 15-3. Flow of automatic generation of configuration files.

we also need in the base system library I/O pin descriptions of all processors supported in our system.

- **Software RTOS Library**

In our case, the software RTOS library consists of Atalanta [17].

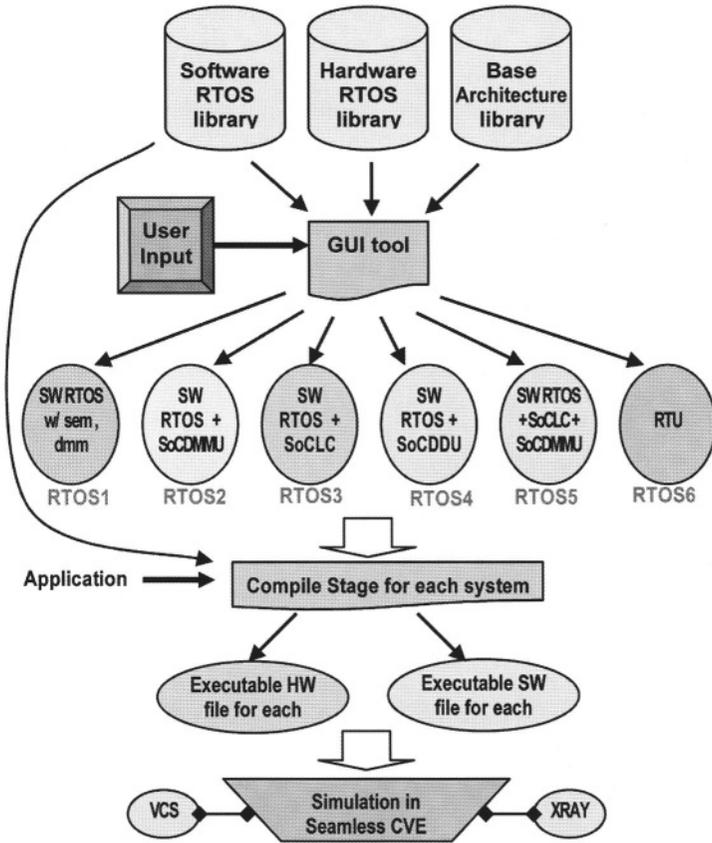
- **User Input**

Currently the user can select number of processors, type of each processor (e.g., PowerPC 750 or ARM9TDMI), deadlock detection in hardware (SoCDDU) or software, dynamic memory management in hardware (SoCDMMU) or software, a lock cache in hardware (SoCLC), and different Inter-Procedure Call (IPC) methods. (Note that while all the IPC methods are partly implemented in software, the IPC methods might also depend on hardware support – specifically, if SoCLC is chosen, then lock variables will be memory mapped to the SoCLC.)

The three files on the right-hand-side of Figure 15-3 (the Makefile, User.h and Verilog files) show the configuration files output to glue together the hardware/software RTOS Intellectual Property (IP) library components chosen in the multiprocessor SoC specified by the user.

### 3.2. Target SoC

The target system for the  $\delta$  Framework is an SoC consisting of custom logic, reconfigurable logic and multiple PEs sharing a common memory, as shown in Figure 15-1. Note that all of the hardware RTOS components (SoCDMMU, SoCLC and SoCDDU) have well-defined interfaces to which any PE – including a hardware PE (i.e., a non Von-Neumann or non-instruction-set processing element) – can connect and thus use the hardware RTOS component’s features. In other words, both the custom logic and reconfigurable logic can contain specialized PEs which interface to the hardware RTOS components.



Colour picture

Figure 15-4. Six custom hardware/software RTOS examples and simulation.

Figure 15-4 shows the generation of five different hardware/software RTOSes based on five different user specifications (specified using Figure 15-2). Rather than discussing the details of the overview of Figure 15-4, we instead focus on specific comparisons, such as RTOS1 versus RTOS2, in the following section. After the specific comparisons, we will return to the overview of Figure 15-4 and then conclude.

## 4. CUSTOM HARDWARE RTOS COMPONENTS

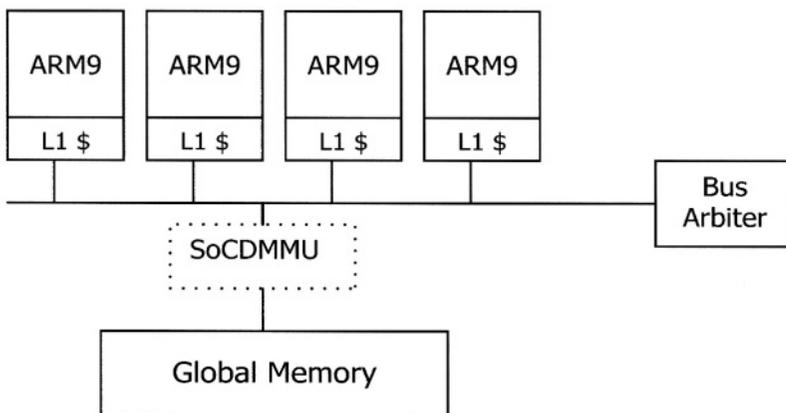
### 4.1. The SoCDMMU (RTOS1 vs. RTOS2)

In a multiprocessor SoC such as the one shown in Figure 15-1, current state-of-the-art dynamic memory management typically relies on optimized libraries. In fact, in the case of real-time systems, most commercial RTOSes

do not support dynamic memory management in the kernel due to the large overheads and unpredictable timing behavior of the code needed to implement, for example, the `malloc()` function in C. For example, eCOS does not currently support dynamic memory allocation in its kernel but instead provides such allocation as a user-level function. A review of prior work in software and hardware RTOS support for dynamic memory management is contained in [14], [15] and [18].

Figure 15-5 shows the SoC Dynamic Memory Management Unit (SoCDMMU) used in an SoC with four ARM processors. The SoCDMMU dynamically allocates Level 2 (L2) memory – “Global Memory” in Figure 15-5 – as requested by a Processing Element (PE), e.g., an ARM processor. Each PE handles the local dynamic memory allocation of the L2 memory among the processes/threads running on the PE.

The SoCDMMU assumes that the SoC L2 memory is divided into blocks or pages. For example, a 16 MB L2 memory could be divided into 256 equally sized blocks each of size 64 KB. The SoCDMMU supports four types of allocations of Global Memory (L2): (i) exclusive allocation *G\_alloc\_ex* where the PE allocating the memory will be the only user of the memory; (ii) read-write allocation *G\_alloc\_rw* where the PE allocating the memory may read from or write to the memory; (iii) read-only allocation *G\_alloc\_ro* where the PE allocating the memory will only read from (and never write to) the memory; and (iv) deallocation *G\_dealloc\_ex*. Figure 15-6 shows the worst-case execution time (WCET) of each of these four instructions of the SoCDMMU when implemented in semi-custom VLSI using a TSMC 0.25 $\mu$  library from LEDA Systems and a 200MHz clock [14].



**Figure 5. SoCDMMU Used in a Multiprocessor SoC**

*Figure 15-5. SoCDMMU used in a multiprocessor SoC.*

<b>Command</b>	<b>Number of Cycles</b>
<i>G_alloc_ex</i>	4
<i>G_alloc_rw</i>	4
<i>G_alloc_ro</i>	3
<i>G_dealloc</i>	4
4-Processors WCET	16

Figure 15-6. SoCDMMU WCETs.

Example 5-1: Consider the multiprocessor SoC shown in Figure 15-5. Each ARM processor runs at 200 MHz and has instruction and data L1 caches each of size 64 MB. The global memory is 16 MB and requires five cycles of latency to access the first word. A handheld device utilizing this SoC can be used for Orthogonal Frequency Division Multiplexing (OFDM) communication as well as other applications such as MPEG2. Initially the device runs an MPEG2 video player. When the device detects an incoming signal it switches to the OFDM receiver. The switching time (which includes the time for memory management) should be short or the device might lose the incoming message. The following table shows the sequence of memory deallocations by the MPEG2 player and then the sequence of memory allocations for the OFDM application.

<b>MPEG-2 Player</b>	<b>OFDM Receiver</b>
2 Kbytes	34 Kbytes
500 Kbytes	32 Kbytes
5 Kbytes	1 Kbytes
1500 Kbytes	1.5 Kbytes
1.5 Kbytes	32 Kbytes
0.5 Kbytes	8 Kbytes
	32 Kbytes

We measure the execution time for the memory (de)allocations from the application programming interface (API) level, e.g., from malloc(). In the case where the SoCDMMU is used, the API makes special calls via memory-mapped I/O locations to request and receive global memory from the SoCDMMU. Thus, Figure 15-7 shows execution times which compare software API plus hardware execution time (using the SoCDMMU) versus only software (using optimized ARM software). Please note that the

	Using the SoCDMMU	Using ARM SDT <i>malloc()</i> and <i>free()</i>	Speedup
Average Case	281 cycles	1240 cycles	4.4X*
Worst Case	1244 cycles	4851 cycles	3.9X

Figure 15-7. Speedup in dynamic memory allocation time.

speedups shown exceed 10x when compared to GCC libc memory management functions [14, 15, 18].

An area estimate for Figure 15-5 indicates that, excluding the SoCDMMU, 154.75 million transistors are used [18]. Since the hardware area used by the SoCDMMU logic is approximately 7,500 transistors, 30 K transistors are needed for a four processor configuration such as Figure 15-5. An additional 270 K transistors are needed for memory used by the SoCDMMU (mainly for virtual to physical address translation) for the example above [18]. Thus, for 300 K (0.19%) extra chip area in this 154.75 million transistor chip, a 4–10x speedup in dynamic memory allocation times is achieved. ■

#### 4.2. The SoCLC (RTOS1 vs. RTOS3)

The System-on-a-Chip Lock Cache (SoCLC) removes lock variables from the memory system and instead places them in a special on-chip “lock cache.” Figure 15-8 shows a sample use of the SoCLC in a 4-processor system.

The right-hand-side of Figure 15-8 shows that the SoCLC keeps track of lock requestors ( $Pr_1$  through  $Pr_N$ ) and generates interrupts to wake up the next in line (in FIFO or priority order) when a lock becomes available [11–13]. The core idea of the SoCLC is to implement atomic test-and-set in an SoC without requiring any changes to the processor cores used.

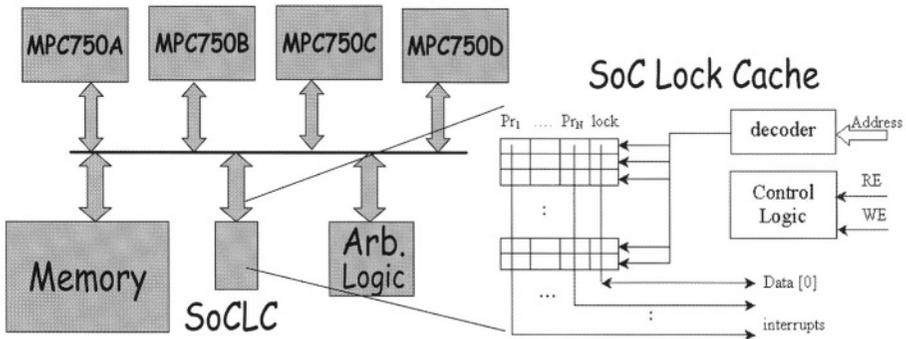
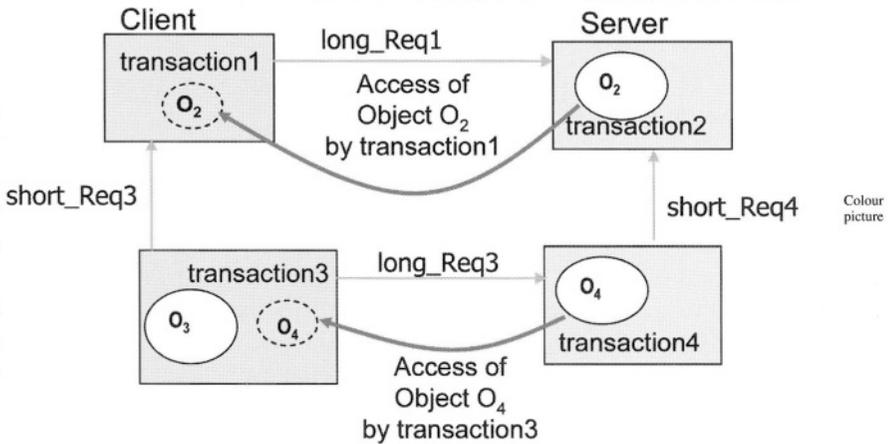


Figure 15-8. SoCLC in an SoC with four MPC750s.

Example 5-2: The following picture shows a database example [28]. Each database access requires a lock so that no other process can accidentally alter the database object at the same time. Note that a “short” request indicates that no context switch is allowed to occur since all access-es to the object are very short (e.g., less than the time it takes to switch context). A “long” request, on the other hand, requires a significant amount of time holding the database object (and hence the lock); thus, for a “long” request, if the lock is not available then the requesting process is context switched out by the RTOS running on the processor.



We ran this example with 40 tasks – 10 server tasks and 30 client tasks – on the four-MPC750 SoC architecture shown in Figure 15-8 with 10 task on each MPC750. The results are shown in the following table. Please note that *lock latency* is the time required to access a lock in the absence of contention (i.e., the lock is free); also, please note that the 908 cycles shown in the following table for lock latency in the “With SoCLC” is measured from before to after the software application level API call. Thus, in the “With SoCLC” case, part of the 908 cycles reported are spent executing API software including specialized assembly instructions which

	* Without SoCLC	With SoCLC	Speedup
Lock Latency (clock cycles)	1200	908	1.32x
Lock Delay (clock cycles)	47264	23590	2.00x
Execution Time (clock cycles)	36.9M	29M	1.27x

\* Semaphores for long CSEs and spin-locks for short CSEs are used instead of SoCLC.

interact with the SoCLC hardware via memory-mapped I/O reads and writes. Furthermore, please note that lock delay is the average time taken to acquire a lock, including contention. Thus, part of the 2× speedup in *lock delay* when using the SoCLC is achieved due to faster lock access time, and part is due to reduced contention and thus reduced bottlenecks on the bus. The overall speedup for this example is 27%.

Since each lock variable requires only one bit, the hardware cost is very low. Table 15-4 (see the end of this chapter) reports an example where 128 lock variables (which are enough for many real-time applications) cost approximately 7,400 logic gates of area. ■

## 5. THE $\delta$ HARDWARE/SOFTWARE RTOS GENERATION FRAMEWORK FOR SOC

### 5.1. Hardware and Software RTOS Library Components

The  $\delta$  Framework implements a novel approach for automating the partitioning of a hardware/software RTOS between a few pre-designed partitions. The flow of automatic generation of configuration files is shown in Figure 15-9. Specifically, our framework, given the intellectual property (IP) library of processors and RTOS components, translates the user choices into a hardware/software RTOS for an SoC. The GUI tool generates configuration files: header files for C pre-processing, a Makefile and some domain specific code files such as Verilog files to glue the system together. To test our tool, we execute our different RTOS configurations in the Mentor Graphics Seamless Co-Verification Environment (CVE) [19]. The Seamless framework provides Processor Support Packages (PSPs) and Instruction Set Simulators (ISSes) for processors, e.g., for ARM920T and MPC750.

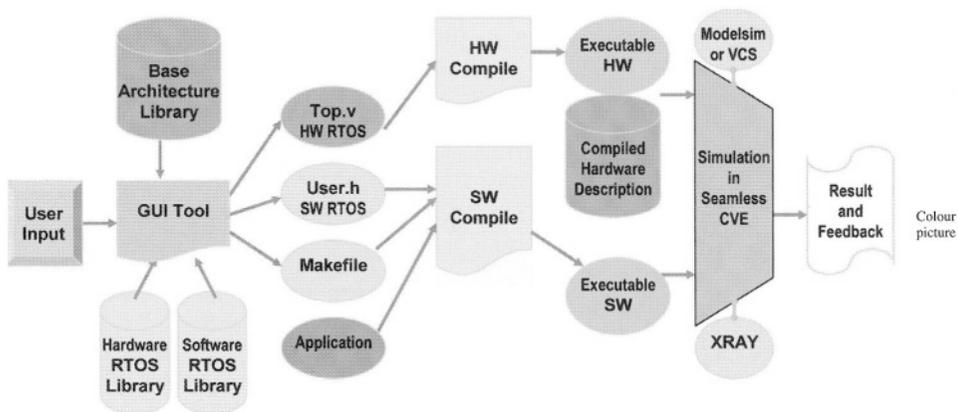
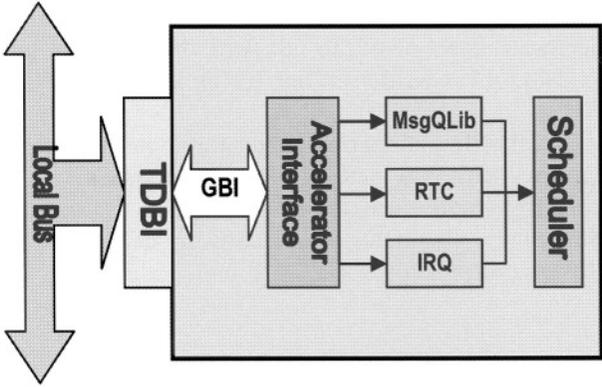


Figure 15-9. The  $\delta$  framework and current simulation environment.

For RTOS hardware IP, we start with an IP library of hardware components consisting of the SoCDMMU, SoCLC, System-on-a-Chip Deadlock Detection Unit (SoCDDU) [16] and Real-Time Unit (RTU) [20–24]. The SoCDMMU and SoCLC were already described in the previous section; IP-generators for the SoCDMMU and SoCLC can generate RTL Verilog code for almost any configuration of the SoCDMMU and/or SoCLC desired [26, 27]. The SoCDDU performs a novel parallel hardware deadlock detection based on implementing deadlock searches on the resource allocation graph in hardware [16]. It provides a very fast and very low area way of checking deadlock at run-time with dedicated hardware. The SoCDDU reduces deadlock detection time by 99% as compared to software.

Figure 15-10 shows a hardware RTOS unit called RTU [20–24]. The RTU is a hardware operating system that moves the scheduling, inter-process communication (IPC) such as semaphores as well as time management control such as time ticks and delays from the software OS-kernel to hardware. The RTU decreases the system overhead and can improve predictability and response time by an order of magnitude. (This increased performance is due to the reduced CPU load when the RTOS functionality is placed into hardware.) The RTU also dramatically reduces the memory footprint of the OS to consist of only a driver that communicates with the RTU. Thus, less cache misses are seen when the RTU RTOS is used. The RTU also supports task and semaphore creation and deletion, even dynamically at run time.

For RTOS software IP, we use the Atalanta RTOS, a shared-memory multi-processor real-time operating system developed at the Georgia Institute of Technology [17]. This RTOS is specifically designed for supporting multiple processors with large shared memory in which the RTOS is located and is similar to many other small RTOSes. All PEs (currently supported are either all MPC750 processors or all ARM9 processors) execute the same RTOS code and share kernel structures, data and states of all tasks. Each PE, however, runs its own task(s) designated by the user. Almost all software modules are



Colour picture

Figure 15-10. The Real-Time Unit (RTU).

pre-compiled and stored into the Atalanta library. However, some modules have to be linked to the final executable file from the object module itself because some function names have to be the same in order to provide the same API to the user. For example, if the deadlock detection function could be implemented in the RTOS either in software or in hardware, then the function name that is called by an application should be the same even though the application could use, depending on the particular RTOS instantiated in the SoC, either the software deadlock detection function or a device driver for the SoCDDU. By having the same API, the user application does not need modification whichever method – either software or hardware – is actually implemented in the RTOS and the SoC.

## 5.2. Implementation

In this section, we describe which configuration files are generated and how these files are used to make a custom hardware/software RTOS.

To ease the user effort required to manually configure the hardware/software RTOS, we made the GUI tool shown in Figure 15-2 for user inputs. With the GUI tool, the user can select necessary RTOS components that are most suitable for his application. Currently, the user may select the following items in software: IPC methods such as semaphores, mailboxes and queues; schedulers such as priority or round-robin scheduler; and/or a deadlock detection module. The user may also select the following items in hardware: SoCDMMU, SoCLC, SoCDDU and/or RTU. With the  $\delta$  Framework, the hardware/software RTOS configuration process is easily scalable according to the number of PEs and IPC methods. One example of scalability is that if the user selects the number of PEs, the tool can adaptively generate an appropriate configuration that contains the given number of PE wrappers and the interfaces gluing PEs to the target system.

For pre-fabrication design space exploration, different PEs and the number of PEs can be selected. For post-fabrication customization of a platform SoC with reconfigurable logic (e.g., a specific fabricated version of Figure 15-1), the user can decide whether or not to put parts of the RTOS into the reconfigurable logic. The tool is written in the Tcl/Tk language [9].

**Example 5-3: Makefile generation.** After the input data shown in Figure 15-2 is entered by the user in the GUI, the clicks the Generate button shown on the bottom right of Figure 15-2. This causes the tool to generate a Makefile containing assignments saying software deadlock detection object module and device driver for SoCLC are included. ■

5.2.1. The linking process of specialized software components for a function with a different implementation

To generate a smaller RTOS from the given library, only the needed components are included in the final executable file. One of the methods to achieve this is very straightforward. For example, when the user selects the software deadlock detection component, then the tool generates a *Makefile* that includes the software deadlock detection object. On the other hand, when the user selects the hardware deadlock detection unit, then the tool generates a different *Makefile* that includes only the software device driver object containing APIs that manipulate the hardware deadlock detection unit. Therefore, the final executable file will have either the software module or the device driver module.

5.2.2. IPC module linking process

On the other hand, when the user selects IPC methods, the inclusion process is more complicated than the linking process of specialized software components. IPC modules that implement IPC methods (such as queue, mailbox, semaphore and event) are provided as separate files. The linking process for IPC modules is shown in Figure 15-11. From the GUI of the tool, the user can choose one or more IPC methods according to his application requirements. For example, as shown in Figure 15-3, when the user selects only the semaphore component among IPC methods, the tool generates a *user.h* file which has a semaphore definition and is used by the C pre-processor. Without automation, *user.h* must be written by hand. This *user.h* file will then be

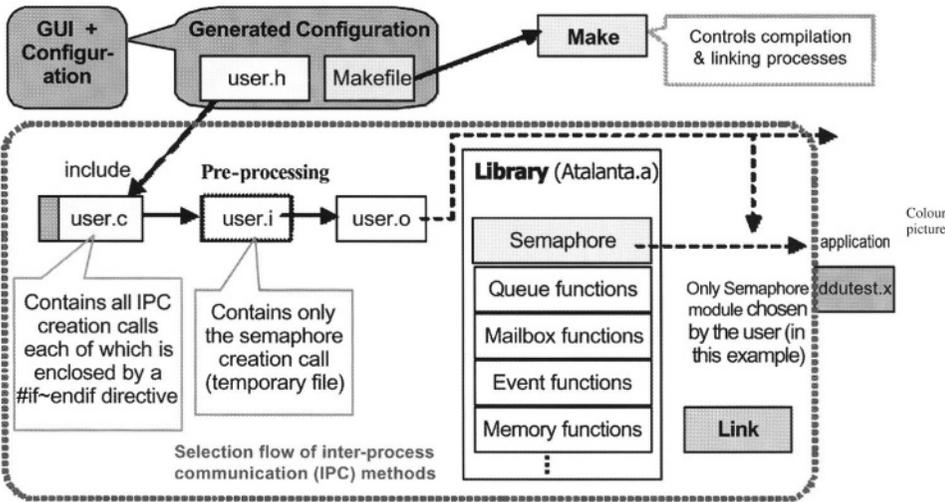


Figure 15-11. Example of the IPC module linking process.

included into *user.c*, which contains all the call routines of IPC creation functions. Each of the call routines is enclosed by a *#if~#endif* compiler directive and also corresponds to a C source file. During C pre-processing, the compiler will include the semaphore call routine. Thus, during linking, the linker will include the semaphore module from the Atalanta library in the final executable file because the semaphore creation function is needed (the creation and management functions for semaphores are contained in *semaphore.c* in Atalanta [17]).

Please note that, as described earlier, we use PSPs from Mentor Graphics; therefore, we only generate PE wrappers instead of IP cores themselves.

### 5.2.3. The configuration of the hardware modules of an RTOS

In this section, we describe how a user-selected hardware RTOS component is integrated to the target architecture. The final output of the tool is a Verilog hardware description language (HDL) header file that contains hardware IP modules such as PE wrapper, memory, bus, SoCLC and/or SoCDDU. Here, we define the “Base” system as an SoC architecture that contains only essential components (needed for almost all systems) such as PE wrappers, an arbiter, an address decoder, a memory and a clock, which are stored in the Base Architecture Library shown in Figure 15-2. Optional hardware RTOS components such as SoCLC and SoCDDU are stored in the Hardware RTOS Library. Optional hardware RTOS components chosen by the user are integrated together with the Base system. The generation of a target SoC Verilog header file (which contains the Base architecture together with configured hardware RTOS components, if any) starts with an architecture description that is a data structure describing one of the target SoC architectures. From the tool, if the user does not select any hardware RTOS components, a Verilog header file containing only the Base system will be generated. On the other hand, if the user wants to use SoCLC to support fast synchronization among PEs (e.g., to meet hard timing deadlines of an application), he can select *SoCLC* in the hardware component selection. Then the tool will automatically generate an application specific hardware file containing SoCLC. This hardware file generation is performed by a C program, *Archi\_gen*, which is compiled with a TCL wrapper.

**Example 5-4: Verilog file generation.** Here we describe in more detail the generation process of a top-level Verilog file for an SoCLC system (see Figure 15-8) step by step. When the user selects ‘*SoCLC*’ in the tool, as shown in Figure 15-2, and clicks the ‘*Generate*’ button, the tool calls a hardware generation program, *Archi\_gen* (written in C), with architecture parameters. Then the tool makes a compile script for compiling the generated Verilog file. Next, *Archi\_gen* makes the hardware Verilog file as follows (see Figure 15-12). First, *Archi\_gen* extracts (i) all needed modules according to the SoCLC system description of Figure 15-8.

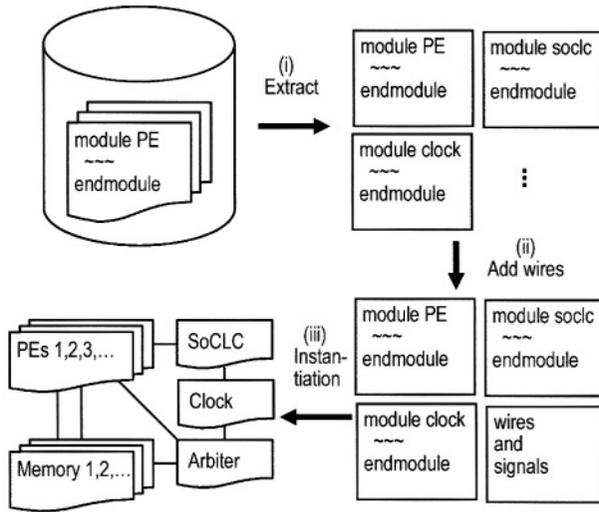


Figure 15-12. HDL file generation process.

Second, Archi\_gen generates (ii) the code for wiring up the SoCLC system (including all buses). Third, Archi\_gen generates (iii, see Figure 15-12) the instantiation code according to the instantiation type of the hardware modules chosen. In this step, Archi\_gen also generates instantiation code for the appropriate number of PEs according to the user selection of the number of PEs in the tool. In this example, Archi\_gen generates code for MPC750 instantiation four times (since the architecture has four MPC750s). Fourth, Archi\_gen extracts the initialization code (needed for test) for necessary signals according to the SoCLC initialization description. Finally, a Verilog header file containing an SoCLC system hardware architecture is made. ■

## 6. EXPERIMENTAL RESULTS

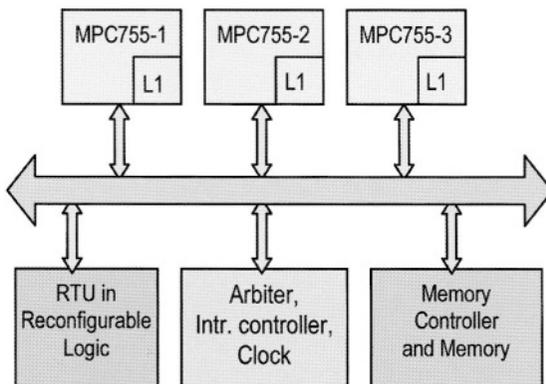
Figure 15-4 showed six possible hardware/software RTOS configurations generated by the  $\delta$  Framework. Here, we use the word “system” to refer to an SoC architecture with an RTOS. Each SoC architecture has four PEs and additional hardware modules – for example, the modules shown in Figure 15-8 (and described in Section 5.2.3). The first configuration (RTOS1) in Figure 15-4, marked as “SW RTOS w/ sem, dmm” (with semaphores and dynamic memory management software), was used in Sections 4.1 and 4.2 for comparisons with the second configuration (RTOS2) and the third (RTOS3), namely, systems containing the SoCDMMU and SoCLC, respectively. Comparisons involved the fourth configuration (RTOS4), “SW RTOS + SoCDDU” which is a system utilizing SoCDDU, and the fifth configura-

tion (RTOS5) are available in references [9] and [16]. In this section, we will focus our attention on a comparison of the first configuration (RTOS1), the third configuration (RTOS3) and the last configuration (RTOS6).

To compare RTOS1, RTOS3 and RTOS6, we decided to use the database example described in Example 5-2. The base architecture for all three systems is the same: three MPC755 processors connected via a single bus to 16 MB of L2 memory and to reconfigurable logic. This architecture can be seen in Figure 15-13 with the RTU instantiated in the reconfigurable logic. Each MPC755 has separate instruction and data caches each of size 32 KB. Our first system configuration of Figure 15-13 uses RTOS1; therefore, with a pure software RTOS, synchronization is performed with software semaphores and spin-locks in the system. Note that in this first system, all of the reconfigurable logic is available for other uses as needed. The second system uses RTOS3 and thus instantiates an SoCLC in the reconfigurable logic block of Figure 15-13. Finally, as shown in Figure 15-13, our third system includes the RTU (see Figure 15-10 in Section 5), exploiting the reconfigurable logic for scheduling, synchronization and even time-related services. In short, the hardware RTU in Figure 15-13 handles most of the RTOS services.

Simulations of two database examples were carried out on each of these three systems using Seamless CVE [15], as illustrated on the far right-hand side of Figure 15-9. We used Modelsim from Mentor Graphics for mixed VHDL/Verilog simulation and XRAY™ debugger from Mentor Graphics for application code debugging. To simulate each configured system, both the software part including application and the hardware part of the Verilog top module were compiled. Then the executable application and the multi-processor hardware setup consisting of three MPC755's were connected in Seamless CVE.

Experimental results in Table 15-1 present the total execution time of (i) simulation with software semaphores, (ii) simulation with SoCLC (hardware-supported semaphores) and (iii) simulation with RTU. As seen in the table,



Colour picture

Figure 15-13. An SoC architecture with an RTU.

Table 15-1. Average-case simulation results of the example.

Total execution time		Pure SW*	With SoCLC	With RTU
6 tasks	(in cycles)	100398	71365	67038
	Reduction	0%	29%	33%
30 tasks	(in cycles)	379400	317916	279480
	Reduction	0%	16%	26%

\* Semaphores are used in pure software while a hardware mechanism is used in SoCLC and RTU.

the RTU system achieves a 33% reduction over case (i) in the total execution time of the 6-task database application. On the other hand, the SoCLC system showed a 29% reduction over case (i). We also simulated these systems with a 30-task database application, where the RTU system and the SoCLC system showed 26% and 16% reductions, respectively, compared to the pure software RTOS system of case (i). The reason why smaller execution time reductions are seen when comparing to the pure software system in the 30-task case is that, when compared to the 6-task case, software for the 30-task case was able to take much more advantage of the caches.

In order to gain some insight to explain the performance differences observed, we looked in more detail at the different scenarios and RTOS interactions. Table 15-2 shows the total number of interactions including semaphore interactions and context switches while executing the applications. Table 15-3 shows in which of three broad areas – communication using the bus, context switching and computation – PEs have spent their clock cycles. The numbers for communication shown in Table 15-3 indicate the time period between just before posting a semaphore and just after acquiring the semaphore in a task that waits the semaphore and has the highest priority for the semaphore. For example, if Task 1 in PE1 releases a semaphore for

Table 15-2. Number of interactions.

Times	6 tasks	30 tasks
Number of semaphore interactions	12	60
Number of context switches	3	30
Number of short locks	10	58

Table 15-3. Average time spent on (6 task case).

Cycles	Pure SW	With SoCLC	With RTU
Communication	18944	3730	2075
Context switch	3218	3231	2835
Computation	8523	8577	8421

Task 2 in PE2 (which is waiting for the semaphore), the communication time period would be between just before a `post_semaphore` statement (semaphore release call) of Task 1 in PE1 and just after a `pend_semaphore` statement (semaphore acquisition call) of Task 2 in PE2. In a similar way, the numbers for context switch were measured. The time spent on communication in the pure software RTOS case is prominent because the pure software RTOS does not have any hardware notifying mechanism for semaphores, while the RTU and the SoCLC system exploit an interrupt notification mechanism for emaphores.

We also noted that the context switch time when using the RTU is not much less than others. To explain why, recall that a context switch consists of three steps: (i) pushing all PE registers to the current task stack, (ii) selecting (scheduling) the next task to be run, and (iii) popping all PE registers from the stack of the next task. While Step (ii) can be done by hardware, steps (i) and (iii) cannot be done by hardware in general PEs because all registers inside a PE must be stored to or restored from the memory by the PE itself. That is why the context switch time of the RTU cannot be reduced significantly (as evidenced in Table 15-3).

We synthesized and measured the hardware area of the SoCLC and RTU with TSMC 0.25um standard cell library from LEDA [19]. The number of total gates for the SoCLC was 7435 and the number of total gates for the RTU was approximately 250,000 as shown in Table 15-4.

Table 15-4. Hardware area in total gates.

Total area	SoCLC (64 short CS locks + 64 long CS locks)	RTU for 3 processors
TSMC 0.25 mm library from LEDA	7435 gates	About 250000 gates

In conclusion, from the information about (i) the size of a specific hardware RTOS component, (ii) the simulation results and (iii) available reconfigurable logic, the user can choose which configuration is most suitable for his or her application or set of applications.

## 7. CONCLUSION

We have presented some of the issues involved in partitioning an OS between hardware and software. Specific examples involved hardware/software RTOS partitions show up to 10x speedups in dynamic memory allocation and 16% to 33% reductions in execution time for a database example where parts of the RTOS are placed in hardware. We have shown how the  $\delta$  Framework can automatically configure a specific hardware/software RTOS specified by the user. Our claim is that SoC architectures can only benefit from early codesign with a software/hardware RTOS to be run on the SoC.

## ACKNOWLEDGEMENTS

This research is funded by NSF under INT-9973120, CCR-9984808 and CCR-0082164. We acknowledge donations received from Denali, Hewlett-Packard Company, Intel, LEDA, Mentor Graphics, SUN and Synopsys.

## REFERENCES

1. F. Balarin, M. Chiodo, A. Jurecska, and L. Lavagno. "Automatic Generation of Real-Time Operating System for Embedded Systems." *Proceedings of the Fifth International Workshop on Hardware/Software Co-Design (CODES/CASHE '97)*, pp. 95–100, 1997, <http://www.computer.org/proceedings/codes/7895/78950095abs.htm>.
2. L. Gauthier, S. Yoo, and A. Jerraya. "Automatic Generation and Targeting of Application-Specific Operating Systems and Embedded Systems Software." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 20, No. 11, pp. 1293–1301, November 2001.
3. D. Lyonnard, S. Yoo, A. Baghdadi, and A. Jerraya. "Automatic Generation of Application-Specific Architectures for Heterogeneous Multiprocessor System-on-Chip." *Proceedings of the 38th ACM/IEEE Design Automation Conference (DAC'01)*, pp. 518–523, June 2001.
4. S. Vercauteren, B. Lin, and H. Man. "Constructing Application-Specific Heterogeneous Embedded Architectures from Custom Hardware/Software Applications." *Proceedings of the 33rd ACM/IEEE Design Automation Conference (DAC'96)*, pp. 521–526, June 1996.
5. R. K. Gupta. *Co-Synthesis of Hardware and Software for Digital Embedded Systems*. Kluwer Academic Publishers, Boston, Massachusetts, USA, 1995.
6. G. De Micheli and M. Sami (eds.). *Hardware/Software Co-Design*. Publishers, Norwell, Massachusetts, USA, 1996.
7. *Proceedings of the IEEE*, Special Issue on Hardware/Software Co-Design, IEEE Press, Piscataway, New Jersey, USA, March 1997.
8. V. Mooney and D. Blough. "A Hardware-Software Real-Time Operating System Framework for SoCs." *IEEE Design & Test of Computers*, Volume 19, Issue 6, pp. 44–51, November-December 2002.
9. J. Lee, K. Ryu, and V. Mooney. "A Framework for Automatic Generation of Configuration Files for a Custom Hardware/Software RTOS." *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA '02)*, pp. 31–37, June 2002.
10. J. Lee, V. Mooney, A. Daleby, K. Ingstrom, T. Klevin, and L. Lindh. "A Comparison of the RTU Hardware RTOS with a Hardware/ Software RTOS." *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC '03)*, pp. 683–688, January 2003.
11. B. Saglam (Akgul) and V. Mooney. "System-on-a-Chip Processor Support in Hardware." *Proceedings of Design, Automation, and Test in Europe (DATE '01)*, pp. 633–639, March 2001.
12. B. S. Akgul, J. Lee, and V. Mooney. "System-on-a-Chip Processor Synchronization Hardware Unit with Task Preemption Support." *Proceedings of International Conference Compilers, Architecture, and Synthesis for Embedded Systems (CASES '01)*, pp. 149–157, October 2001.
13. B. S. Akgul and V. Mooney. "The System-on-a-Chip Lock Cache." *Design Automation of Embedded Systems*, Vol. 7, Nos. 1–2, pp. 139–174, September 2002.
14. M. Shalan and V. Mooney. "A Dynamic Memory Management Unit for Embedded Real-Time System-on-a-Chip." *Proceedings of International Conference Compilers, Architecture, and Synthesis for Embedded Systems (CASES '00)*, pp. 180–186, October 2000.
15. M. Shalan and V. Mooney. "Hardware Support for Real-Time Embedded Multiprocessor

- System-on-a-Chip Memory Management.” *Proceedings of International Symposium Hardware/Software Co-design (CODES '02)*, pp. 79–84, May 2002.
16. P. Shiu, Y. Tan, and V. Mooney. “A Novel Parallel Deadlock Detection Algorithm and Architecture.” *Proceedings of International Symposium Hardware/Software Codesign (CODES '01)*, pp. 30–36, April 2001.
  17. D. Sun et al. “Atalanta: A New Multiprocessor RTOS Kernel for System-on-a-Chip Applications,” Tech. Report GIT-CC-02-19, Georgia Institute of Technology, Atlanta, 2002, <http://www.cc.gatech.edu/pubs.html>.
  18. M. Shalan and V. Mooney. “Hardware Support for Real-Time Embedded Multiprocessor System-on-a-Chip Memory Management.” Georgia Institute of Technology, Atlanta, Georgia, Technical Report GIT-CC-03-02, 2003, [http://www.cc.gatech.edu/tech\\_reports/](http://www.cc.gatech.edu/tech_reports/).
  19. Mentor Graphics, Hardware/Software Co-Verification: Seamless, <http://www.mentor.com/seamless>.
  20. L. Lindh and F. Stanischewski. “FASTCHART – Idea and Implementation.” *Proceedings of the International Conference on Computer Design (ICCD'91)*, pp. 401–404, 1991.
  21. L. Lindh. “Idea of FASTHARD – A Fast Deterministic Hardware Based Real-Time Kernel.” *Proceedings of the European Workshop on Real-Time Systems*, June 1992.
  22. L. Lindh, J. Stärner, and J. Furunäs, “From Single to Multiprocessor Real-Time Kernels in Hardware.” *Real-Time Technology and Applications Symposium (RTAS'95)*, pp. 42–43, May 1995.
  23. J. Adomat, J. Furunäs, L. Lindh, and J. Stärner. “Real-Time Kernels in Hardware RTU: A Step Towards Deterministic and High Performance Real-Time Systems.” *Proceedings of the European Workshop on Real-Time Systems*, pp. 164–168, June 1996.
  24. RealFast, <http://www.realfast.se>
  25. J. Lee, V. Mooney, A. Daleby, K. Ingström, T. Klevin, and L. Lindh. “A Comparison of the RTU Hardware RTOS with a Hardware/Software RTOS.” *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC'03)*, pp. 683–688, January 2003.
  26. B. S. Akgul and V. Mooney. “PARLAK: Parametrized Lock Cache Generator.” *Proceedings of the Design Automation and Test in Europe Conference (DATE'03)*, pp. 1138–1139, March 2003.
  27. M. Shalan, E. Shin, and V. Mooney. “DX-Gt: Memory Management and Crossbar Switch Generator for Multiprocessor System-on-a-Chip.” *Workshop on Synthesis and System Integration of Mixed Information Technologies (SASIMI'03)*, April 2003.
  28. M. Olson. “Selecting and Implementing an Embedded Database System.” *IEEE Computer Magazine*, pp. 27–34, September 2000.

# Chapter 16

## EMBEDDED SW IN DIGITAL AM-FM CHIPSET

M. Sarlotte, B. Candaele, J. Quevremont, and D. Merel

*THALES Communications, Gennevilliers France*

**Abstract.** DRM, the new standard for digital radio broadcasting in AM band requires integrated devices for radio receivers at low cost and very low power consumption. A chipset is currently designed based upon an ARM9 multi-core architecture. This paper introduces the application itself, the HW architecture of the SoC and the SW architecture which includes physical layer, receiver management, the application layer and the global scheduler based on a real-time OS. Then, the paper presents the HW/SW partitioning and SW breakdown between the various processing cores. The methodology used in the project to develop, validate and integrate the SW covering various methods such as simulation, emulation and co-validation is described. Key points and critical issues are also addressed.

One of the challenges is to integrate the whole receiver in the mono-chip with respect to the real-time constraints linked to the audio services.

**Key words:** HW/SW co-design, ARM core, physical layer, AHB multi-layers, network-on-chip

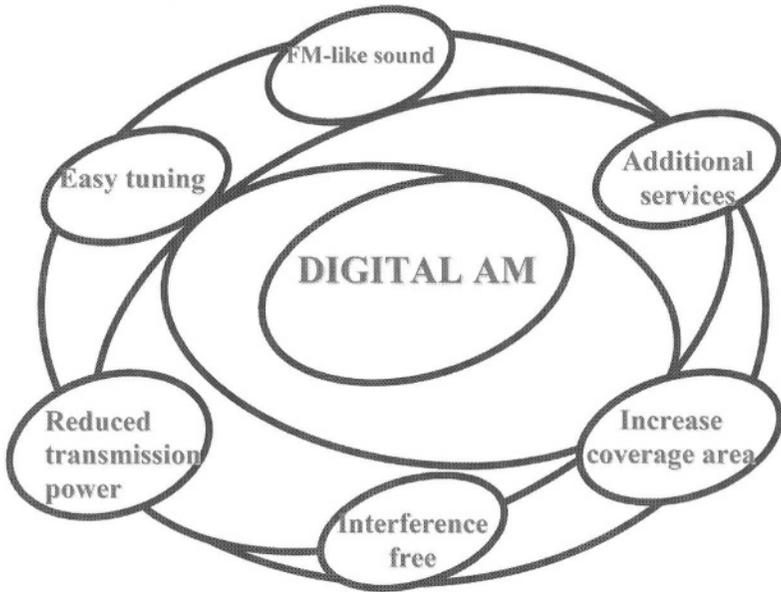
### 1. DRM APPLICATION

DRM (Digital Radio Mondiale) is a new standard promoted by the major broadcasters and receiver manufacturers. The physical layer is based upon an OFDM modulation and includes multiple configurations (channel width, code rate, protection, . . .) to cope with various propagation schemes within the short and medium waves (0 to 30 MHz). The physical layer provides up to 78kbits/s of useful data rate. The service layers are mainly composed by enhanced MP3 audio decoder and data services. The standard also introduces complementary services like Automatic Frequency Switching (AFS) and several data services. Figure 16-1 presents the main improvements proposed in this new standard.

The receiver signal processing reference requires about 500 Mips to handle the whole treatment.

### 2. DIAM SOC ARCHITECTURE

A full software implementation will not meet the specific constraints of the radio receivers in term of die size and power consumption. The first step is the HW/SW partitioning based upon the existing full-software reference



Colour picture

Figure 16-1. AM broadcasting improvements.

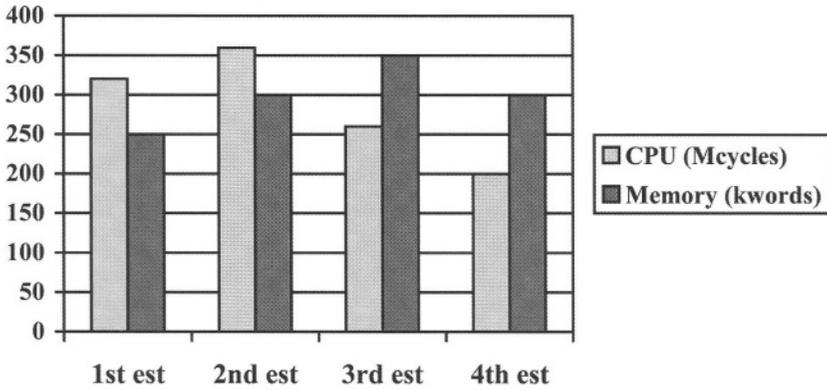
receiver. Profiling tools have been used to extract the memory and the CPU requirements for each sub function. Dedicated modules have been identified as candidates for a wired implementation. A Viterbi decoder including de-puncturing and a digital down converter (wide band conversion) have been selected.

The remaining CPU requirements decrease then below 200 Mips which are achievable using state of the art technology. Due to the characteristics of the channel coder, standard embedded DSP processors are not compatible with the addressing requirements. The selected solution is based on two ARM9 core plugged on a multi-layer AHB matrix.

The challenge is then to size the architecture in term of CPU, memory but also the throughput according to the targeted algorithms. For this purpose, a simulation tool from ARM has been used (ADS 1.2) to run quickly the code and to measure the impact of the core instructions.

However, the simulation does not take into account the real architecture of the platform in term of code execution. Main points are the impact of the cache, of the tightly coupled memory and the impact of the communication network during the execution. The variation of the estimations performed during the progress of the development highlights the difficulty to size the global architecture. Figure 16-2 shows the main evolutions of the CPU/Memory requirements.

Margin has been set to avoid any real-time issue with the first version of the platform. For flexibility purpose, each resource is also visible by each core



Colour picture

Figure 16-2. CPU/memory requirements (estimation).

thru the communication network structure but no arbiter has been integrated within the chip. The software has to manage this constraint and to properly allocate each resource. This flexible approach allows to change the software partitioning when required.

Figure 16-3 presents an overview of the platform architecture of the DIAM base band. The interconnection network is based on an AHB matrix and an APB bus for the low data rate peripherals. Two external memories (one associated with each core) are introduced to suppress potential bottlenecks between the two cores.

The first version of the platform is processed in an ATMEL CMOS 0.18 μm technology.

### 3. SW ARCHITECTURE

Basically, the software has been split according to the data flow and the available CPU. The physical layer has been mapped to one core and the application layers are mapped onto the second core. Cache and TCM sizes have been tuned to the specific requirements of the most demanding function. TCM are assigned to the audio coder which presents the strongest constraint regarding the real-time.

Figure 16-4 presents an overview of the software architecture.

The platform boot is not shown although its definition is quite tricky. This dedicated SW is in charge of the initial configuration of each core and is closely linked to the platform itself. By structure, each core starts executing its SW at the same address (generally @0). Without any precaution both cores will execute the same code. The retained solution for the DIAM platform is based upon a hierarchical approach and a remap technique which allows to translate the base of the slave core. This translation is performed during the boot of the master core which then allows the boot of the slave core.

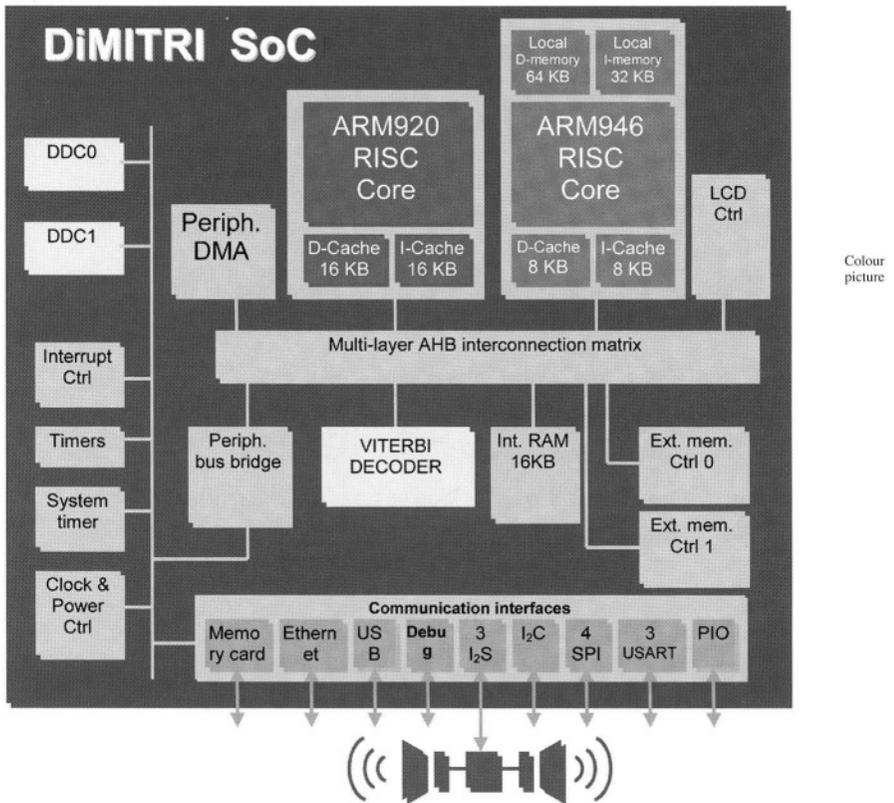


Figure 16-3. Architecture of DIAM base band.

#### 4. SOFTWARE INTEGRATION

One of the challenges of such a development is the concurrence of the HW design and the SW coding. Main issues are related to SW validation without the final platform and on the other hand to assess the performances of the architecture. To manage these issues, an heterogeneous strategy has been set-up driven by the capacity of the tools and the system characteristics.

The low-level SW (drivers and boot) has been jointly developed by the HW and SW team. As they are closely linked to the platform their validation has been performed using VHDL/Verilog simulation including a C model for the cores.

For the upper layer, a standard approach has been adopted with the exception of the MLC. Basic modules have been identified and optimised directly in ASM instead of C approach to obtain a library (ex: Filtering, FFT, . . .). This library has been validated using the ARM tools (ADS V1.2) and reference patterns provided by the signal processing algorithm. Then, the

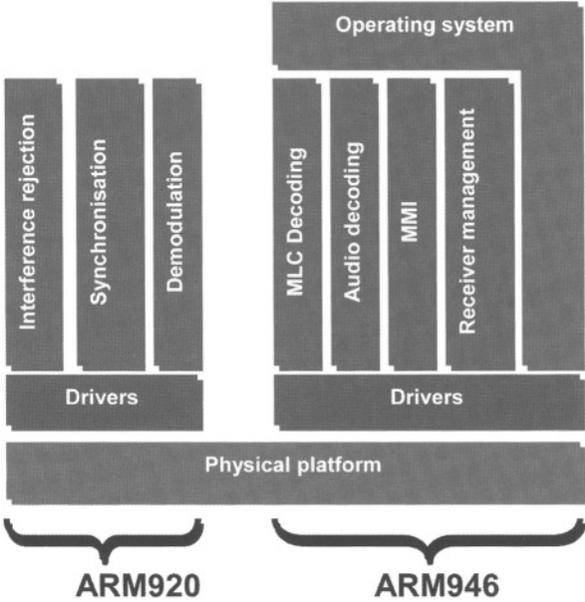


Figure 16-4. SW architecture of DIAM base band.

application is developed using the critical functions of the library. The results are validated using the Integrator board from ARM.

The MLC coding scheme (Multi-Level coding) is based upon a convolutional code combined with a complex modulation scheme. The decoding function is based on a classical Viterbi decoding algorithm (including depuncturing features). The complete function is implemented in HW and SW. Due to the complexity and the flexibility of the MLC a complete simulation of HW and SW is not achievable. A prototype on Excalibur platform from Altera has been developed. The Viterbi decoder with the de-puncturing model has been integrated in the PLD part of the devices and the SW has been mapped onto the ARM9 core. 36 sequences have been defined to obtain sufficient validation coverage and achieve a high level of confidence. These tests have highlighted one mismatch between the real implementation and the bit true model on the post processing metrics engine. The prototyping board allows to run the complete validation plan in less than 1 hour instead of 300 hours using classical HDL simulations.

Previous steps perform unitary validation without any exchange between SW modules. The next challenge is to validate the global software covering exchanges, memory mapping strategy, cache strategy, signalling protocol and the global performances of the platform. The foreseen solution is to develop a prototyping board integrating both cores and the relevant modules. Alternative solutions such as Seamless have been investigated but the execution time is too slow regarding the minimum set of simulations to be

done (at least few seconds). Based upon the current figures, 3 seconds of simulation (synchronisation and preliminary receiving phase) will take approximately 40 days (assumption: 1 Million instruction per hour). This validation will be the next step of our design.

## 5. CONCLUSIONS

The methodology applied during this HW/SW co-design allows us to cover efficiently the platform requirements. Several remaining points are still open and will be validated during the integration phase directly on the physical platform (mainly the OS, scheduler and real-time issue). However, all the open points are related to the SW and do not impact the HW definition of the platform.

Nevertheless new solution has to be built to allow complete validation of SoC including HW modules and all the SW layer including scheduler and operating system.

PART V:

SOFTWARE OPTIMISATION FOR EMBEDDED SYSTEMS

*This page intentionally left blank*

# CONTROL FLOW DRIVEN SPLITTING OF LOOP NESTS AT THE SOURCE CODE LEVEL

Heiko Falk<sup>1</sup>, Peter Marwedel<sup>1</sup>, and Francky Catthoor<sup>2</sup>

<sup>1</sup> *University of Dortmund, Computer Science 12, Germany;* <sup>2</sup> *IMEC, Leuven, Belgium*

**Abstract.** This article presents a novel source code transformation for control flow optimization called loop nest splitting which minimizes the number of executed *if*-statements in loop nests of embedded multimedia applications. The goal of the optimization is to reduce runtimes and energy consumption. The analysis techniques are based on precise mathematical models combined with genetic algorithms. The application of our implemented algorithms to three real-life multimedia benchmarks using 10 different processors leads to average speed-ups by 23.6%–62.1% and energy savings by 19.6%–57.7%. Furthermore, our optimization also leads to advantageous pipeline and cache performance.

**Key words:** cache, code size, energy, genetic algorithm, loop splitting, loop unswitching, multimedia, pipeline, polytope, runtime, source code transformation

## 1. INTRODUCTION

In recent years, the power efficiency of embedded multimedia applications (e.g. medical image processing, video compression) with simultaneous consideration of timing constraints has become a crucial issue. Many of these applications are data-dominated using large amounts of data memory. Typically, such applications consist of deeply nested *for*-loops. Using the loops' index variables, addresses are calculated for data manipulation. The main algorithm is usually located in the innermost loop. Often, such an algorithm treats particular parts of its data specifically, e.g. an image border requires other manipulations than its center. This boundary checking is implemented using *if*-statements in the innermost loop (see e.g. Figure 17-1).

Although common subexpression elimination and loop-invariant code motion [19] are already applied, this code is sub-optimal w.r.t. runtime and energy consumption for several reasons. First, the *if*-statements lead to a very irregular control flow. Any jump instruction in a machine program causes a control hazard for pipelined processors [19]. This means that the pipeline needs to be stalled for some instruction cycles, so as to prevent the execution of incorrectly prefetched instructions.

Second, the pipeline is also influenced by data references, since it can also be stalled during data memory accesses. In loop nests, the index variables are accessed very frequently resulting in pipeline stalls if they can not be

```

for (x=0; x<36; x++) { x1=4*x;
  for (y=0; y<49; y++) { y1=4*y;           /* y-loop */
    for (k=0; k<9; k++) { x2=x1+k-4;
      for (l=0; l<9; l++) { y2=y1+l-4;
        for (i=0; i<4; i++) { x3=x1+i; x4=x2+i;
          for (j=0; j<4; j++) { y3=y1+j; y4=y2+j;
            if (x3<0 || 35<x3 || y3<0 || 48<y3)
              then_block_1; else else_block_1;
            if (x4<0 || 35<x4 || y4<0 || 48<y4)
              then_block_2; else else_block_2; }}}}}}}

```

Figure 17-1. A typical Loop Nest (MPEG 4 Full Search Motion Estimation).

kept in processor registers. Since it has been shown that 50%–75% of the power consumption in embedded multimedia systems is caused by memory accesses [20, 25], frequent transfers of index variables across memory hierarchies contribute negatively to the total energy balance.

Finally, many instructions are required to evaluate the *if*-statements, also leading to higher runtimes and power consumption. For the MPEG4 code above, all shown operations are in total as complex as the computations performed in the *then*- and *else*-blocks of the *if*-statements.

In this article, a new formalized method for the analysis of *if*-statements occurring in loop nests is presented solving a particular class of the NP-complete problem of the satisfiability of integer linear constraints. Considering the example shown in Figure 17-1, our techniques are able to detect that

- the conditions  $x_3 < 0$  and  $y_3 < 0$  are never true,
- both *if*-statements are true for  $x \geq 10$  or  $y \geq 14$ .

Information of the first type is used to detect conditions not having any influence on the control flow of an application. This kind of redundant code (which is not typical dead code, since the results of these conditions are used within the *if*-statement) can be removed from the code, thus reducing code size and computational complexity of a program.

Using the second information, the entire loop nest can be rewritten so that the total number of executed *if*-statements is minimized (see Figure 17-2). In order to achieve this, a new *if*-statement (the *splitting-if*) is inserted in the  $y$  loop testing the condition  $x \geq 10 \mid \mid y \geq 14$ . The *else*-part of this new *if*-statement is an exact copy of the body of the original  $y$  loop shown in Figure 17-1. Since all *if*-statements are fulfilled when the *splitting-if* is true, the *then*-part consists of the body of the  $y$  loop without any *if*-statements and associated *else*-blocks. To minimize executions of the *splitting-if* for values of  $y \geq 14$ , a second  $y$  loop is inserted in the *then*-part counting from the current value of  $y$  to the upper bound 48 (see Figure 17-2).

As shown by this example, our technique is able to generate linear control

```

for (x=0; x<36; x++) { x1=4*x;
  for (y=0; y<49; y++)
    if (x>=10 || y>=14)                                /* Splitting-If */
      for (; y<49; y++)                                  /* Second y-loop */
        for (k=0; k<9; k++)
          ...                                           /* 1- & i-loop omitted */
          for (j=0; j<4; j++) {
            then_block_1; then_block_2; }
    else { y1=4*y;
      for (k=0; k<9; k++) { x2=x1+k-4;
        ...                                           /* 1- & i-loop omitted */
        for (j=0; j<4; j++) { y3=y1+j; y4=y2+j;
          if (0 || 35<x3 || 0 || 48<y3)
            then_block_1; else else_block_1;
          if (x4<0 || 35<x4 || y4<0 || 48<y4)
            then_block_2; else else_block_2; }}}}}

```

Figure 17-2. Loop Nest after splitting.

flow in the hot-spots of an application. Furthermore, accesses to memory are reduced significantly since a large number of branching, arithmetic and logical instructions and index variable accesses is removed.

The techniques presented in this article are fully automated. Since a loop analysis is crucial for our techniques, a high-level intermediate representation preserving loop information is required. Since this is not the case for state-of-the-art optimizing compilers (e.g. Sun-IR [18], LANCE [12]), our techniques are not realized as a compiler optimization but as a processor independent source code transformation. This independence enables a very detailed benchmarking using 10 different programmable processors.

A survey of work related to loop optimizations and source code transformations is provided in section 2. Section 3 presents the analytical models and algorithms for loop nest splitting. Section 4 shows the effects of our optimization on pipelines and caches, runtimes, energy dissipation and code sizes. Section 5 summarizes and concludes this article.

## 2. RELATED WORK

Loop transformations have been described in literature on compiler design for many years (see e.g. [1, 19]) and are often integrated into today's optimizing compilers. Classical *loop splitting* (or *loop distribution/fission*) creates several loops out of an original one and distributes the statements of the original loop body among all loops. The main goal of this optimization is to enable

the parallelization of a loop due to fewer data dependencies [1] and to possibly improve I-cache performance due to smaller loop bodies. In [10] it is shown that loop splitting leads to increased energy consumption of the processor and the memory system. Since the computational complexity of a loop is not reduced, this technique does not solve the problems that are due to the properties discussed previously.

Loop unswitching is applied to loops containing loop-invariant *if*-statements [19]. The loop is then replicated inside each branch of the *if*-statement, reducing the branching overhead and decreasing code sizes of the loops [1]. The goals of loop unswitching and the way how the optimization is expressed are equivalent to the topics of the previous section. But since the *if*-statements must not depend on index variables, loop unswitching can not be applied to multimedia programs. It is the contribution of the techniques presented in this article that we explicitly focus on loop-variant conditions. Since our analysis techniques go far beyond those required for loop splitting or unswitching and have to deal with entire loop nests and sets of index variables, we call our optimization technique *loop nest splitting*.

In [11], an evaluation of the effects of four loop transformations (loop unrolling, interchange, fusion and tiling) on memory energy is given. The authors have observed that these techniques reduce the energy consumed by data accesses, but that the energy required for instruction fetches is increased significantly. They draw the conclusion that techniques for the simultaneous optimization of data and instruction locality are needed. This article demonstrates that loop nest splitting is able to achieve these aims.

In [15], classical loop splitting is applied in conjunction with function call insertion at the source code level to improve the I-cache performance. After the application of loop splitting, a large reduction of I-cache misses is reported for one benchmark. All other parameters (instruction and data memory accesses, D-cache misses) are worse after the transformation. All results are generated with cache simulation software which is known to be unprecise, and the runtimes of the benchmark are not considered at all.

Source code transformations are studied in literature for many years. In [9], array and loop transformations for data transfer optimization are presented using a medical image processing algorithm [3]. The authors only focus on the illustration of the optimized data flow and thus neglect that the control flow gets very irregular since many additional *if*-statements are inserted. This impaired control flow has not yet been targeted by the authors. As we will show in this article, loop nest splitting applied as postprocessing stage is able to remove the control flow overhead created by [9] with simultaneous further data transfer optimization. Other control flow transformations for acceleration of address-dominated applications are presented in [7].

Genetic algorithms (GA) have proven to solve complex optimization problems by imitating the natural optimization process (see e.g. [2] for an overview). Since they are able to revise unfavorable decisions made in a previous optimization phase, GA's are adequate for solving non-linear

optimization problems. They have been successfully utilized in the domains of code generation for irregular architectures [13, 16] or VLSI CAD [4].

### 3. ANALYSIS AND OPTIMIZATION ALGORITHM

This section presents the techniques required for loop nest splitting. The optimization consists of four sequentially executed tasks. In the beginning, all conditions in a loop nest are analyzed separately without considering any interdependencies in-between. First, it is detected if conditions have any influence on the control flow of a loop nest. If not, such redundant conditions are removed from the code. For every single remaining condition, an optimized geometric model is created in the second step. Third, these geometric models are combined to a global search space representing all conditions in all *if*-statements of the loop nest simultaneously. To obtain an optimized result for a loop nest, this global search space is explored in a fourth step. Before going into details (cf. also [5, 6]), some preliminaries are required.

#### Definition 1:

1. Let  $\Lambda = \{L_1, \dots, L_N\}$  be a *loop nest* of depth  $N$ , where  $L_i$  denotes a single loop.
2. Let  $i_l$ ,  $lb_l$  and  $ub_l$  be the *index variable*, *lower bound* and *upper bound* of loop  $L_i \in \Lambda$  with  $lb_l \leq i_l \leq ub_l$ .

The optimization goal for loop nest splitting is to determine values  $lb'_l$  and  $ub'_l$  for every loop  $L_i \in \Lambda$  with

- $lb'_l \geq lb_l$  and  $ub'_l \leq ub_l$ ,
- all loop-variant *if*-statements in  $\Lambda$  are satisfied for all values of the index variables  $i_l$  with  $lb'_l \leq i_l \leq ub'_l$ ,
- loop nest splitting by all values  $lb'_l$  and  $ub'_l$  leads to the minimization of *if*-statement execution.

The values  $lb'_l$  and  $ub'_l$  are used for generating the splitting *if*-statement. The techniques described in the following require that some preconditions are met:

1. All loop bounds  $lb_l$  and  $ub_l$  are constants.
2. *If*-statements have the format  $if (C_1 \oplus C_2 \oplus \dots)$  with loop-invariant conditions  $C_x$  that are combined with logical operators  $\oplus \in \{\&\&, ||\}$ .
3. Loop-variant conditions  $C_x$  are affine expressions of  $i_l$  and can thus have the format

$$C_x = \sum_{l=1}^N (c_l * i_l) + c \geq 0 \text{ for constant } c_l, c \in \mathbb{Z}.$$

Assumption 2 is due to the current state of implementation of our tools. By applying *de Morgan's rule* on  $!(C_1 \oplus C_2)$  and inverting the comparators

in  $C_1$  and  $C_2$ , the logical *NOT* can also be modeled. Since all Boolean functions can be expressed with  $\&\&$ ,  $||$  and  $!$ , precondition 2 is not a limitation. Without loss of generality,  $a==b$  can be rewritten as  $a\leq b \ \&\& \ b\leq a$  ( $a!=b$  analogous) so that the required operator  $\geq$  of precondition 3 is not a restriction, either.

### 3.1. Condition satisfiability

In the first phases of the optimization algorithm, all affine conditions  $C_x$  are analyzed separately. Every condition defines a subset of the total iteration space of a loop nest. This total iteration space is an  $N$ -dimensional space limited by all loop bounds  $lb_i$  and  $ub_i$ . An affine condition  $C_x$  can thus be modeled as follows by a polytope:

#### Definition 2:

1.  $P = \{x \in \mathbb{Z}^N | Ax = a, Bx \geq b\}$  is called a *polyhedron* for  $A, B \in \mathbb{Z}^{m \times N}$ ,  $a, b \in \mathbb{Z}^m$  and  $m \in \mathbb{N}$ .
2. A polyhedron  $P$  is called a *polytope* if

Every condition  $C_x$  can be represented by a polytope  $P_x$  by generating inequalities for the affine condition  $C_x$  itself and for all loop bounds. For this purpose, an improved variant of the Motzkin algorithm [17] is used and combined with some simplifications removing redundant constraints [23].

After that, we can determine in constant time if the number of equalities  $Ax = a$  of  $P_x$  is equal to the dimension of  $P_x$  plus 1. If this is true,  $P_x$  is overconstrained and defines the empty set as proven by Wilde [23]. If instead  $P_x$  only contains the constraints for the loop bounds,  $C_x$  is satisfied for all values of the index variables  $i_j$ . Such conditions that are always satisfied or unsatisfied are replaced by their respective truth value in the *if*-statement and are no longer considered during further analysis.

### 3.2. Condition optimization

For conditions  $C$  that are not eliminated by the previous method, a polytope  $P_C$  is created out of values  $lb'_{C,i}$  and  $ub'_{C,i}$  for all loops  $L_i \in \Lambda$  such that  $C$  is satisfied for all index variables  $i_j$  with  $lb'_{C,i} \leq i_j \leq ub'_{C,i}$ . These values are chosen so that a loop nest splitting using  $lb'_{C,i}$  and  $ub'_{C,i}$  minimizes the execution of *if*-statements.

Since affine expressions (see precondition 3) are linear monotone functions, it is unnecessary to deal with two values  $lb'_{C,i}$  and  $ub'_{C,i}$ . If  $C$  is true for a value  $v \in [lb'_{C,i}, ub'_{C,i}]$  and  $c_i > 0$ ,  $C$  must also be true for  $v + 1, v + 2, \dots$  ( $c_i < 0$  analogous). This implies that either  $lb'_{C,i} = lb_i$  or  $ub'_{C,i} = ub_i$ . Thus, our optimization algorithm only computes values  $v'_{C,i}$  for  $C$  and all loops  $L_i \in \Lambda$  with  $v'_{C,i} \in [lb_i, ub_i]$ .  $v'_{C,i}$  designates one of the former values  $lb'_{C,i}$  or  $ub'_{C,i}$ , the other one is set to the correct upper or lower loop bound.

The optimization of the values  $v'_{C,l}$  is done by a genetic algorithm (GA) [2]. The chromosome length is set to the number of index variables  $i_l$   $C$  depends on:  $\{|c_l|c_l \neq 0\}$ . For every such variable  $i_l$ , a gene on the chromosome represents  $v'_{C,l}$ . Using the  $v'_{C,l}$  values of the fittest individual, the optimized polytope  $P_C$  is generated as the result of this phase:

$$P_C = \{(x_1, \dots, x_N) \in \mathbb{Z}^N \mid lb_l \leq x_l \leq ub_l, L_l \in \Lambda, \\ x_l (v'_{C,l} \text{ if } c_l > 0, \\ x_l (v'_{C,l} \text{ if } c_l < 0)\}$$

The fitness of an individual  $I$  is the higher, the fewer *if*-statements are executed when splitting  $\Lambda$  using the values  $v'_{C,l}$  encoded in  $I$ . Since only the fittest individuals are selected, the GA minimizes the execution of *if*-statements. Consequently, an individual implying that  $C$  is not satisfied has a very low fitness. For an individual  $I$ , the fitness function computes the number of executed *if*-statements  $IF_{Tot}$ . Therefore, the following values are required:

**Definition 3:**

1. The *total iteration space (TS)* of a loop nest  $\Lambda$  is the total number of executions of the body of loop  $L_N$ :

$$TS = \prod_{l=1}^N (ub_l - lb_l + 1)$$

2. The *constrained iteration space (CS)* is the total iteration space reduced to the ranges  $r_l$  represented by  $v'_{C,l}$ :

$$CS = \prod_{l=1}^N r_l \text{ and } r_l = \begin{cases} ub_l - lb_l + 1 & \text{if } c_l = 0, \\ ub_l - v'_{C,l} + 1 & \text{if } c_l > 0, \\ v'_{C,l} + lb_l + 1 & \text{else} \end{cases}$$

3. The *innermost loop  $\lambda$*  is the index of the loop where a loop nest splitting has to be done for a given set of  $v'_{C,l}$  values:

$$\lambda = \max \{l \mid L_l \in \Lambda, r_l \neq ub_l - lb_l + 1\}$$

The aggregate number of executed *if*-statements  $IF_{Tot}$  is computed as follows:

- $IF_{Tot} = IF_{Orig} + IF_{Split}$   
(*if*-statements in the *else*-part of the splitting-*if* plus the splitting-*if* itself)
- $IF_{Orig} = TS - CS$   
(all iterations of  $\Lambda$  minus the ones where the splitting-*if* evaluates to true)
- $IF_{Split} = TP_{Split} + EP_{Split}$   
(splitting-*if* is evaluated as often as its *then*- and *else*-parts are executed)
- $TS_{Split} = CS / \prod_{l=\lambda+1}^N (ub_l - lb_l + 1) * r_\lambda$

(all loop nest iterations where splitting-*if* is true divided by all loop iterations located in the *then*-part)

$$- EP_{Split} = IF_{Orig} / \prod_{l=\lambda+1}^N (ub_l - lb_l + 1)$$

(all loop nest iterations where splitting-if is false divided by all loop iterations located in the else-part)

The computation of  $IF_{Split}$  is that complex because the duplication of the innermost loop  $\lambda$  in the *then*-part of the splitting-if (e.g. the  $y$  loop in Figure 17-2) has to be considered. Since  $IF_{Tot}$  does not depend linearly on  $v'_{C,i}$ , a modeling of the optimization problem using integer linear programming (ILP) is impossible, so that we chose to use a GA.

Example: For a condition  $C = 4*x+k+i-40 >= 0$  and the loop nest of Figure 17-1, our GA can generate the individual  $I = (10, 0, 0)$ . The numbers encoded in  $I$  denote the values  $v'_{C,x}$ ,  $v'_{C,k}$  and  $v'_{C,i}$  so that the following intervals are defined:  $x \in [10, 35]$ ,  $k \in [0, 8]$ ,  $i \in [0, 3]$ . Since only variable  $x$  is constrained by  $I$ , the  $x$ -loop would be split using the condition  $x >= 10$ . The formulas above imply a total execution of 12,701,020 *if*-statements ( $IF_{Tot}$ ).

### 3.3. Global search space construction

After the execution of the first GA, a set of *if*-statements  $IF_i = (C_{i,1} \oplus C_{i,2} \oplus \dots \oplus C_{i,n})$  consisting of affine conditions  $C_{i,j}$  and their associated optimized polytopes  $P_{i,j}$  are given. For determining index variable values where all *if*-statements in a program are satisfied, a polytope  $G$  modeling the global search space has to be created out of all  $P_{i,j}$ .

In a first step, a polytope  $P_i$  is built for every *if*-statement  $IF_i$ . In order to reach this goal, the conditions of  $IF_i$  are traversed in their natural execution order  $\pi$  which is defined by the precedence rules of the operators  $\&\&$  and  $||$ .  $P_i$  is initialized with  $P_{i,\pi(1)}$ . While traversing the conditions of *if*-statement  $i$ ,  $P_i$  and  $P_{i,\pi(j)}$  are connected either with the intersection or union operators for polytopes:  $\forall j \in \{2, \dots, n\}: P_i = P_i \uplus P_{i,\pi(j)}$  with

$$\uplus = \begin{cases} \cap & \text{if } C_{i,\pi(j-1)} \ \&\& \ C_{i,\pi(j)} \\ \cup & \text{if } C_{i,\pi(j-1)} \ || \ C_{i,\pi(j)} \end{cases}$$

$P_i$  models those ranges of the index variables where one *if*-statement is satisfied. Since all *if*-statements need to be satisfied, the global search space  $G$  is built by intersecting all  $P_i$ :  $G = \cap P_i$ . Since polyhedra are not closed under the union operator, the  $P_i$  defined above are no real polytopes. Instead, we use finite unions of polytopes for which the union and intersection operators are closed [23].

### 3.4. Global search space exploration

Since all  $P_i$  are finite unions of polytopes, the global search space  $G$  also is a finite union of polytopes. Each polytope of  $G$  defines a region where all *if*-statements in a loop nest are satisfied. After the construction of  $G$ , appropriate regions of  $G$  have to be selected so that once again the total number of executed *if*-statements is minimized after loop nest splitting.

Since unions of polytopes (i.e. logical *OR* of constraints) cannot be modeled using ILP, a second GA is used here. For a given global search space  $G = R_1 \cup R_2 \cup \dots \cup R_M$ , each individual  $I$  consists of a bit-vector where bit  $I_r$  determines whether region  $R_r$  of  $G$  is selected or not:  $I = (I_1, I_2, \dots, I_M)$  with  $I_r = 1 \Leftrightarrow$  region  $R_r$  is selected.

$$\begin{aligned}
 IF_I &= 0; \\
 \forall i_1 \in [lb_1, ub_1] \\
 &\dots \\
 \forall i_\lambda \in [lb_\lambda, ub_\lambda] \\
 IF_I &= IF_I + 1; \\
 &\text{if } (G_I = \text{true for } (i_1, \dots, i_\lambda)) \\
 &\quad i_\lambda = ub_\lambda; \\
 &\text{else} \\
 IF_I &= IF_I + IF_{\lambda+1};
 \end{aligned}$$

Figure 17-3. Global *If*-statement counter.

#### Definition 4:

1. For an individual  $I$ ,  $G_I$  is the global search space  $G$  reduced to those regions selected by  $I$ :  $G_I = R_r$  with  $I_r = 1$
2. The innermost loop  $\lambda$  is the index of the loop where the loop nest has to be split when considering  $G_I$ :  $\lambda = \max\{l | L_l \in \Lambda, i_l \text{ is used in } G_I\}$
3.  $\nu_l$  denotes the number of *if*-statements located in the body of loop  $L_l$  but not in any other loop  $L'_l$  nested in  $L_l$ . For Figure 17-1,  $\nu_3$  is equal to 2, all other  $\nu_l$  are zero.
4.  $IF_I$  denotes the number of executed *if*-statements when the loop nest  $\Lambda' = \{L_1, \dots, L_N\}$  would be executed:

$$\begin{aligned}
 IF_I &= (ub_I - lb_I + 1) \times (IF_{I+1} + \nu_I) \\
 IF_{N+1} &= 0
 \end{aligned}$$

The fitness of an individual  $I$  represents the number  $IF_I$  of *if*-statements that are executed when splitting  $\Lambda$  using the regions  $R_r$  selected by  $I$ .  $IF_I$  is incremented by one for every execution of the splitting-*if*. If the splitting-*if* is true, the counter remains unchanged. If not,  $IF_I$  is incremented by the number of executed original *if*-statements (see Figure 17-3).

After the GA has terminated, the innermost loop  $\lambda$  of the best individual defines where to insert the splitting-*if*. The regions  $R_r$  selected by this indi-

vidual serve for the generation of the conditions of the splitting-if and lead to the minimization of *if*-statement executions.

#### 4. BENCHMARKING RESULTS

All techniques presented here are fully implemented using the SUIF [24], Polylib [23] and PGAPack [14] libraries. Both GA's use the default parameters provided by [14] (population size 100, replacement fraction 50%, 1,000 iterations). Our tool was applied to three multimedia programs. First, a cavity detector for medical imaging (*CAV* [3]) having passed the DTSE methodology [9] is used. We apply loop nest splitting to this transformed application for showing that we are able to remove the overhead introduced by DTSE without undoing the effects of DTSE. The other benchmarks are the MPEG4 full search motion estimation (*ME* [8], see Figure 17-1) and the QSDPCM algorithm [22] for scene adaptive coding.

Since all polyhedral operations [23] have exponential worst case complexity, loop nest splitting also is exponential overall. Yet, the effective runtimes of our tool are very low, between 0.41 (QSDPCM) and 1.58 (CAV) CPU seconds are required for optimization on an AMD Athlon (1.3 GHz). For obtaining the results presented in the following, the benchmarks are compiled and executed before and after loop nest splitting. Compilers are always invoked with all optimizations enabled so that highly optimized code is generated.

Section 4.1 illustrates the impacts of loop nest splitting on CPU pipeline and cache behavior. Section 4.2 shows how the runtimes of the benchmarks are affected by loop nest splitting for ten different processors. Section 4.3 shows in how far code sizes increase and demonstrates that our techniques are able to reduce the energy consumption of the benchmarks considerably.

##### 4.1. Pipeline and Cache Behavior

Figure 17-4 shows the effects of loop nest splitting observed on an Intel Pentium III, Sun UltraSPARC III and a MIPS R10000 processor. All CPUs have a single main memory but separate level 1 instruction and data caches. The off-chip L2 cache is a unified cache for both data and instructions in all cases.

To obtain these results, the benchmarks were compiled and executed on the processors while monitoring performance-measuring counters available in the CPU hardware. This way, reliable values are generated without using erroneous cache simulators. The figure shows the performance values for the optimized benchmarks as a percentage of the un-optimized versions denoted as 100%.

The columns `Branch Taken` and `Pipe stalls` show that we are able to generate a more regular control flow for all benchmarks. The number of taken

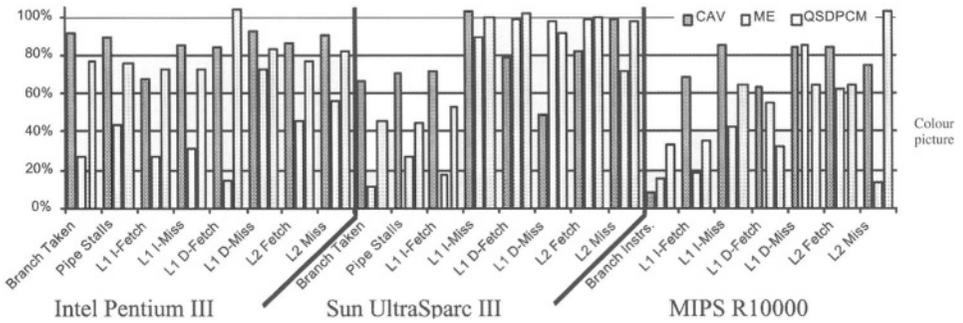


Figure 17-4. Pipeline and Cache Behavior after Loop Nest Splitting.

branch instructions is reduced between 8.1% (CAV Pentium) and 88.3% (ME Sun) thus leading to similar reductions of pipeline stalls (10.4%–73.1%). For the MIPS, a reduction of executed branch instructions by 66.3% (QSDPCM) – 91.8% (CAV) was observed. The very high gains for the Sun CPU are due to its complex 14-stage pipeline which is very sensitive to stalls.

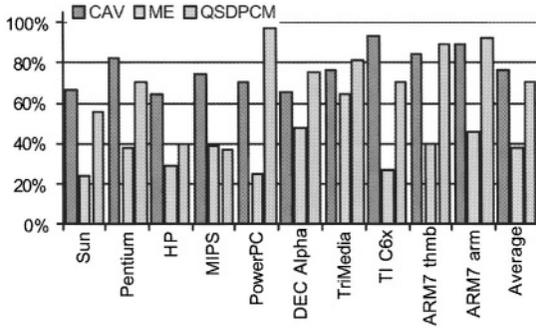
The results clearly show that the L1 I-cache performance is improved significantly. I-fetches are reduced by 26.7% (QSDPCM Pentium) – 82.7% (ME Sun), and I-cache misses are decreased largely for Pentium and MIPS (14.7%–68.5%). Almost no changes were observed for the Sun. Due to less index variable accesses, the L1 D-caches also benefit. D-cache fetches are reduced by 1.7% (ME Sun) – 85.4% (ME Pentium); only for QSDPCM, D-fetches increase by 3.9% due to spill code insertion. D-cache misses drop by 2.9% (ME Sun) – 51.4% (CAV Sun). The very large register file of the Sun CPU (160 integer registers) is the reason for the slight improvements of the L1 D-cache behavior for ME and QSDPCM. Since these programs only use very few local variables, they can be stored entirely in registers even before loop nest splitting.

Furthermore, the columns L2 Fetch and L2 Miss show that the unified L2 caches also benefit significantly, since reductions of accesses (0.2%–53.8%) and misses (1.1%–86.9%) are reported in most cases.

#### 4.2. Execution Times

All in all, the factors mentioned above lead to speed-ups between 17.5% (CAV Pentium) and 75.8% (ME Sun) for the processors considered in the previous section (see Figure 17-5). To demonstrate that these improvements not only occur on these CPUs, additional runtime measurements were performed for an HP-9000, PowerPC G3, DEC Alpha, TriMedia TM-1000, TI C6x and an ARM7TDMI, the latter both in 16-bit thumb- and 32-bit arm-mode.

Figure 17-5 shows that all CPUs benefit from loop nest splitting. CAV is sped up by 7.7% (TI) – 35.7% (HP) with mean improvements of 23.6%. Since loop nest splitting generates very regular control flow for ME, huge gains



Colour picture

Figure 17-5. Runtimes after Loop Nest Splitting.

appear here. ME is accelerated by 62.1% on average with minimum and maximum speed-ups of 36.5% (TriMedia) and 75.8% (Sun). For QSDPCM, the improvements range from 3% (PowerPC) – 63.4% (MIPS) (average: 29.3%).

Variations among different CPUs depend on several factors. As already stated previously, the layout of register files and pipelines are important parameters. Also, different compiler optimizations and register allocators influence the runtimes. Due to lack of space, a detailed study can not be given here.

### 4.3. Code Sizes and Energy Consumption

Since code is replicated, loop nest splitting entails larger code sizes (see Figure 17-6a). On average, the CAV benchmark’s code size increases by 60.9%, ranging from 34.7% (MIPS) – 82.8% (DEC). Though ME is sped up most, its code enlarges least. Increases by 9.2% (MIPS) – 51.4% (HP) lead to a mean growth of only 28%. Finally, the code of QSDPCM enlarges between 8.7% (MIPS) – 101.6% (TI) leading to an average increase of 61.6%.

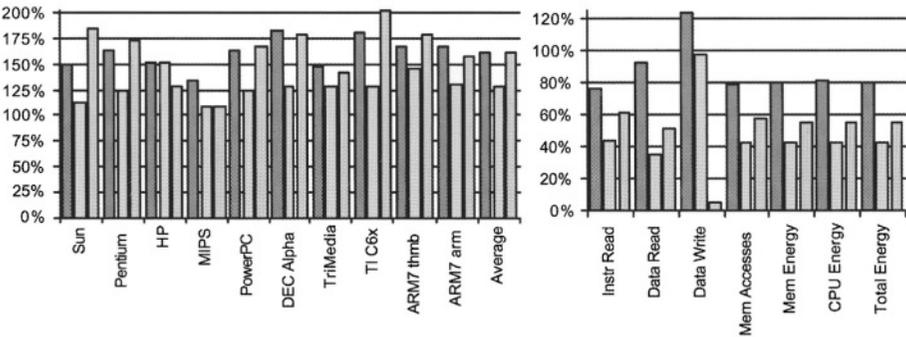


Figure 17-6. a) Code Sizes b) Energy Consumption after Loop Nest Splitting.

These increases are not serious, since the added energy required for storing these instructions is compensated by the savings achieved by loop nest splitting. Figure 17-6b shows the evolution of memory accesses and energy consumption using an instruction-level energy model [21] for the ARM7 core considering bit-toggles and offchip-memories and having an accuracy of 1.7%.

Column `Instr Read` shows reductions of instruction memory reads of 23.5% (CAV) – 56.9% (ME). Moreover, our optimization reduces data memory accesses significantly. Data reads are reduced up to 65.3% (ME). For QSDPCM, the removal of spill code reduces data writes by 95.4%. In contrast, the insertion of spill code for CAV leads to an increase of 24.5%. The sum of all memory accesses (`Mem accesses`) drops by 20.8% (CAV) – 57.2% (ME).

Our optimization leads to large energy savings in both the CPU and its memory. The energy consumed by the ARM core is reduced by 18.4% (CAV) – 57.4% (ME), the memory consumes between 19.6% and 57.7% less energy. Total energy savings by 19.6% – 57.7% are measured. These results demonstrate that loop nest splitting optimizes the locality of instructions and data accesses simultaneously as desired by Kim, Irwin et al. [11]

Anyhow, if code size increases (up to a rough theoretical bound of 100%) are critical, our algorithms can be changed so that the splitting-if is placed in some inner loop. This way, code duplication is reduced at the expense of lower speed-ups, so that trade-offs between code size and speed-up can be realized.

## 5. CONCLUSIONS

We present a novel source code optimization called loop nest splitting which removes redundancies in the control flow of embedded multimedia applications. Using polytope models, code without effect on the control flow is removed. Genetic algorithms identify ranges of the iteration space where all *if*-statements are provably satisfied. The source code of an application is rewritten in such a way that the total number of executed *if*-statements is minimized.

A careful study of 3 benchmarks shows significant improvements of branching and pipeline behavior. Also, caches benefit from our optimization since I- and D-cache misses are reduced heavily (up to 68.5%). Since accesses to instruction and data memory are reduced largely, loop nest splitting thus leads to large power savings (19.6%–57.7%). An extended benchmarking using 10 different CPUs shows mean speed-ups of the benchmarks by 23.6%–62.1%.

The selection of benchmarks used in this article illustrates that our optimization is a general and powerful technique. Not only typical real-life embedded code is improved, but also negative effects of other source code transformations introducing control flow overheads into an application are eliminated. In the future, our analytical models are extended for treating more

classes of loop nests. Especially, support for loops with variable bounds will be developed.

## REFERENCES

1. D. F. Bacon, S. L. Graham and O. J. Sharp. "Compiler Transformations for High-Performance Computing." *ACM Computing Surveys*, Vol. 26, p. 4, December 1994.
2. T. Bäck. *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, Oxford, 1996.
3. M. Bister, Y. Taeymans and J. Cornelis. "Automatic Segmentation of Cardiac MR Images." *IEEE Journal on Computers in Cardiology*, 1989.
4. R. Drechsler. *Evolutionary Algorithms for VLSI CAD*. Kluwer Academic Publishers, Boston, 1998.
5. H. Falk. *Control Flow Optimization by Loop Nest Splitting at the Source Code Level*. Research Report 773, University of Dortmund, Germany, 2002.
6. H. Falk and P. Marwedel. "Control Flow driven Splitting of Loop Nests at the Source Code Level." In *Design, Automation and Test in Europe (DATE)*, Munich, 2003.
7. H. Falk, C. Ghez, M. Miranda and Rainer Leupers. "High-level Control Flow Transformations for Performance Improvement of Address-Dominated Multimedia Applications." In *Synthesis and System Integration of Mixed Technologies (SASIMI)*, Hiroshima, 2003.
8. S. Gupta, M. Miranda et al. "Analysis of High-level Address Code Transformations for Programmable Processors." In *Design, Automation and Test in Europe (DATE)*, Paris, 2000.
9. Y. H. Hu (ed.). *Data Transfer and Storage (DTS) Architecture Issues and Exploration in Multimedia Processors*, Vol. Programmable Digital Signal Processors – Architecture, Programming and Applications. Marcel Dekker Inc., New York, 2001.
10. M. Kandemir, N. Vijaykrishnan et al. "Influence of Compiler Optimizations on System Power." In *Design Automation Conference (DAC)*, Los Angeles, 2000.
11. H. S. Kim, M. J. Irwin et al. "Effect of Compiler Optimizations on Memory Energy." In *Signal Processing Systems (SIPS)*. Lafayette, 2000.
12. R. Leupers. *Code Optimization Techniques for Embedded Processors – Methods, Algorithms and Tools*. Kluwer Academic Publishers, Boston, 2000.
13. R. Leupers and F. David. "A Uniform Optimization Technique for Offset Assignment Problems." In *International Symposium on System Synthesis (ISSS)*, Hsinchu, 1998.
14. D. Levine. *Users Guide to the PGAPack Parallel Genetic Algorithm Library*. Technical Report ANL-95/18, Argonne National Laboratory, 1996.
15. N. Liveris, N. D. Zervas et al. "A Code Transformation-Based Methodology for Improving I-Cache Performance of DSP Applications." In *Design, Automation and Test in Europe (DATE)*, Paris, 2002.
16. M. Lorenz, R. Leupers et al. "Low-Energy DSP Code Generation Using a Genetic Algorithm." In *International Conference on Computer Design (ICCD)*, Austin, 2001.
17. T. S. Motzkin, H. Raiffa et al. "The Double Description Method." *Theodore S. Motzkin: Selected Papers*, 1953.
18. S. S. Muchnick. "Optimizing Compilers for SPARC." *SunTechnology*, Vol. 1, p. 3. 1988.
19. S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, 1997.
20. M. R. Stan and W. P. Bursleson. "Bus-Invert Coding for Low-Power I/O." *IEEE Transactions on VLSI Systems*, Vol. 3, p. 1, 1995.
21. S. Steinke, M. Knauer, L. Wehmeyer and P. Marwedel. "An Accurate and Fine Grain Instruction-Level Energy Model Supporting Software Optimizations." In *Power and Timing Modeling, Optimization and Simulation (PATMOS)*, Yverdon-Les-Bains, 2001.

22. P. Strobach. "A New Technique in Scene Adaptive Coding." In *European Signal Processing Conference (EUSIPCO)*, Grenoble, 1988.
23. D. K. Wilde. *A Library for doing polyhedral Operations*. Technical Report 785, IRISA Rennes, France, 1993.
24. R. Wilson, R. French et al. *An Overview of the SUIF Compiler System*. <http://suif.stanford.edu/suif/suif1>, 1995.
25. S. Wuytack, F. Catthoor et al. "Power Exploration for Data Dominated Video Applications." In *International Symposium on Low Power Electronics and Design (ISLPED)*, Monterey, 1996.

*This page intentionally left blank*

## Chapter 18

# DATA SPACE ORIENTED SCHEDULING

M. Kandemir<sup>1</sup>, G. Chen<sup>1</sup>, W. Zhang<sup>1</sup> and I. Kolcu<sup>2</sup>

<sup>1</sup> *CSE Department, The Pennsylvania State University, University Park, PA 16802, USA;*

<sup>2</sup> *Computation Department, UMIST, Manchester M60 1QD, UK*

*E-mail: {kandemir,gchen,wzhang}@cse.psu.edu,i.kolcu@umist.ac.uk*

**Abstract.** With the widespread use of embedded devices such as PDAs, printers, game machines, cellular telephones, achieving high performance demands an optimized operating system (OS) that can take full advantage of the underlying hardware components. We present a locality conscious process scheduling strategy for embedded environments. The objective of our scheduling strategy is to maximize reuse in the data cache. It achieves this by restructuring the process codes based on data sharing patterns between processes. Our experimentation with five large array-intensive embedded applications demonstrate the effectiveness of our strategy.

**Key words:** Embedded OS, process scheduling, data locality

## 1. INTRODUCTION

As embedded designs become more complex, so does the process of embedded software development. In particular, with the added sophistication of Operating System (OS) based designs, developers require strong system support. A variety of sophisticated techniques maybe required to analyze and optimize embedded applications. One of the most striking differences between traditional process schedulers used in general purpose operating systems and those used in embedded operating systems is that it is possible to customize the scheduler in the latter [7]. In other words, by taking into account the specific characteristics of the application(s), we can have a scheduler tailored to the needs of the workload.

While previous research used compiler support (e.g., loop and data transformations for array-intensive applications) and OS optimizations (e.g., different process scheduling strategies) in an isolated manner, in an embedded system, we can achieve better results by considering the interaction between the OS and the compiler. For example, the compiler can analyze the application code statically (i.e., at compile time) and pass some useful information to the OS scheduler so that it can achieve a better performance at runtime. This work is a step in this direction. We use compiler to analyze the process codes and determine the portions of each process that will be executed in each time quanta in a pre-emptive scheduler. This can help the scheduler in circumstances where the compiler can derive information from that code that

would not be easily obtained during execution. While one might think of different types of information that can be passed from the compiler to the OS, in this work, we focus on information that helps improve data cache performance. Specifically, we use the compiler to analyze and restructure the process codes so that the data reuse in the cache is maximized.

Such a strategy is expected to be viable in the embedded environments where process codes are extracted from the same application and have significant data reuse between them. In many cases, it is more preferable to code an embedded application as a set of co-operating processes (instead of a large monolithic code). This is because in general such a coding style leads to better modularity and maintainability. Such light-weight processes are often called threads. In array-intensive embedded applications (e.g., such as those found in embedded image and video processing domains), we can expect a large degree of data sharing between processes extracted from the same application. Previous work on process scheduling in the embedded systems area include works targeting instruction and data caches. A careful mapping of the process codes to memory can help reduce the conflict misses in instruction caches significantly. We refer the reader to [8] and [4] for elegant process mapping strategies that target instruction caches. Li and Wolfe [5] present a model for estimating the performance of multiple processes sharing a cache.

Section 1.2 presents the details of our strategy. Section 1.3 presents our benchmarks and experimental setting, and discusses the preliminary results obtained using a set of five embedded applications. Section 1.4 presents our concluding remarks.

## 2. OUR APPROACH

Our process scheduling algorithm takes a data space oriented approach. The basic idea is to restructure the process codes to improve data cache locality. Let us first focus on a single array; that is, let us assume that each process manipulates the same array. We will relax this constraint shortly. We first *logically* divide the array in question into tiles (these tiles are called *data tiles*). Then, these data tiles are visited one-by-one (in some order), and we determine for each process a set of iterations (called *iteration tile*) that will be executed in each of its quanta. This approach will obviously increase data reuse in the cache (as in their corresponding quanta the processes manipulate the same set of array elements). The important questions here are how to divide the array into data tiles, in which order the tiles should be visited, and how to determine the iterations to execute (for each process in each quanta). In the following discussion, after presenting a simple example illustrating the overall idea, we explain our approach to these three issues in detail.

Figure 18-1 shows a simple case (for illustrative purposes) where three processes access the same two-dimensional array (shown in Figure 18-1 (a)). The array is divided into four data tiles. The first and the third processes

have two nests while the second process has only one nest. In Figure 18-1(b), we show for each nest (of each process) how the array is accessed. Each outer square in Figure 18-1(b) corresponds to an iteration space and the shadings in each region of an iteration space indicate the array portion accessed by that region. Each different shading corresponds to an iteration tile, i.e., the set of iterations that will be executed when the corresponding data tile is being processed. That is, an iteration tile access the data tile(s) with the same type of shading. For example, we see that while the nest in process II accesses all portions of the array, the second nest in process III accesses only half of the array. Our approach visits each data tile in turn, and at each step executes (for each process) the iterations that access that tile (each step here corresponds to three quanta). This is depicted in Figure 18-1(c). On the other hand, a straightforward (pre-emptive) process scheduling strategy that does not pay attention to data locality might obtain the schedule illustrated in Figure 18-1(d), where at each time quanta each process executes the half of the iteration space of a nest (assuming all nests have the same number of iterations). Note that the iterations executed in a given quanta in this schedule do not reuse the data elements accessed by the iterations executed

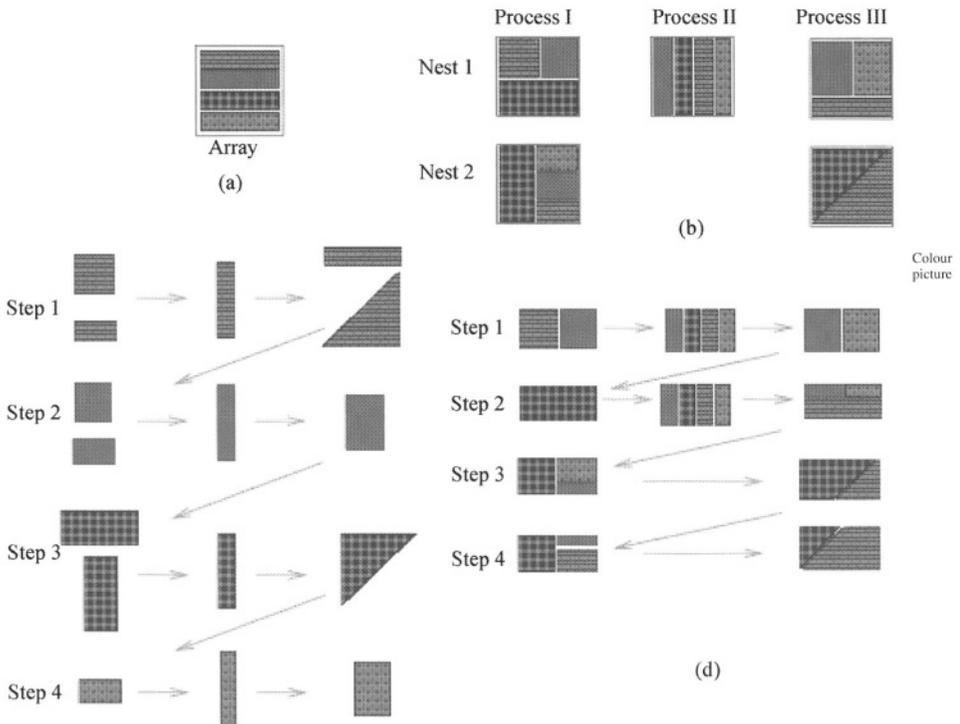


Figure 18-1. (a) An array divided into four portions (data tiles). (b) Access pattern exhibited. (c) Scheduling steps for our approach. (d) Steps of an alternative scheduling.

in the previous quanta. Consequently, we can expect that data references would be very costly due to frequent off-chip memory accesses. When we compare the access patterns in Figures 18-1(c) and (d), we clearly see that the one in Figure 18-1(c) is expected to result in a much better data locality. When we have multiple arrays accessed by the processes in the system, we need to be careful in choosing the array around which the computation is restructured. In this section, we present a possible array selection strategy as well.

### 2.1. Selecting data tile shape/size

The size of a data tile (combined with the amount of computation that will be performed for each element of it) determines the amount of work that will be performed in a quanta. Therefore, selecting a small tile size tends to cause frequent process switchings (and incur the corresponding performance overhead), while working with large tile sizes can incur many cache misses (if the cache does not capture locality well). Consequently, our objective is to select the largest tile size that does not overflow the cache. The tile shape, on the other hand, is strongly dependent on two factors: data dependences and data reuse. Since we want to execute all iterations that manipulate the data tile elements in a single quanta, there should not be any circular dependence between the two iteration tiles belonging to the same process. In other words, all dependences between two iteration tiles should flow from one of them to the other. Figure 18-2 shows a legal (iteration space) tiling in (a) and an illegal tiling in (b). The second factor that influences the selection of a data tile shape is the degree of reuse between the elements that map on the same tile. More specifically, if, in manipulating the elements of a given data tile, we make frequent accesses to array elements outside the said tile, this means that we do not have good intra-tile locality. An ideal data tile shape must be such that the iterations in the corresponding iteration tile should access only the elements in the corresponding data tile.

Data tiles in  $m$ -dimensional data (array) space can be defined by  $m$  families of parallel hyperplanes, each of which is an  $(m-1)$  dimensional hyperplane. The tiles so defined are parallelepipeds (except for the ones that reside on the boundaries of data space). Note that each data tile is an  $m$ -dimensional subset of the data space. Let  $\mathcal{M}$  be an  $m$ -by- $m$  nonsingular matrix whose each



Colour picture

Figure 18-2. (a) Legal iteration space tiling. (b) Illegal tiling. Each node denotes an iteration point and each group of nodes is an iteration tile. The arrows between iteration points indicate data dependences.

row is a vector perpendicular to a tile boundary.  $\mathcal{M}$  matrix is referred to as the *data tile matrix*. It is known that  $\mathcal{M}^{-1}$  is matrix, each column of which gives the direction vector for a tile boundary (i.e., its columns define the shape of the data tile).

Let  $\mathcal{F}$  be a matrix that contains the vectors that define the relations between array elements. More specifically, if, during the same iteration, two array elements  $\vec{a}_1$  and  $\vec{a}_2$  are accessed together, then  $\vec{a}_2 - \vec{a}_1$  is a column in  $\mathcal{F}$  (assuming that  $\vec{a}_2$  is lexicographically greater than  $\vec{a}_1$ ). It can be proven (but not done so here due to lack of space) that the non-zero elements in  $\mathcal{M}\mathcal{F}$  correspond to the tile-crossing edges in  $\mathcal{F}$ . So, a good  $\mathcal{M}$  should maximize the number of zero elements in  $\mathcal{M}\mathcal{F}$ . The entire iteration space can also be tiled in a similar way that the data space is tiled. An iteration space tile shape can be expressed using a square matrix  $\mathcal{H}$  (called the *iteration tile matrix*), each row of which is perpendicular to one plane of the (iteration) tile. As in the data tile case, each column of  $\mathcal{H}^{-1}$  gives a direction vector for a tile boundary. It is known that, in order for an iteration tile to be legal (i.e., dependence-preserving),  $\mathcal{H}$  should be chosen such that *none* of the entries in  $\mathcal{H}\mathcal{D}$  is negative, where  $\mathcal{D}$  is the dependence matrix. For clarity, we write this constraint as  $\mathcal{H}\mathcal{D} \geq 0$ .

Based on the discussion above, our tile selection problem can be formulated as follows. Given a loop nest, select an  $\mathcal{M}$  such that the corresponding  $\mathcal{H}$  does not result in circular data dependences and that  $\mathcal{M}\mathcal{F}$  has the minimum number of non-zero elements. More technically, assuming that  $\mathcal{G}$  is the access (reference) matrix,<sup>1</sup> we have  $\mathcal{M} = \mathcal{G}\mathcal{H}$ . Assuming that  $\mathcal{G}$  is invertible (if not, pseudo-inverses can be used), we can obtain  $\mathcal{H} = \mathcal{G}^{-1}\mathcal{M}$ . Therefore, the condition that needs to be satisfied is  $\mathcal{G}^{-1}\mathcal{M} \geq 0$ . That is, we need to select an  $\mathcal{M}$  matrix such that the number of non-zero entries in  $\mathcal{M}\mathcal{F}$  is minimum and  $\mathcal{G}^{-1}\mathcal{M} \geq 0$ .

As an example, let us consider the nest in Figure 18-3(a). Since there are three references to array  $A$ , we have three columns in  $\mathcal{F}$  (one between each

$$\begin{aligned} &\text{for } i = 3 \dots N \\ &\quad \text{for } j = 2 \dots N \\ &\quad\quad B[i][j] = A[i-2][j-1] + A[i-1][j-1] + A[i][j] \end{aligned} \tag{a}$$

$$\begin{aligned} &\text{for } i = 3 \dots N \\ &\quad \text{for } j = 2 \dots N \\ &\quad\quad A[i][j] = A[i-2][j-1] + A[i-1][j-1] + B[i][j] \end{aligned} \tag{b}$$

*Figure 18-3.* Two example nests. Note that while both the nests have similar data access patterns, the second nest also exhibits data dependences. Consequently, selecting a suitable data tile matrix/iteration tile matrix for the second case is more difficult.

pair of distinct references). Obtaining the differences between the subscript expressions, we have

$$\mathcal{F} = \begin{pmatrix} 1 & 1 & 2 \\ 0 & 1 & 1 \end{pmatrix}$$

Since there are no data dependences in the code, we can select any data tile matrix  $\mathcal{M}$  that minimizes the number of non-zero entries in  $\mathcal{MF}$ . Assuming

$$\mathcal{M} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

we have

$$\mathcal{MF} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} 1 & 1 & 2 \\ 0 & 1 & 1 \end{pmatrix} = \begin{pmatrix} a & a+b & 2a+b \\ c & c+d & 2c+d \end{pmatrix}$$

Recall that our objective is to minimize the number of non-zero entries in  $\mathcal{MF}$ . One possible choice is  $a = 0$ ,  $b = 1$ ,  $c = 1$ , and  $d = -1$ , which gives

$$\mathcal{M} = \begin{pmatrix} -1 & 1 \\ 0 & -1 \end{pmatrix}$$

Another alternative is  $a = -1$ ,  $b = 1$ ,  $c = 0$ , and  $d = -1$ , which results in

$$\mathcal{M} = \begin{pmatrix} -1 & 1 \\ 1 & -1 \end{pmatrix}$$

Note that in both the cases we zero out two entries in  $\mathcal{MF}$ .

Now, let us consider the nested loop in Figure 18-3(b). As far as array  $A$  is concerned, while the access pattern exhibited by this nest is similar to the one above, this nest also contains data dependences (as array  $A$  is both updated and read). That is,

$$\mathcal{F} = \begin{pmatrix} 1 & 1 & 2 \\ 0 & 1 & 1 \end{pmatrix} \quad \text{and} \quad \mathcal{D} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$

Where  $\mathcal{D}$  is the data dependence matrix. Consequently, we need to select an  $\mathcal{M}$  such that

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \geq 0$$

In other words, we need to satisfy  $b \geq 0$ ,  $d \geq 0$ ,  $a + b \geq 0$ , and  $c + d \geq 0$ . Considering the two possible solutions given above, we see that while  $a = 0$ ,  $b = 1$ ,  $c = 1$ , and  $d = -1$  satisfy these inequalities  $a = -1$ ,  $b = 1$ ,  $c = 0$ , and  $d = -1$  do not satisfy them. That is, the data dependences restrict our flexibility in choosing the entries of the data tile matrix  $\mathcal{M}$ .

*Multiple references to the same array:* If there are multiple references to a given array, we proceed as follows. We first group the references such that if two references are uniformly references, they are placed into the same group. Two references  $\mathcal{G}_1\vec{l} + \vec{g}_1$  and  $\mathcal{G}_2\vec{l} + \vec{g}_2$  are said to be uniformly generated if

and only if  $\mathcal{G}_1 = \mathcal{G}_2$ . For example,  $A[i + 1][j - 1]$  and  $A[i][j]$  are uniformly generated, whereas  $A[i][j]$  and  $A[j][i]$  are not. Then, we count the number of references in each uniformly generated reference group, and the access matrix of the group with the highest count is considered as the representative reference of this array in the nest in question. This is a viable approach because of the following reason. In many array-intensive benchmarks, most of the references (of a given array) in a given nest are uniformly generated. This is particularly true for array-intensive image and video applications where most computations are of stencil type. Note that in our example nests in Figure 18-3, each array has a single uniformly generated reference set.

*Multiple arrays in a nest:* If we have more than one array in the code, the process of selecting suitable data tile shapes becomes more complex. It should be noted that our approach explained above is a data space centric one; that is, we reorder the computation according to the data tile access pattern. Consequently, if we have two arrays in the code, we can end up with two different execution orders. Clearly, one of these orders might be preferable over the other. Let us assume, for the clarity of presentation, that we have two arrays,  $A$  and  $B$ , in a given nest. Our objective is to determine two data tile matrices  $\mathcal{M}_A$  (for array  $A$ ) and  $\mathcal{M}_B$  (for array  $B$ ) such that the total number of non-zero elements in  $\mathcal{M}_A \mathcal{F}_A$  and  $\mathcal{M}_B \mathcal{F}_B$  is minimized. Obviously, data dependences in the code need also be satisfied (i.e.,  $\mathcal{H}\mathcal{D} = 0$ , where  $\mathcal{H}$  is the iteration tile matrix). We can approach this problem in two ways. Let us first focus on  $\mathcal{M}_A$ . If we select a suitable  $\mathcal{M}_A$  so that the number of non-zero elements in  $\mathcal{M}_A \mathcal{F}_A$  is minimized, we can determine an  $\mathcal{H}$  for the nest in question using this  $\mathcal{M}_A$  and the access matrix  $\mathcal{G}_A$ . More specifically,  $\mathcal{H} = \mathcal{G}_A^{-1} \mathcal{M}_A$  (assuming that  $\mathcal{G}_A$  is invertible). After that, using this  $\mathcal{H}$ , we can find an  $\mathcal{M}_B$  from  $\mathcal{M}_B = \mathcal{G}_B \mathcal{H}$ . An alternative way would start with  $\mathcal{M}_B$ , then determine  $\mathcal{H}$ , and after that, determine  $\mathcal{M}_A$ . One way of deciding which of these strategies is better than the other is to look at the number of zero (or non-zero) entries in the resulting  $\mathcal{M}_A \mathcal{F}_A$  and  $\mathcal{M}_B \mathcal{F}_B$  matrices. Obviously, in both the cases, if there are data dependences, the condition  $\mathcal{H}\mathcal{D} \geq 0$  needs also be satisfied.

*Multiple nests in a process code:* Each nest can have a different iteration tile shape for the same data tile, and a legal iteration tile (for a given data tile) should be chosen for each process separately. As an example, let us focus on a process code that contains two nests (that access the same array). Our approach proceeds as follows. If there are no dependences, we first find a  $\mathcal{M}$  such that the number of non-zero entries in  $\mathcal{M}\mathcal{F}_1$  and  $\mathcal{M}\mathcal{F}_2$  is minimized. Here,  $\mathcal{F}_1$  and  $\mathcal{F}_2$  are the matrices that capture the relations between array elements in the first nest and the second nest, respectively. Then, using this  $\mathcal{M}$ , we can determine  $\mathcal{H}_1$  and  $\mathcal{H}_2$  from  $\mathcal{H}_1 = \mathcal{G}_1^{-1} \mathcal{M}$  and  $\mathcal{H}_2 = \mathcal{G}_2^{-1} \mathcal{M}$ , respectively. In these expressions,  $\mathcal{G}_1$  and  $\mathcal{G}_2$  are the access matrices in the first and the second nest, respectively. It should be noted, however, that the nests in a given process do not need to use the same data tile matrix  $\mathcal{M}$  matrix for the same array. In other words, it is possible (and in some cases actually benefi-

cial) for each nest to select a different  $\mathcal{M}$  depending on its  $\mathcal{F}$  matrix. This is possible because our data space tiling is a logical concept; that is, we do not physically divide a given array into tiles or change the storage order of its elements in memory. However, in our context, it is more meaningful to work with a single  $\mathcal{M}$  matrix for all nests in a process code. This is because when a data tile is visited (during scheduling), we would like to execute *all* iteration tiles from *all* nests (that access the said tile). This can be achieved more easily if we work with a single data tile shape (thus, single  $\mathcal{M}$ ) for the array throughout the computation.

There is an important point that we need to clarify before proceeding further. As mentioned earlier, our approach uses the  $\mathcal{F}$  matrix to determine the data tile shape to use. If we have only a single nest, we can build this  $\mathcal{F}$  matrix considering each pair of data references to the array. If we consider multiple nests simultaneously, on the other hand, i.e., try to select a single data tile matrix  $\mathcal{M}$  for the array for the entire program, we need to have an  $\mathcal{F}$  matrix that reflects the combined affect of all data accesses. While it might be possible to develop sophisticated heuristics to obtain such a global  $\mathcal{F}$  matrix, in this work, we obtain this matrix by simply combining the columns of individual  $\mathcal{F}$  matrices (coming from different nests). We believe that this is a reasonable strategy given the fact that most data reuse occurs within the nests rather than between the nests.

## 2.2. Tile traversal order

Data tiles should be visited in an order that is acceptable from the perspective of data dependences [3]. Since in selecting the iteration tiles (based on the data tiles selected) above we eliminate the possibility of circular dependences between iteration tiles, we know for sure that there exists at least one way of traversing the data tiles that lead to legal code. In finding such an order, we use a strategy similar to classical list scheduling that is frequently used in compilers from industry and academia. Specifically, given a set of data tiles (and an ordering between them), we iteratively select a data tile at each step. We start with a tile whose corresponding iteration tile does not have any incoming data dependence edges. After scheduling it, we look the remaining tiles and select a new one. Note that scheduling a data tile might make some other data tiles scheduleable. This process continues until all data tiles are visited. This is a viable approach as in general the number of data tiles for a given array is small (especially, when the tiles are large).

## 2.3. Restructuring process codes and code generation

It should be noted that the discussion in the last two subsections is a bit simplified. The reason is that we assumed that the array access matrices are invertible (i.e., nonsingular). In reality, most nests have different access matrices and some access matrices are not invertible. Consequently, we need

to find a different mechanism for generation iteration tiles from data tiles. To achieve this, we employ a polyhedral tool called the Omega Library [2]. The Omega library consists of a set of routines for manipulating linear constraints over Omega sets which can include integer variables, Presburger formulas, and integer tuple relations and sets. We can represent iteration spaces, data spaces, access matrices, data tiles, and iteration tiles using Omega sets. For example, using this library, we can generate iterations that access the elements in a given portion of the array.

## 2.4. Overall algorithm

Based on the discussion in the previous subsections, we present the sketch of our overall algorithm in Figure 18-4. In this algorithm, in the for-loop between lines 2 and 14, we determine the array around which we need to restructure the process codes. Informally, we need to select an array such that the number of non-zero entries in  $\mathcal{M}_s \mathcal{F}_s$  should be minimum, where  $1 \leq s \leq L$ ,  $L$  being the total number of arrays in the application. To achieve this, we try each array in turn. Note that this portion of our algorithm works on the entire application code. Next, in line 15, we determine a schedule order for the processes. In this work, we do not propose a specific algorithm to achieve this; however,

```

1.  max-count ← 0
2.  for each array  $A_i$  in the application to
3.    determine  $\mathcal{F}_i$ 
4.    determine  $\mathcal{M}_i$ 
5.    determine  $\mathcal{F}$  for each nest
6.    for each array  $A_j$  where  $j \neq i$  do
7.      determine  $\mathcal{M}_j$ 
8.      count ← the number of non-zero elements in  $\sum_i^L \mathcal{M}_i \mathcal{F}_i$ 
9.      if count > max-count do
10.         k ← 1
11.         max-count = count
12.       endif
13.     endfor
14.  endfor
15.  determine a schedule order for the processes
16.  restructure (tile) each process code considering  $\mathcal{M}_k$ 
    using the Omega Library)
17.  if there are leftover iterations do
18.    select another array  $A_{k'}$  such that  $k' \neq k$ 
19.    drop  $A_k$  from consideration and goto 6
20.  endif

```

Figure 18-4. Overall scheduling algorithm.

several heuristics can be used. In line 16, we restructure the process codes (i.e., tile them).

### 3. EXPERIMENTS

#### 3.1. Implementation and experimental methodology

We evaluate our data space oriented process scheduling strategy using a simple in-house simulator. This simulator takes as input a program (represented as a process dependence graph), a scheduling order, a data cache configuration (associativity, cache size, and line size), and cache simulator as parameters, and records the number of cache hits/misses in the data cache and execution cycles. For cache simulations, we used a modified version of Dinero III [6]. The code transformations are implemented using the SUIF compiler infrastructure from Stanford University [1] and the Omega Library [2].

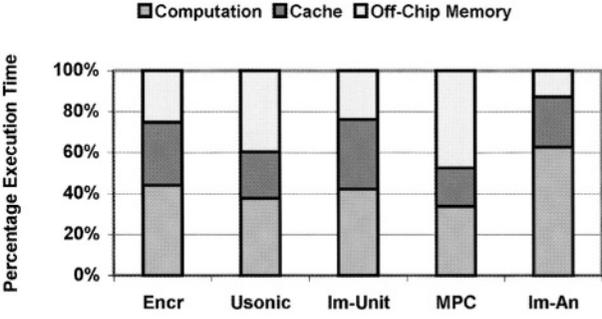
#### 3.2. Benchmarks

Our benchmark suite contains five large array-intensive embedded applications. *Encr* implements the core computation of an algorithm for digital signature for security. It has two modules, each with eleven processes. The first module generates a cryptographically-secure digital signature for each outgoing packet in a network architecture. User requests and packet send operations are implemented as processes. The second module checks the digital signature attached to an incoming message.

*Usonic* is a feature-based object estimation kernel. It stores a set of encoded objects. Given an image and a request, it extracts the potential objects of interest and compares them one-by-one to the objects stored in the database (which is also updated). *Im-Unit* is an image processing package that works with multiple image formats. It has built-in functions for unitary image transformations, noise reduction, and image morphology. *MPC* is a medical pattern classification application that is used for interpreting electrocardiograms and predicting survival rates. *Im-An* is an image analysis algorithm that is used for recognizing macromolecules.

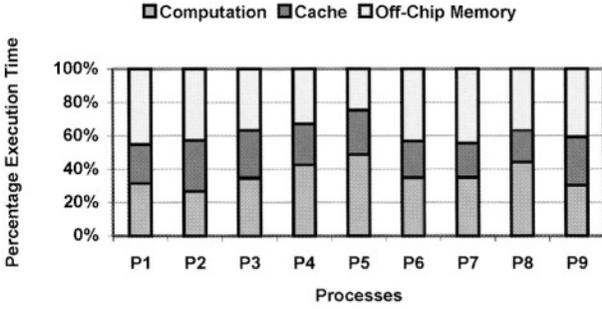
#### 3.3. Results

In Figure 18-5, we divide the execution time of each application into three sections: the time spent in CPU by executing instructions, the time spent in cache accesses, and the time spent in accesses to the off-chip memory. We clearly see from this graph that the off-chip memory accesses can consume significant number of memory cycles. To understand this better, we give in Figure 18-6 the execution time breakdown for the processes of the *MPC* benchmark. We observe that each process spends a significant portion of its



Colour picture

Figure 18-5. Execution time breakdown.



Colour picture

Figure 18-6. Execution time breakdown for the process of MPC.

execution cycles in waiting for off-chip memory accesses to complete. Since we know that there is significant amount of data reuse between successively scheduled processes of this application, the data presented by the graph in Figure 18-6 clearly indicates that the data cache is not utilized effectively. That is, the data left by a process in the data cache is not reused by the next process (which means poor data cache behavior).

To check whether this trend is consistent when time quanta is modified, we performed another set of experiments. The graph in Figure 18-7 shows the execution time breakdown for different time quanta. The values shown in this graph are the values averaged over all benchmarks in our experimental suite. One can see that although increasing quanta leads to better cache locality, the trends observed above still holds (in fact, memory stall cycles constitute nearly 25% of the total execution cycles with our largest quanta).

Figure 18-8 presents the execution time savings for all benchmarks with different cache capacities. These results indicate that even with large data cache sizes (e.g., 32 KB and 64 KB), our strategy brings significant execution time benefits. As expected, we perform better with small cache sizes. Considering the fact that data set sizes keep increasing, we expect that the results with small cache sizes indicate future trends better.

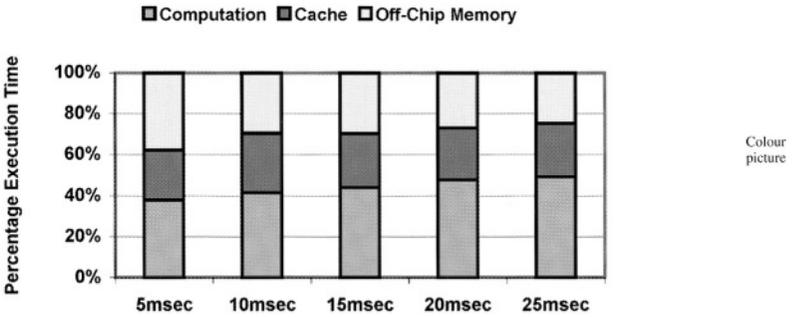


Figure 18-7. Execution time breakdown with different quanta length.

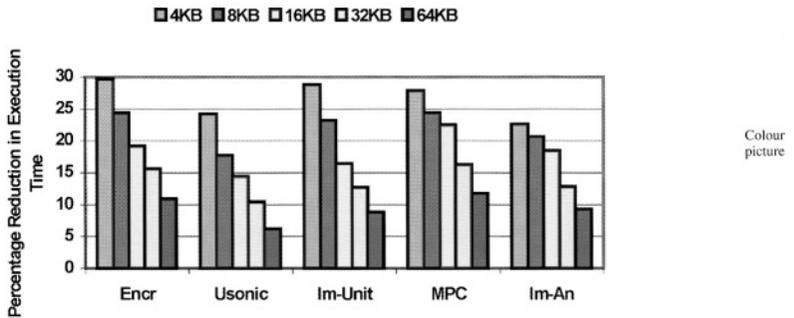


Figure 18-8. Percentage savings in execution time with different data cache capacities.

In the last set of experiments, we compared our strategy (denoted *DSOS*) in Figure 18-9) with four different scheduling strategies using our default configuration. *Method1* is a locality-oriented process scheduling strategy. It restructures process codes depending on the contents of the data cache. *Method2* is like *Method1* except that it also changes the original scheduling order in an attempt to determine the best process to schedule next (i.e., the

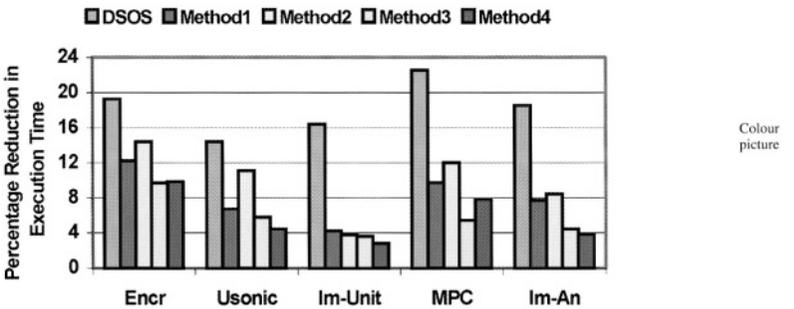


Figure 18-9. Percentage savings in execution time with different scheduling strategies.

one that will reuse the cache contents most). *Method3* does not restructure the process codes and but tries to reorder their schedule order to minimize cache conflicts. *Method4* is similar to the original (default) scheme except that the arrays are placed in memory so as to minimize the conflict misses. To obtain this version, we used extensive array padding. We see from these results that our strategy outperforms the remaining scheduling strategies for all benchmarks in our experimental suite. The reason for this is that in many cases it is not sufficient to just try to reuse the contents of the cache by considering the previously-scheduled process.

#### 4. CONCLUDING REMARKS

Process scheduling is a key issue in any multi-programmed system. We present a locality conscious scheduling strategy whose aim is to exploit data cache locality as much as possible. It achieves this by restructuring the process codes based on data sharing between processes. Our experimental results indicate that the scheduling strategy proposed brings significant performance benefits.

#### NOTE

1 A reference to an array can be represented by  $\mathbf{G}\vec{i} + \vec{g}$ , where  $\mathbf{G}$  is a linear transformation matrix called the array reference (access) matrix,  $\vec{g}$  is the offset (constant) vector;  $\vec{i}$  is a column vector, called iteration vector, whose elements written left to right represent the loop indices  $i_1, i_2, \dots, i_n$ , starting from the outermost loop to the innermost in the loop nest.

#### REFERENCES

1. S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and C. W. Tseng. "The SUIF Compiler for Scalable Parallel Machines." In *Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing*, February, 1995.
2. W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and David Wonnacott. "The Omega Library Interface Guide." *Technical Report CS-TR-3445*, CS Department, University of Maryland, College Park, MD, March 1995.
3. I. Kodukula, N. Ahmed, and K. Pingali. "Data-Centric Multi-Level Blocking." In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1997.
4. C-G. Lee et al. "Analysis of Cache Related Preemption Delay in Fixed-Priority Preemptive Scheduling." *IEEE Transactions on Computers*, Vol. 47, No. 6, June 1998.
5. Y. Li and W. Wolfe. "A Task-Level Hierarchical Memory Model for System Synthesis of Multiprocessors." *IEEE Transactions on CAD*, Vol. 18, No. 10, pp. 1405-1417, October 1999.
6. WARTS: Wisconsin Architectural Research Tool Set. <http://www.cs.wisc.edu/~larus/warts.html>
7. W. Wolfe. *Computers as Components: Principles of Embedded Computing System Design*, Morgan Kaufmann Publishers, 2001.
8. A. Wolfe. "Software-Based Cache Partitioning for Real-Time Applications." In *Proceedings of the Third International Workshop on Responsive Computer Systems*, September 1993.

*This page intentionally left blank*

# COMPILER-DIRECTED ILP EXTRACTION FOR CLUSTERED VLIW/EPIC MACHINES

Satish Pillai and Margarida F. Jacome

*The University of Texas at Austin, USA*

**Abstract.** Compiler-directed ILP extraction techniques are critical to effectively exploiting the significant processing capacity of contemporaneous VLIW/EPIC machines. In this paper we propose a novel algorithm for *ILP extraction* targeting *clustered* EPIC machines that integrates three powerful techniques: predication, speculation and modulo scheduling. In addition, our framework schedules and binds operations to clusters, generating actual VLIW code. A key innovation in our approach is the ability to perform resource constrained speculation in the context of a complex (phased) optimization process. Specifically, in order to enable maximum utilization of the resources of the clustered processor, and thus maximize performance, our algorithm judiciously speculates operations on the predicated modulo scheduled loop body using a set of effective load based metrics. Our experimental results show that by jointly considering different extraction techniques in a resource aware context, the proposed algorithm can take maximum advantage of the resources available on the clustered machine, aggressively improving the initiation interval of time critical loops.

**Key words:** ILP extraction, clustered processors, EPIC, VLIW, predication, modulo scheduling, speculation, optimization

## 1. INTRODUCTION

Multimedia, communications and security applications exhibit a significant amount of instruction level parallelism (ILP). In order to meet the performance requirements of these demanding applications, it is important to use compilation techniques that expose/extract such ILP and processor datapaths with a large number of functional units, e.g., VLIW/EPIC processors.

A basic VLIW datapath might be based on a *single* register file shared by all of its functional units (FUs). Unfortunately, this simple organization does not scale well with the number of FUs [12]. *Clustered* VLIW datapaths address this poor scaling by restricting the connectivity between FUs and registers, so that an FU on a cluster can only read/write from/to the cluster's register file, see e.g. [12]. Since data may need to be transferred among the machine's clusters, possibly resulting in increased latency, it is important to develop performance enhancing techniques that take such data transfers into consideration.

Most of the applications alluded to above have only a few time critical

kernels, i.e. a small fraction of the entire code (sometimes as small as 3%) is executed most of the time, see e.g. [16] for an analysis of the MediaBench [5] suite of programs. Moreover, most of the processing time is typically spent executing the 2 innermost loops of such time critical loop nests – in [16], for example, this percentage was found to be about 95% for MediaBench programs. Yet another critical observation made in [16] is that there exists considerably high control complexity within these loops. This strongly suggests that in order to be effective, ILP extraction targeting such time critical inner loop bodies must handle control/branching constructs.

The key contribution of this paper is a novel resource-aware algorithm for compiler-directed *ILP extraction* targeting *clustered* EPIC machines that integrates three powerful ILP extraction techniques: predication, control speculation and software pipelining/modulo scheduling. An important innovation in our approach is the ability to perform resource constrained speculation in the context of a complex (phased) optimization process. Specifically, in order to enable maximum utilization of the resources of the clustered processor, and thus maximize performance, our algorithm judiciously speculates operations on the predicated modulo scheduled loop body using a set of effective load based metrics. In addition to extracting ILP from time-critical loops, our framework schedules and binds the resulting operations, generating actual VLIW code.

## 1.1. Background

The performance of a loop is defined by the average rate at which new loop iterations can be started, denoted initiation interval (II). Software pipelining is an ILP extraction technique that retimes [20] loop body operations (i.e., overlaps multiple loop iterations), so as to enable the generation of more compact schedules [3, 18]. Modulo scheduling algorithms exploit such technique during scheduling, so as to expose additional ILP to the datapath resources, and thus decrease the initiation interval of a loop [23].

Predication allows one to concurrently schedule alternative paths of execution, with only the paths corresponding to the realized flow of control being allowed to actually modify the state of the processor. The key idea in predication is to eliminate branches through a process called *if-conversion* [8]. If-conversion, transforms conditional branches into (1) operations that define predicates, and (2) operations guarded by predicates, corresponding to alternative control paths.<sup>1</sup> A guarded operation is committed only if its predicate is true. In this sense, if-conversion is said to convert control dependences into data dependences (i.e., dependence on predicate values), generating what is called a *hyperblock* [11].

Control speculation “breaks” the *control dependence* between an operation and the conditional statement it is dependent on [6]. By eliminating such dependences, operations can be moved out of conditional branches, and can be executed before their related conditionals are actually evaluated. Compilers

exploit control speculation to reduce control dependence height, which enables the generation of more compact, higher ILP static schedules.

## 1.2. Overview

The proposed algorithm iterates through three main phases: *speculation*, *cluster binding* and *modulo scheduling* (see Figure 19-4). At each iteration, relying on effective load balancing heuristics, the algorithm incrementally extracts/exposes additional (“profitable”) ILP, i.e., speculates loop body operations that are more likely to lead to a decrease in initiation interval during modulo scheduling. The resulting loop operations are then assigned to the machine’s clusters and, finally, the loop is modulo scheduled generating actual VLIW code. An additional critical phase (executed right after binding) attempts to leverage required data transfers across clusters to realize the speculative code, thus minimizing their potential impact on performance (see details in Section 4).

To the best of our knowledge, there is no other algorithm in the literature on compiler-directed predicated code optimizations that jointly considers control speculation and modulo scheduling in the context of clustered EPIC machines. In the experiments section we will show that, by jointly considering speculation and modulo scheduling in a resource aware context, and by minimizing the impact in performance of data transfers across clusters, the algorithm proposed in this paper can dramatically reduce a loop’s initiation interval with upto 68.2% improvement with respect to a baseline algorithm representative of the state of the art.

## 2. OPTIMIZING SPECULATION

In this Section, we discuss the process of deciding which loop operations to speculate so as to achieve a more effective utilization of datapath resources and improve performance (initiation interval). The algorithm for optimizing speculation, described in detail in Section 3, uses the concept of load profiles to estimate the distribution of load over the scheduling ranges of operations, as well as across the clusters of the target VLIW/EPIC processor. Section 2.1 explains how these load profiles are calculated. Section 2.2 discusses how the process of speculation alters these profiles and introduces the key ideas exploited by our algorithm.

### 2.1. Load profile calculation

The load profile is a measure of resource requirements needed to execute a Control Data Flow Graph (CDFG) with a desirable schedule latency. Note that, when predicated code is considered, the branching constructs actually correspond to the definition of a predicate (denoted *PD* in Figure 19-1), and

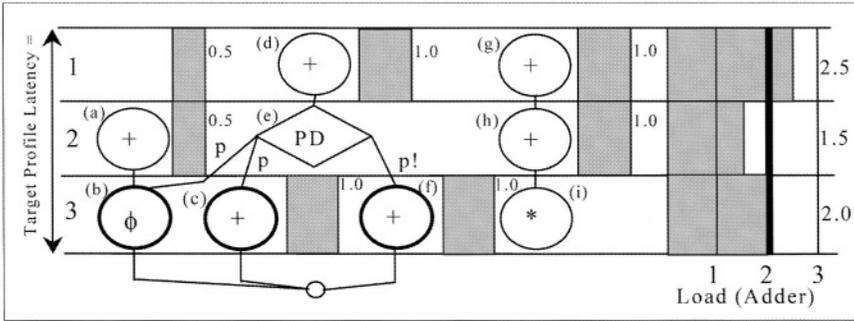


Figure 19-1. Adder load profile calculation.

the associated control dependence corresponds to a data dependence on the predicate values (denoted  $p$  and  $p'$ ). Accordingly, the operations shown in bold in Figure 19-1, Figure 19-2 and Figure 19-3 show operations that are predicated on the predicate values (either  $p$  or  $p'$ ).

Also note that speculation modifies the original CDFG. Consider, for example, the CDFG shown in Figure 19-2(a) and its modified version shown in Figure 19-3(a) obtained by speculating an addition operation (labeled  $a$ ) using the SSA-PS transformation [9]. As can be seen in Figure 19-3(a), the data dependence of that addition operation to predicate  $p$  was eliminated and a  $\phi$  operation, reconciling the renamed variable to its original value, was added. For more details on the SSA-PS transformation, see [9].

To illustrate how the load profiles used in the optimization process are calculated, consider again the CDFG in Figure 19-1. Note that the load profile is calculated for a given target profile latency (greater than the critical path of the CDFG). First, As Soon As Possible (ASAP) and As Late As Possible (ALAP) scheduling is performed to determine the scheduling range of each operation. For the example, operation  $a$  in Figure 19-1 has a scheduling range of 2 steps (i.e. step 1 and step 2), while all other operations have a scheduling range of a single step. The mobility of an operation is defined as

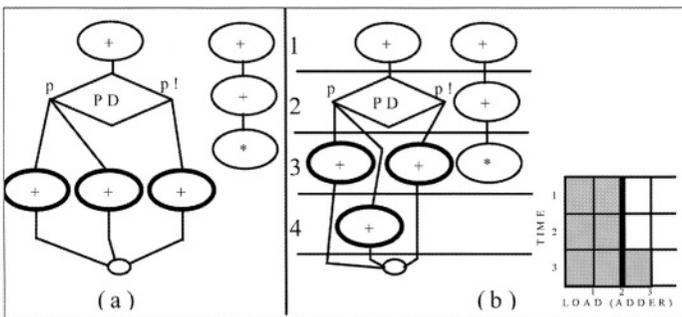


Figure 19-2. Predicated control flow graph (a) and schedule with no speculated operations (b).

$\mu(op) = alap(op) - asap(op) + 1$  and equals 2 for operation  $a$ . Assuming that the probability of scheduling an operation at any time step in its time frame is given by a uniform distribution [22], the contribution to the load of an operation  $op$  at time step  $t$  in its time frame is given by  $1/\mu(op)$ . In Figure 19-1, the contributions to the load of all the addition operations (labeled  $a, c, d, f, g$  and  $h$ ) are indicated by the shaded region to the right of the operations. To obtain the total load for type  $fu$  at a time step  $t$ , we sum the contribution to the load of all operations that are executable on resource type  $fu$  at time step  $t$ . In Figure 19-1, the resulting total *adder* load profile is shown.

The thick vertical line in the load profile plot is an indicator of the capacity of the machine’s datapath to execute instructions at a particular time step. (In the example, we assume a datapath that contains 2 adders.) Accordingly, in step 1 of the load profile, the shaded region to the right of the thick vertical line represents an over-subscription of the adder datapath resource. This indicates that, in the actual VLIW code/schedule, one of the addition operations (i.e. either operation  $a, d$  or  $g$ ) will need to be delayed to a later time step.

### 2.2. Load balancing through speculation

We argue that, in the context of VLIW/EPIC machines, the goal of compiler transformations aimed at speculating operations should be to “redistribute/balance” the load profile of the original/input kernel so as to enable a more effective utilization of the resources available in the datapath. More specifically, the goal should be to judiciously modify the scheduling ranges of the kernel’s operations, via speculation, such that overall resource contention is decreased/minimized, and consequently, code performance improved. We illustrate this key point using the example CDFG in Figure 2(a), and assuming a datapath with 2 adders, 2 multipliers and 1 comparator. Figure 19-2(b) shows the *adder* load profile for the original kernel while Figure 19-3(a) and Figure 19-3(b) show the load profiles for the resulting CDFG’s with one and three

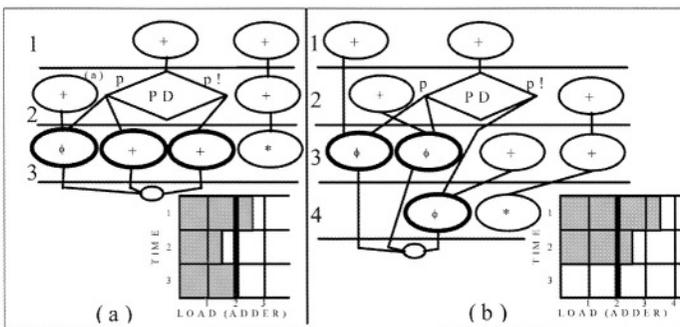


Figure 19-3. Schedule with 1 speculated operation (a) and schedule with 3 speculated operations (b).

(all) operations speculated, respectively [22].<sup>2</sup> As can be seen, the load profile in Figure 19-3(a) has a smaller area above the line representing the datapath’s resource capacity, i.e., implements a better redistribution of load and, as a result, allowed for a better schedule under resource constraints (only 3 steps) to be derived for the CDFG. From the above discussion, we see that a technique to judiciously speculate operations is required, in order to ensure that speculation provides consistent performance gains on a given target machine. Accordingly, we propose an optimization phase, called OPT-PH (described in Section 3), which given an input hyperblock and a target VLIW/EPIC processor, possibly clustered, decides which operations should be speculated, so as to maximize the execution performance of the resulting VLIW code.

### 3. OPT-PH – AN ALGORITHM FOR OPTIMIZING SPECULATION

Our algorithm makes incremental decisions on speculating individual operations, using a heuristic ordering of nodes. The suitability of an operation for speculation is evaluated based on two metrics, Total Excess Load (*TEL*) and Speculative Mobility (*Spec- $\mu$* ), to be discussed below. Such metrics rely on a previously defined binding function (assigning operations to clusters), and on a target profile latency (see Section 2.1).

#### 3.1. Total Excess Load (TEL)

In order to compute the first component of our ranking function, i.e. Total Excess Load (*TEL*), we start by calculating the load distribution profile for each cluster  $c$  and resource type  $fu$ , at each time step  $t$  of the target profile latency alluded to above. This is denoted by  $Clust\_Load_{fu, c}(t)$ . To obtain the cluster load for type  $fu$ , we sum the contribution to the load of all operations bound to cluster  $c$  (denoted by  $nodes\_to\_clust(c)$ ) that are executable on resource type  $fu$  (denoted by  $nodes\_on\_typ(fu)$ ) at time step  $t$ .

$$Clust\_Load_{fu, c}(t) \doteq \sum_{\forall op \in S | t \in if(op)} \frac{1}{\mu(op)}$$

where  $S = nodes\_to\_clust(c) \cap nodes\_on\_typ(fu)$

Formally:

$Clust\_Load_{fu, c}(t)$  is represented by the shaded regions in the load profiles in Figure 19-2 and Figure 19-3.

Recall that our approach attempts to “flatten” out the load distribution of operations by moving those operations that contribute to greatest excess load to earlier time steps. To characterize Excess Load (*EL*), we take the difference between the actual cluster load, given above, and an ideal load, i.e.,

$$Diff_{fu, c}(t) = Clust\_Load_{fu, c}(t) - Ideal\_Load_{fu, c}$$

The ideal load, denoted by  $Ideal\_Load_{fu, c}$ , is a measure of the balance in load necessary to efficiently distribute operations both over their time frames as well as across different clusters. It is given by,

$$Ideal\_Load_{fu, c} = \max\{Avg\_Load_{fu, c}, Clust\_Capacity_{fu, c}\}$$

where  $Clust\_Capacity_{fu, c}$  is the number of resources of type  $fu$  in cluster  $c$  and  $Avg\_Load_{fu, c}$  is the average cluster load over the target profile latency  $pr$ , i.e.,

$$Avg\_Load_{fu, c} = \frac{1}{pr} \sum_{t=1}^{pr} Clust\_Load_{fu, c}(t)$$

As shown above, if the resulting average load is smaller than the actual cluster capacity, the ideal load value is upgraded to the value of the cluster capacity. This is performed because load unbalancing per se is not necessarily a problem unless it leads to over-subscription of cluster resources. The average load and cluster capacity in the example of Figure 19-2 and Figure 19-3 are equal and, hence, the ideal load is given by the thick vertical line in the load profiles. The Excess Load associated with operation  $op$ ,  $EL(op)$ , can now be computed, as the difference between the actual cluster load and the ideal load over the time frame of the operation, with negative excess loads being set to zero, i.e.,

$$EL(op) = \sum_{t=atop(op)}^{atop(op)} \max\{0, Diff_{typ(op), clust(op)}(t)\}$$

where operation  $op$  is bound to cluster  $clust(op)$  and executes on resource type  $typ(op)$ . In the load profiles of Figure 19-2 and Figure 19-3, excess load is represented by the shaded area to the right of the thick vertical line. Clearly, operations with high excess loads are good candidates for speculation, since such speculation would reduce resource contention at their current time frames, and may thus lead to performance improvements. Thus,  $EL$  is a good indicator of the relative suitability of an operation for speculation.

However, speculation may be also beneficial when there is no resource over-subscription, since it may reduce the CDFG's critical path. By itself,  $EL$  would overlook such opportunities. To account for such instances, we define a second suitability measure, which "looks ahead" for empty scheduling time slots that could be occupied by speculated operations. Accordingly, we define Reverse Excess Load ( $REL$ ) to characterize availability of free resources at earlier time steps to execute speculated operations:

$$REL(op) = \sum_{t=1}^{asap(op)-1} \min\{0, Diff_{typ(op), clust(op)}(t)\}$$

This is shown by the unshaded regions to the left of the thick vertical line in the load profiles of Figure 19-2 and Figure 19-3.

We sum both these quantities and divide by the operation mobility to obtain Total Excess Load per scheduling time step.

$$TEL(op) = \frac{EL(op) + REL(op)}{\mu(op)}$$

### 3.2. Speculative mobility( $Spec_{\mu}$ )

The second component of our ranking function, denoted  $Spec_{\mu}(op)$ , is an indicator of the number of additional time steps made available to the operation through speculation. It is given by the difference between the maximum ALAP value over all predecessors of the operation, denoted by  $pred(op)$ , before and after speculation. Formally:

$$Spec_{\mu}(op) = \max_{\forall n \in pred(op)} alap(n) - \max_{\forall n \in pred(op)|n \neq cond.op}$$

### 3.3. Ranking function

The composite ordering of operations by suitability for speculation, called  $Suit(op)$ , is given by:

$$Suit(op) = TEL(op) + Spec_{\mu}(op)$$

$Suit(op)$  is computed for each operation that is a candidate for speculation, and speculation is attempted on the operation with the highest value, as discussed in the sequel.

## 4. OPTIMIZATION FRAMEWORK

Figure 19-4 shows the complete iterative optimization flow of OPT-PH. During the initialization phase, if-conversion is applied to the original CDFG – control dependences are converted to data dependences by defining the appropriate predicate define operations and data dependence edges, and an initial binding is performed using a modified version of the algorithm presented in [15]. Specifically, the initial phase of this binding algorithm was implemented in our framework along with some changes specific to the SSA-PS transformation [9], as outlined below. The original algorithm adds a penalty for every predecessor of an operation that is bound to a cluster different from the cluster to which the operation itself is being tentatively bound. In our version of the algorithm, such penalty is not added if the predecessor is a predicated move operation. The reason for this modification is the fact that predicated move operations realizing the reconciliation (i.e.  $\phi$  [9]) functions may themselves be used to perform required (i.e. binding related) data transfers across clusters and thus may actually not lead to a performance penalty.

After the initialization phase is performed, the algorithm enters an itera-

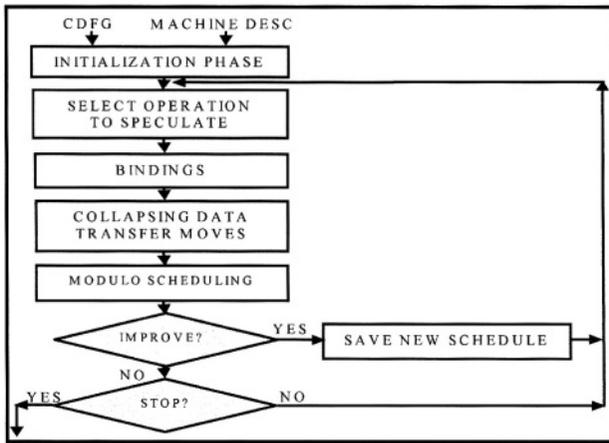


Figure 19-4. Overview of optimization flow.

tive phase. First it decides on the next best candidate for speculation. The ranking function used during this phase has already been described in detail in Section 3. The simplest form of speculating the selected operation is by deleting the edge from its predecessor predicate define operation (denoted *predicate promotion* [6]). In certain cases, the ability to speculate requires renaming and creating a new successor predicated move operation for reconciliation (denoted SSA-PS [9]).

After speculation is done, binding is performed using the binding algorithm described above. The next optimization phase applies the critical transformation of collapsing binding related move operations with reconciliation related predicated moves, see [9]. Finally a modulo scheduler schedules the resulting Data Flow Graph (DFG). A two level priority function that ranks operations first by lower alap and next by lower mobility is used by the modulo scheduler.

If execution latency is improved with respect to the previous best result, then the corresponding schedule is saved. Each new iteration produces a different binding function that considers the modified scheduling ranges resulting from the operation speculated in the previous iteration. The process continues iteratively, greedily speculating operations, until the termination condition is satisfied. Currently this condition is simply a threshold on the number of successfully speculated operations, yet more sophisticated termination conditions can very easily be included.

Since the estimation of cluster loads as well as the binding algorithm depend on the assumed profile latency, we found experimentally that it was important to search over different such profile latencies. Thus, the iterative process is repeated for various profile latency values, starting from the ASAP latency of the original CDFG and incrementing it upto a given percentage of the critical path (not exceeding four steps).

## 5. PREVIOUS WORK

We discuss below work that has been performed in the area of modulo scheduling and speculation. The method presented in [19] uses control speculation to decrease the initiation interval of modulo scheduled loops in control-intensive non-numeric programs. However, since this method is not geared towards clustered architectures, it does not consider load balancing and data transfers across clusters. A modulo scheduling scheme is proposed in [24] for a specialized clustered VLIW micro-architecture with distributed cache memory. This method minimizes inter-cluster *memory* communications and is thus geared towards the particular specialized memory architecture proposed. It also does not explicitly consider conditionals within loops. The software pipelining algorithm proposed in [26] generates a near-optimal modulo schedule for all iteration paths along with efficient code to transition between paths. The time complexity of this method is, however, exponential in the number of conditionals in the loop. It may also lead to code explosion.

A state of the art static, compiler-directed ILP extraction technique that is particularly relevant to the work presented in this paper is standard (if-conversion based) predication with speculation in the form of predicate promotion [6] (see Section 4), and will be directly contrasted to our work.

A number of techniques have been proposed in the area of high level synthesis for performing speculation. However, some of the fundamental assumptions underlying such techniques do not apply to the code generation problem addressed in this paper, for the following reasons. First, the cost functions used for hardware synthesis (e.g. [10, 13]), aiming at minimizing control, multiplexing and interconnect costs, are significantly different from those used by a software compiler, where schedule length is usually the most important cost function to optimize. Second, most papers (e.g. [4, 14, 21, 27, 28]) in the high level synthesis area exploit conditional resource sharing. Unfortunately, this cannot be exploited/accommodated in the actual VLIW/EPIC code generation process, because predicate values are unknown at the time of instruction issue. In other words, two micro-instructions cannot be statically bound to the same functional unit at the same time step, even if their predicates are known to be mutually exclusive, since the actual predicate values become available only after the *execute stage* of the predicate defining instruction, and the result (i.e. the predicate value) is usually forwarded to the write back stage for squashing, if necessary. Speculative execution is incorporated in the framework of a list scheduler by [7]. Although the longest path speculation heuristic proposed is effective, it does not consider load balancing, which is important for clustered architectures. A high-level synthesis technique that increases the density of optimal scheduling solutions in the search space and reduces schedule length is proposed in [25]. Speculation performed here, however, does not involve renaming of variable names and so code motion is comparatively restricted. Also there is no merging of control paths performed, as done by predication.

Certain special purpose architectures, like transport triggered architectures, as proposed in [1], are primarily programmed by scheduling data transports, rather than the CFG’s operations themselves. Code generation for such architectures is fundamentally different, and harder than code generation for the standard VLIW/EPIC processors assumed in this paper, see [2].

## 6. EXPERIMENTAL RESULTS

Compiler algorithms reported in the literature either jointly address speculation and modulo scheduling but do not consider clustered machines, or consider modulo scheduling on clustered machines but cannot handle conditionals i.e., can only address straight code, with no notion of control speculation (see Section 5). Thus, the novelty of our approach makes it difficult to experimentally validate our results in comparison to previous work.

We have however devised an experiment that allowed us to assess two of the key contributions of this paper, namely, the proposed load-balancing-driven incremental control speculation and the phasing for the complete optimization problem. Specifically, we compared the code generated by our algorithm to code generated by a baseline algorithm that binds state of the art predicated code (with predicate promotion only) [6] to a clustered machine and then modulo schedules the code. In order to ensure fairness, the baseline algorithm uses the same binding and modulo scheduling algorithms implemented in our framework.

We present experimental results for a representative set of critical loop kernels extracted from the MediaBench [5] suite of programs and from TI’s benchmark suite [17] (see Table 19-1). The frequency of execution of the

Table 19-1. Kernel characteristics.

Kernel	Benchmark	Main Inner Loop from Function
Lsqsolve	Rasta/Lsqsolve	eliminate()
Csc	Mpeg	csc()
Shortterm	Gsm	fast_Short_term_synthesis_filtering()
Store	Mpeg2dec	conv422to444()
Ford	Rasta	FORD1()
Jquant	Jpeg	find_nearby_colors()
Huffman	Epic	encode_stream()
Add	Gsm	gsm_div()
Pixel1	Mesa	gl_write_zoomed_index_span()
Pixel2	Mesa	gl_write_zoomed_stencil_span()
Intra	Mpeg2enc	iquant1_intra()
Nonintra	Mpeg2enc	iquant1_non_intra()
Vdthresh8	TI suite	
Collision	TI suite	
Viterbi	TI suite	

selected kernels for typical input data sets was determined to be significant. For example, the kernel `Lsqsolve` from the least squares solver in the `rasta` distribution, one of the kernels where OPT-PH performs well, takes more than 48% of total time taken by the solver. Similarly, `Jquant`, the function `find_nearby_colors()` from the `Jpeg` program, yet another kernel where OPT-PH performs well, takes more than 12% of total program execution time.

The test kernels were manually compiled to a 3-address like intermediate representation that captured all dependences between instructions. This intermediate representation together with a parameterized description of a clustered VLIW datapath was input to our automated optimization tool (see Figure 19-4). Four different clustered VLIW datapath configurations were used for the experiments. All the configurations were chosen to have relatively small clusters since these datapath configurations are more challenging to compile to, as more data transfers may need to be scheduled. The FU's in each cluster include Adders, Multipliers, Comparators, Load/Store Units and Move Ports. The datapaths are specified by the number of clusters followed by the number of FU's of each type, respecting the order given above. So a configuration denoted 3C: 1111111111 specifies a datapath with three clusters, each cluster with 1 FU of each type. All operations are assumed to take 1 cycle except Read/Write, which take 2 cycles. A move operation over the bus takes 1 cycle. There are 2 intercluster buses available for data transfers.

Detailed results of our experiments (15 kernels compiled for 4 different datapath configurations) are given in Table 19-2. For every kernel and datapath configuration, we present the performance of modulo scheduled standard predicated code with predicate promotion (denoted `Std. Pred-MOD`) compared to that of code produced by our approach (denoted `OPT-PH-MOD`). As can be seen, our technique consistently produces shorter initiation intervals and gives large increases in performance, upto a maximum of 68.2%. This empirically demonstrates the effectiveness of the speculation criteria and load balancing scheme developed in the earlier Sections. The breaking of control dependences and efficient redistribution of operation load both over their time frames as well as across clusters, permits dramatic decreases in initiation interval.

Finally, although the results of our algorithm cannot be compared to results produced by high level synthesis approaches (as discussed in Section 5), we implemented a modified version of the list scheduling algorithm proposed in [7], that performed speculation “on the fly”, and compared it to our technique. Again, our algorithm provided consistently faster schedules with performance gains upto 57.58%.

## 7. CONCLUSIONS

The paper proposed a novel resource-aware algorithm for compiler-directed *ILP extraction* targeting *clustered EPIC* machines. In addition to extracting

Table 19-2. Initiation interval.

Kernel	3C:11111111			3C:12121111		
	Std. Pred.-MOD	OPT-PH-MOD	%Imp	Std. Pred.-MOD	OPT-PH-MOD	%Imp
Lsqsolve	4	3	25	3	3	0
Csc	12	12	0	12	12	0
Shortterm	5	4	20	4	4	0
Store	22	9	59.1	22	7	68.2
Ford	4	4	0	4	4	0
Jquant	31	28	9.7	31	26	16.1
Huffman	4	4	0	4	4	0
Add	2	2	0	2	2	0
Pixel1	5	4	20	5	4	20
Pixel2	4	3	25	4	3	25
Intra	8	6	25	8	6	25
Nonintra	9	7	22.2	6	6	0
Vdthresh8	4	4	0	4	4	0
Collision	7	7	0	7	7	0
Viterbi	24	10	58.3	8	8	0

Kernel	4C:11111111			4C:12121111		
	Std. Pred.-MOD	OPT-PH-MOD	%Imp	Std. Pred.-MOD	OPT-PH-MOD	%Imp
Lsqsolve	4	4	0	4	4	0
Csc	11	11	0	11	11	0
Shortterm	5	5	0	5	4	20
Store	8	7	12.5	7	7	0
Ford	12	9	25	12	9	25
Jquant	31	28	9.7	29	28	3.5
Huffman	4	4	0	4	4	0
Add	3	3	0	2	2	0
Pixel1	4	4	0	4	4	0
Pixel2	3	3	0	3	3	0
Intra	6	6	0	6	6	0
Nonintra	9	7	22.2	7	7	0
Vdthresh8	4	4	0	4	4	0
Collision	6	6	0	6	6	0
Viterbi	9	8	11.1	9	8	11.1

ILP from time-critical loops, our framework schedules and binds the resulting operations, generating actual VLIW code. Key contributions of our algorithm include: (1) an effective phase ordering for this complex optimization problem; (2) efficient load balancing heuristics to guide the optimization process; and (3) a technique that allows for maximum flexibility in speculating individual operations on a segment of predicated code.

## ACKNOWLEDGEMENTS

This work is supported by an NSF ITR Grant ACI-0081791 and an NSF Grant CCR-9901255.

## NOTES

<sup>1</sup> Note that operations that define predicates may also be guarded.

<sup>2</sup> Those load profiles were generated assuming a minimum target profile latency equal to the critical path of the kernel.

## REFERENCES

1. H. Corporaal. "TTAs: Missing the ILP Complexity Wall." *Journal of Systems Architecture*, 1999.
2. H. Corporaal and J. Hoogerbrugge. *Code Generation for Transport Triggered Architectures*, 1995.
3. K. Ebcioglu. "A Compilation Technique for Software Pipelining of Loops with Conditional Jumps." In *ISCA*, 1987.
4. C. J. Tseng et al. "Bridge: A Versatile Behavioral Synthesis System." In *DAC*, 1988.
5. C. Lee et al. "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems." In *MICRO*, 1997.
6. D. August et al. "Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture." In *ISCA*, 1998.
7. Ganesh Lakshminarayana et al. "Incorporating Speculative Execution into Scheduling of Control-Flow Intensive Behavioral Descriptions." In *DAC*, 1998.
8. J. R. Allen et al. "Conversion of Control Dependence to Data Dependence." In *POPL*, 1983.
9. M. Jacome et al. "Clustered VLIW Architectures with Predicated Switching." In *DAC*, 2001.
10. S. Gupta et al. "Speculation Techniques for High Level Synthesis of Control Intensive Designs." In *DAC*, 2001.
11. S. Mahlke et al. "Effective Compiler Support for Predicated Execution Using the Hyperblock." In *MICRO*, 1992.
12. S. Rixner et al. "Register Organization for Media Processing." In *HPCA*, 1999.
13. Sumit Gupta et al. "Conditional Speculation and its Effects on Performance and Area for High-Level Synthesis." In *ISSS*, 2001.
14. T. Kim et al. "A Scheduling Algorithm for Conditional Resource Sharing." In *ICCAD*, 1991.
15. V. Lapinskii et al. "High-Quality Operation Binding for Clustered VLIW Datapaths." In *DAC*, 2001.
16. J. Fritts. *Architecture and Compiler Design Issues in Programmable Media Processors*, Ph.D. Thesis, 2000.
17. <http://www.ti.com>.
18. Monica Lam. *A Systolic Array Optimizing Compiler*. Ph.D. Thesis, Carnegie Mellon University, 1987.
19. Daniel M. Lavery and Wen Mei W. Hwu. "Modulo Scheduling of Loops in Control-Intensive Non-Numeric Programs." In *ISCA*, 1996.
20. C. E. Leiserson and J. B. Saxe. "Retiming Synchronous Circuitry." In *Algorithmica*, 1991.
21. N. Park and A. C. Parker. "SEHWA: A Software Package for Synthesis of Pipelines from Behavioral Specifications." In *IEEE Transactions on CAD*, 1988.
22. P. G. Paulin and J. P. Knight. "Force-Directed Scheduling in Automatic Data Path Synthesis." In *DAC*, 1987.

23. B. Ramakrishna Rau. "Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops." In *ISCA*, 1994.
24. J. Sanchez and A. Gonzalez. "Modulo sScheduling for a Fully-Distributed Clustered VLIW Architecture." In *ISCA*, 2000.
25. L. Dos Santos and J. Jess. "A Reordering Technique for Efficient Code Motion." In *DAC*, 1999.
26. M. G. Stodeley and C. G. Lee. "Software Pipelining Loops with Conditional Branches." In *ISCA*, 1996.
27. K. Wakabayashi and H. Tanaka. "Global Scheduling Independent of Control Dependencies Based on Condition Vectors." In *DAC*, 1992.
28. P. F. Yeung and D. J. Rees. "Resources Restricted Global Scheduling." In *VLSI 1991*, 1991.

*This page intentionally left blank*

## Chapter 20

# STATE SPACE COMPRESSION IN HISTORY DRIVEN QUASI-STATIC SCHEDULING

Antonio G. Lomeña<sup>1</sup>, Marisa López-Vallejo<sup>1</sup>, Yosinori Watanabe<sup>2</sup> and Alex Kondratyev<sup>2</sup>

<sup>1</sup> *Electronic Engineering Department. ETSI Telecomunicación, Universidad Politécnica de Madrid, Ciudad Universitaria s/n, 28040, Madrid, Spain, E-mail:*

*{lomena,marisa}@die.upm.es;* <sup>2</sup> *Cadence Berkeley Laboratories, 2001 Addison Street, 3rd Floor, Berkeley, CA 94704, USA, E-mail: {watanabe,kalex}@cadence.com*

**Abstract.** This paper presents efficient compression techniques to avoid the state space explosion problem during quasi-static task scheduling of embedded, reactive systems.

Our application domain is targeted to one-processor software synthesis, and the scheduling process is based on Petri net reachability analysis to ensure cyclic, bounded and live programs. We suggest two complementary compression techniques that effectively reduce the size of the generated schedule and make the problem tractable for large specifications.

Our experimental results reveal a significant reduction in algorithmic complexity (both in memory storage and CPU time) obtained for medium and large size problems.

**Key words:** hash table, state explosion, quasi-static scheduling

## 1. INTRODUCTION

During the last years, the use of design methodologies based on formal methods has been encouraged as a means to tackle the increasing complexity in the design of electronic systems. However, traditional formal verification methods such as model checking or reachability analysis have the drawback of requiring huge computing resources.

In this paper we address the problem of software synthesis for embedded, reactive systems, using Petri nets (PNs) as our underlying formal model and reachability analysis as a way to formally obtain a valid quasi-static task schedule that ensures cyclic, buffer bounded, live programs [2]. To overcome the state space explosion problem that is inherent to the reachability analysis, we suggest two complementary techniques for schedule compression. The first technique explores the reachability space at a coarser granularity level by doing scheduling in big moves that combine the firing of several transitions at once. The second technique uses a dynamic, history based criterion that prunes the state space of uninteresting states for our application domain.

The paper is organized as follows. Next section reviews previous work related to the reduction of the state space explosion. Section 3 states the

problem definition. In section 4 we present techniques that identify promising sequences of transitions to be fired as a single instance during scheduling. Section 5 describes procedures to eliminate repetitions of markings included in the resulting schedule, and presents the results obtained for several experiments. Section 6 concludes the paper.

## 2. RELATED WORK

The detection of symmetries as a method to mitigate the state explosion problem has been previously studied in [8]. This approach has the drawback of its computational complexity, which is exponential with respect to the graph size.

The reduction theory tries to overcome this problem by performing a controlled simplification of the system specification. In [7] a series of transformations that preserve liveness, safety and boundedness in ordinary PNs are introduced. However, the situations modeled by these rules can be considered somewhat simple. In section 4 we show a powerful approach that combines behavioural and structural information to achieve higher levels of reduction.

Partial order methods are other approach to avoid the state repetitions that has been successfully employed, for example, in the formal verifier SPIN [9]. Specifically, the persistent set theory [5] allows verifying properties that only depend on the final state of the system and not on the history of traversed states. Unfortunately this method cannot be applied to our case, as we will see in section 3. Similarly, the theory of unfoldings [4] has been mainly developed for safe PNs while our application uses unbounded nets.

A different approach consists in storing the state space in a memory efficient manner. Implicit enumeration methods such as *binary decision diagrams*, *BDDs*, or *interval decision diagrams*, *IDDs*, aim at this goal [11]. However, the construction of these graphs tends to be time consuming and their performance highly depends on the specific problem. This makes them unattractive for our applications.

*Hash* tables [6] are another way to manage the storage of the reached states. We will use them in our work due to their simplicity and high efficiency.

## 3. PROBLEM DEFINITION

Our synthesis problem belongs to the same class that was introduced in [2]. We deal with a system specification composed of concurrent processes. Each process may have input and output ports to communicate with other processes or with the environment. Ports communicating with the environment are called *primary ports*. Primary ports written by the environment are called *uncontrollable* since the arrival of events to them is not regulated by the system.

The communication through ports occurs by means of unidirectional, point-to-point channels.

One of the main steps of our software synthesis methodology is the generation of a task schedule, verifying that (1) it is a cyclic schedule, (2) it has no deadlocks and (3) the schedule requires bounded memory resources (in other words, the cyclic execution of the program makes use of a finite number of buffers). The system is specified in a high level language similar to C but modified to allow communication operations. Processes are described as sequential programs that are executed concurrently. Later this specification is compiled to its underlying Petri net model. The scheduling process is carried out by means of reachability analysis for that Petri net [2, 7].

Traditionally, each of the processes of the system specification will be separately compiled on the target architecture. On the contrary, our synthesis process builds a set of *tasks* from the functional processes that are present in the starting specification. Each task is associated to one uncontrollable input port and performs the operations required to react to an event of that port. The novel idea is that those tasks may differ from the user specified processes. The compiler transformations are applied on each of these tasks, therefore optimizing the code that must be executed in response to an external event.

Figure 20-1. A sketches this idea. It depicts two processes A and B, each of them reading data from its corresponding ports. Processes communicate by means of the channel C.

However, the data that process B reads from port  $IN_B$  is processed independently of the data read from channel C. Hence, an efficient scheduling algorithm could reorder the source code to construct the threads 1 and 2 depicted in Figure 20-1.B. In this way, architecture specific optimizations will be performed on these separate code segments or tasks.

Next sections introduce the fundamentals of the scheduling approach used in our software synthesis methodology.

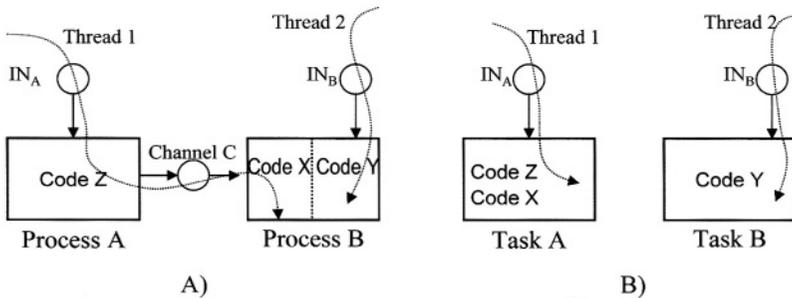


Figure 20-1. Code generation.

### 3.1. Petri net fundamentals

A *Petri net*,  $PN$  is a tuple  $(P, T, F, M_0)$  where  $P$  and  $T$  are sets of nodes of *place* or *transition* type respectively [7].  $F$  is a function of  $(P \times T) \cup (T \times P)$  to the set of non-negative integers. A *marking*  $M$  is a function of  $P$  to the set of non-negative integers. A *number of tokens* of  $p$  in  $M$ , where  $p$  is a *place* of the net, equals to the value of the function  $M$  for  $p$ , that is,  $M[p]$ . If  $M[p] > 0$  the place is said to be marked. Intuitively, the marking values of all the places of the net constitute the system state.

A PN can be represented as a bipartite directed graph, so that in case the function  $F(u, v)$  is positive, there will exist an edge  $[u, v]$  between such graph nodes  $u$  and  $v$ . The value  $F(u, v)$  is the edge weight. A transition  $t$  is enabled in a marking  $M$  if  $M[p] \geq F(p, t) \forall p \in P$ . In this case, the transition can be fired in the marking  $M$ , yielding a new marking  $M'$  given by  $M'[p] = M[p] - F(p, t) + F(t, p) \forall p \in P$ .

A marking  $M'$  is *reachable* from the *initial marking*  $M_0$  if there exists a sequence of transitions that can be sequentially fired from  $M_0$  to produce  $M'$ . Such a sequence is said to be *fireable* from  $M_0$ . The set of markings reachable from  $M_0$  is denoted  $\mathfrak{R}(M_0)$ . The *reachability graph* of a PN is a directed graph in which  $\mathfrak{R}(M_0)$  is a set of nodes and each edge  $[M, M']$  corresponds to a PN transition,  $t$ , such that the firing of  $t$  from marking  $M$  gives  $M'$ .

A transition  $t$  is called *source transition* if  $F(p, t) = 0 \forall p \in P$ . A pair of non source transitions  $t_i$  and  $t_j$  are in equal conflict if  $F(p, t_i) = F(p, t_j) \forall p \in P$ . An *equal conflict set*, *ECS* is a group of transitions that are in equal conflict.

A *place* is a *choice place* if it has more than one successor transition. If all the successor transitions of a choice *place* belong to the same ECS, the *place* is called *equal choice place*. A PN is *Equal Choice* if all the choice *places* are *equal*.

A choice *place* is called *unique choice* if in every marking of  $\mathfrak{R}(M_0)$  that *place* cannot have more than one enabled successor transition. A *unique-choice Petri net*, *UCPN*, is one in which all the choice *places* are either unique or equal. The PNs obtained after compiling our high-level specification present unique-choice ports.

### 3.2. Conditions for valid schedules

A schedule for a given PN is a directed graph where each node represents a marking of the PN and each edge joining two nodes stands for the firing of an enabled transition that leads from one marking to another. A valid schedule has five properties. First, there is a unique root node, corresponding to the starting marking. Second, for every marking node  $v$ , the set of edges that start from  $v$  must correspond to the transitions of an enabled ECS. If this ECS is a set of source transitions,  $v$  is called an *await node*. Third, the nodes  $v$  and  $w$  linked by an edge  $t$  are such that the marking  $w$  is obtained after firing

transition  $t$  from marking  $v$ . Fourth, each node has at least one path to one await node. Fifth, each await node is on at least one cycle.

As a consequence of these properties, valid schedules are cyclic, bounded and have no deadlocks.

### 3.3. Termination conditions in the state space exploration

The existence of source transitions in the PN results in an infinite reachability graph since the number of external events that can be generated is unlimited. Therefore, the design space that is explored must be pruned. Our implementation employs two types of termination criteria: static and dynamic. The static criterion consists in stopping the search whenever the buffer size of a given port exceeds the bounds established by the designer. Besides, we employ a dynamic criterion based on two steps (see [2] for further details):

1. First, the static port bounds, called *place degrees*, are imposed. The degree of a place  $p$  is the maximum of (a) the number of tokens of  $p$  in the initial marking and (b) the maximum weight of the edges incoming to  $p$ , plus the maximum weight of the edges outgoing from  $p$  minus one.

A place of a PN is saturated when its token number exceeds the degree of the place.

2. During the reachability graph construction, a marking will be discarded if it is deemed as *irrelevant*. A marking  $w$  is irrelevant if there exists a predecessor,  $v$ , such that for every place  $p$  of the marking  $M(v)$  of  $v$ , it is verified that (1)  $p$  has at least the same number of tokens in the marking  $M(w)$  (and possibly more) and (2) if  $p$  has more tokens in  $M(w)$  than in  $M(v)$ , then the number of tokens of  $p$  in  $M(v)$  is equal or greater than the degree of  $p$ .

### 3.4. Scheduling algorithm

The scheduling algorithm is based on a recursive approach using two procedures (*proc1* and *proc2*) that alternatively call each other. Procedure *proc1* receives a node representing a marking and iteratively explores all the enabled ECSs of that node. Exploring an ECS is done by calling procedure *proc2*. This one iteratively explores all the transitions that compose that ECS. The exploration of a transition is done by computing the marking node ( $M_{new}$ ) produced after firing it and calling *proc1* for that  $M_{new}$ .

The exploration of a path is terminated in procedure *proc2* when one of the following three outcomes happen:

- Finding an *Entry Point (EP)* for the  $M_{new}$  (an EP is a predecessor of  $M_{new}$  with exactly the same marking).
- The  $M_{new}$  is irrelevant with regard to some predecessor.
- The marking of  $M_{new}$  exceeds the maximum number of tokens specified by the user for some of the ports.

The first outcome leads to a valid schedule constructed so far. If, after returning from the recursive calls we find that the schedule is not valid we would re-explore the path again. To exhaustively explore the subgraph that exists after a given node  $v$  all the ECSs enabled in  $v$  must be explored. This is controlled by procedure *procl*.

The latter two outcomes do not produce a valid schedule. Hence, after returning from the recursive call, a re-exploration of new paths will be done providing that there are still paths that have not been exhaustively explored.

#### 4. REDUCING THE SIZE OF A SCHEDULE

Scheduling deals with an unbounded reachability space stemming from source transitions of the PN given as the specification. Termination criteria make the space under analysis bounded, but it is still very large in general. The efficiency of a scheduling procedure essentially depends on the ways of compacting the explored space. Compression techniques introduced in this section explore the reachability space at a coarser granularity level by identifying sequences of transitions fireable at given markings rather than firing a single transition at a time.

##### 4.1. Trace-based compression

The motivation behind a trace-based compression is given by a simple example of a multi-rate producer-consumer system (see Figure 20-2).

In this system a single iteration of the *producer* requires the *consumer* to iterate twice. A corresponding schedule is shown in Figure 20-2(b). Clearly if a scheduler can identify that it needs to fire twice the sequence of transitions corresponding to a single iteration of the consumer, then it could fire this sequence as a whole without firing individual transitions of the sequence. This would avoid storing internal markings of the consumer's iteration and would compress a schedule (see Figure 20-2(c)).

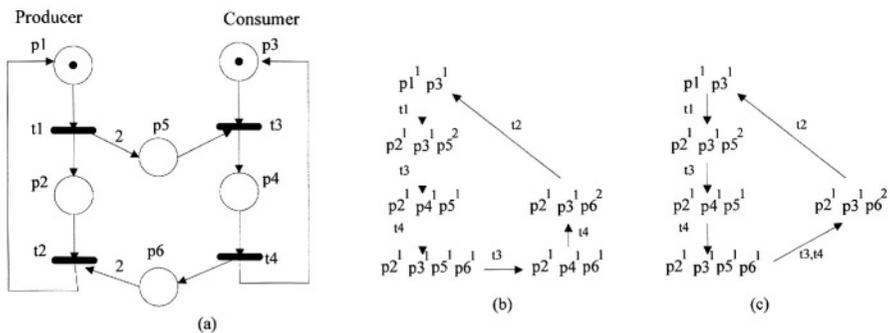


Figure 20-2. Trace-based compression: producer-consumer example.

Two questions need to be answered in applying the above compression technique.

1. How to choose a sequence of transitions which should be fired at once?
2. How to determine upfront that this sequence of transitions is fireable?

Let us first consider the second question, while the first question will be addressed in the Section 4.2. We use marking equations to check if a sequence of transitions is fireable. Since self-loops introduce inaccuracy in calculating the fireable transitions in the marking equations, let us assume in the rest of this section that a specification is provided as a PN without self-loops.<sup>1</sup> In the sequel, we may use a term *trace* to refer to a sequence of transitions.

**DEFINITION 1** (Consumption index of a trace). *Let a trace  $S$ , firing from a marking  $M_1$ , produce a sequence of markings  $M_1^S = M_1 \rightarrow M_2 \rightarrow \dots M_k$ . A consumption index of a place  $p$  with respect to  $M_1^S$  is the maximum  $M_1(p) - M_i(p)$  across all  $M_i \in M_1^S$  ( $consume(p, M_1) = \max_{M_i \in M_1^S} \{M_1(p) - M_i(p)\}$ ).*

Informally,  $consume(p, M^S)$  shows how much the original token count for a place  $p$  deviates while executing a trace  $S$  from a marking  $M^S$ . Applying the marking equations, the following is immediately implied:

**PROPERTY 1.** *A trace  $S$ , fireable from a marking  $M_i$ , is fireable from a marking  $M_j$  if and only if  $\forall p \in P: M_j(p) - consume(p, M_i) \geq 0$ .*

Let us illustrate the application of this trace-based compression with the producer-consumer example in Figure 20-2. When a scheduler has executed a trace  $t_3, t_4$  from a marking  $M_i = p_2^1 p_3^1 p_5^2$  it could compute the consumption indexes for that trace as:

$$\begin{aligned} compute(p_2, M_i) &= compute(p_4, M_i) = 0, \\ compute(p_3, M_i) &= compute(p_5, M_i) = 1 \end{aligned}$$

From this a scheduler can conclude that  $t_3, t_4$  is fireable from  $M_j = p_2^1 p_3^1 p_5^1 p_6^1$  because according to Property 1 this marking has enough token resources for executing the considered trace. Note that once a trace is known to be fireable at a given marking, the marking obtained after firing the trace can be analytically obtained by the marking equation. The scheduler can therefore fire the trace at once and compute the resulting marking, as shown in Figure 20-2(c).

#### 4.2. Path-based compression

The technique shown in the previous section relies on a scheduler to find candidate traces for compressing a schedule. This leaves the first question unanswered: how to find such candidate traces in the first place. This could be addressed through the analysis of the structure of the specification PN.

Let  $L = p_0, t_1, p_1, \dots, t_k, p_k$  be an acyclic path through PN nodes (places and transitions). The direction of edges in the path  $L$  defines a sequence of the transitions of  $L$ , in the order that appears in the path. We say that  $L$  is *fireable* at a given marking if the sequence of transitions defined in this way is fireable at the marking. We consider a sequence of markings of a path as follows.

**DEFINITION 2** (Marking sequence of a path). *A marking sequence  $M(L)$  of a path  $L = p_0, t_1, p_1, \dots, t_k, p_k$  is obtained by the consecutive application of marking equations for firing transitions  $t_1, \dots, t_k$  starting from a marking  $M_0 = \langle 0, \dots, 0 \rangle$ :  $M(L) = \{M_0, M_1, \dots, M_k\}$ , where  $M_i = M_{i-1} - F(p_{i-1}, t_i) + F(t_i, p_i)$ . We call  $M_k$  the intrinsic final marking of  $L$ .*

**DEFINITION 3** (Consumption index of a path). *A consumption index of a place  $p$  with respect to a path  $L$  is the maximum of  $-M_i(p)$  over all  $M_i$  that belong to a marking sequence  $M(L)$ , i.e.  $consume(p, L) = \max_{M_i \in M(L)} -M_i(p)$ .*

The statement similar to Property 1 links consumption indexes of a path with the path fireability.

**PROPERTY 2.** *A path  $L$  is fireable from a marking  $M$  if and only if  $\forall p \in P: M(p) - consume(p, L) \geq 0$ .*

The efficiency of the path-based compression strongly depends upon the choice of the candidate paths. In general, a PN graph has too many paths to be considered exhaustively. However the specification style might favour some of the paths with respect to others. For example it is very common to encapsulate the conditional constructs within a single block. The good specification styles usually suggest entering such blocks through the head (modelled in PN by a choice place) and exiting them through the tail (modelled in PN by a merge place). These paths are clearly good candidates to try during compression. Another potential advantage of paths from conditional constructs is that several of them might start from the same place and end with the same place. If they have the same consumption indexes and if their intrinsic final markings are the same, then only one of the paths needs to be stored because the conditions and final result of their firings are the same. Such alternative paths in PN are called *schedule-equivalent*.

A simple example of a code with conditional is shown in Figure 20-3, where the last parameter of each port access indicates the number of tokens to read or write.

Paths  $L1 = p_1, t_1, p_2, t_2, p_3$  and  $L2 = p_1, t_3, p_4, t_4, p_3$  correspond to alternative branches of computation and, as it is easy to see, are schedule-equivalent because they both start from the same place  $p_1$ , end by the same place  $p_3$  and consume the same number of tokens from input port  $IN$  with the same

```

PROCESS proc1(In_DPORT start,
  In_DPORT in, Out_DPORT out){
  int N, x, y, z;
  while (1) {
    READ_DATA(start, &N, 1);
    if (N>1000) {
      READ_DATA(in, &x, 1);
      y = 10 * x;}
    else {
      READ_DATA(in, &z, 1);
      y = z*z;}
    WRITE_DATA(out, &y, 1);}
}
    
```

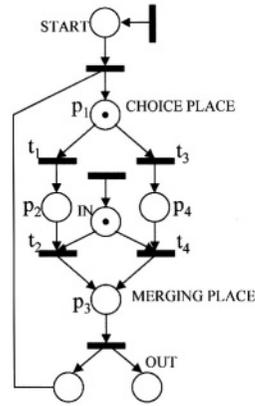


Figure 20-3. FlowC code and PN of an example candidate to path-based compression.

intrinsic final markings. Therefore in a compressed schedule they can be merged in a single transition corresponding to the final result of firing  $L1$  or  $L2$ . Note, that such merging is not possible by any known reduction technique based on structural information solely. It is a combination of structural and behavioural information that significantly expands the capabilities of reduction.

Let us formally check whether they can be fired from the initial marking shown in Figure 20-3.

The marking sequence of a path  $L1$  e.g. is  $\{0, p_1^{-1}p_2^1, p_1^{-1}IN^{-1}p_3^1\}$ . From this sequence one computes the consumption indexes:  $consume(p_1, L1) = consume(IN, L1) = 1$ , while for the rest of the places the consumption indexes are 0. Then, following Property 2 it is easy to conclude that transitions from the path  $L1$  are fireable under the initial marking of Figure 20-3.

Trace-based and path-based compressions are appealing techniques to reduce the complexity of scheduling procedures. An orthogonal approach for simplification of schedules is presented in next section.

### 5. AVOIDING REPETITION OF SCHEDULE STATES

In the conventional procedure from [2] during generation of a schedule its states are pruned looking at their prehistory only. Due to choices in a specification, the states with the same markings could be generated in alternative branches of computation. Because of this some compression possibilities might be overlooked. This section suggests an improvement over the conventional algorithm by removing repeated markings from alternative branches as well. Due to the special constraints that our synthesis application presents, the approach we have finally implemented to reduce the exploration space is based on hash tables. Hash tables are a much less complex method to store old

reached states than implicit representation methods. But, more importantly, their performance is much less problem dependent than that of BDDs or IDDs.

The main problem of hash tables is the existence of collisions. When the table is finite, two or more indexes can be mapped to the same table location, originating a collision. Several solutions can be applied, depending on the kind of verification that has to be done. Since we need to store all the generated states, we must discard the use of recent methods such as the *bit-state hashing* [3] used in the SPIN tool [9] or *hash compaction* [10], in which collisions produce the loss of states. To tackle the collisions we will employ hash tables with subtables implemented by means of *chained lists* [6].

### 5.1. New scheduling algorithm

In our application, the concept of predecessor node plays a key role. When a new marking is generated it is important to know whether it is irrelevant, what forces to keep information about its predecessors.

Thus, it is essential to maintain the scheduling graph structure. In other words, it is not sufficient to keep a set of previously reached states. On the contrary, the order relations among the markings must be stored. Hence, we still keep the current portion of the reachability graph that constitutes a valid schedule.

The scheduling algorithm after including the hash table remains basically the same. The only difference lies in the termination conditions presented in section 3.3. Now, a new termination criterion must be considered. The search will also be stopped whenever a new marking node  $v$  has the same marking as a previously reached node  $w$ . In this case the schedule from  $v$  will be identical to that from  $w$ . Hence node  $v$  will be merged to node  $w$ .

In the next sections we will devise a process model that will allow us to assess the amount of repeated states. Then we will perform some experiments to show the practical benefits, in terms of algorithmic complexity reduction, obtained after the incorporation of the hash tables.

### 5.2. Modeling the number of repeated states

Our implementation is based on a hash table with a chaining scheme to handle the occurrence of collisions. We apply a hash function based on the *xor* logic function to the key string that defines each marking. This function is inspired on that appearing in [1]. When the hash table is filled to a 75% of its capacity, it is enlarged one order of magnitude. The amount of repetitions produced by the cyclic execution of a process,  $P_0$ , can be assessed analyzing each single connected component (SCC) of  $P_0$ . For the moment, we will assume there is only a conditional inside the SCC. If we call  $p_E^{P_0}$  the place that starts the cyclic execution of  $P_0$  (*entry point* of  $P_0$ ), we can consider that the conditional partitions the SCC in the following sections:

**Predecessor path** ( $L_{pred}$ ): from  $p_E^{P_0}$  to the choice place.

**Successor path** ( $L_{succ}$ ): from the merging place to  $p_E^{P_0}$ .

**Branch paths** ( $L_{branch}$ ): those places belonging to each of the branches of the conditional. Given a conditional formed by  $N_B$  branches, the places belonging to a branch path will not belong to any of the other branch paths of the same conditional.

Assuming that the application execution starts in process  $P_0$  with the marking of  $p_E^{P_0}$ ,  $M_0$ , the number of repeated states will be equal to the number of states produced by:

1. Traversing the successor path, plus

Table 20-1. Scheduling results.

	System 1			System 2			System 3		
	WO	W	I%	WO	W	I%	WO	W	I%
# Places	10			26			34		
# Transitions	10			24			29		
# States	19	14	26	309	257	17	694	356	49
# Repeated states	5	0	-	52	0	-	338	0	-
CPU time (sec.)	0	0.01	-	0.25	0.21	16	1.03	0.6	42
Memory (KB)	49.7	60.23	-21	331	337	-2	614.4	542	12

2. Scheduling a cycle of each of the processes reading from ports that were written during step 1 or during the traversal of the paths  $L_{branch}$ . The entry point for one of these cycles is the port used by the process to communicate with  $P_0$ .

After performing steps 1 and 2, the schedule will have produced a cycle with respect to the starting marking  $M_0$ . These concepts are summarized in the following equation:

$$S_{P_0} = L_{pred} + N_B \times L_{succ} + \sum_{i=1}^{N_B} L_{branch_i} \tag{1}$$

where  $S_{P_0}$  is the number of states contained in the cyclic schedule of process  $P_0$  and  $N_B$  is the number of conditional branches. The number of repeated states of this schedule is equal to  $N_B - 1 \times L_{succ}$ .

If there were more than one conditional in the SCC, equation (1) should be applied to each of the subsections in which those conditionals would divide the SCC. If  $P_0$  were composed of several SCCs, equation (1) should be applied to all of them. Also, as the different paths are traversed, it may be necessary either to provide or consume the necessary tokens of the ports accessed during

the traversal. As we are seeking cyclic schedules, the processes that are involved in this token transactions must return to their original state. This means that a cycle of these processes must also be executed. All the produced states would add to the total number of computed repeated states.

### 5.3. Experimental results

The techniques introduced in this section have been tested with several examples. Tables 20-1 and 20-2 offer the scheduling results, performed on a AMD Athlon processor running at 700 MHz with 753 MB of RAM. Columns labelled with *WO* were obtained without employing the hash table.

Table 20-2. Scheduling results (com.).

	System 4			System 5			MPEG		
	WO	W	I%	WO	W	I%	WO	W	I%
# Places	34			34			115		
# Transitions	30			30			106		
# States	2056	702	66	2987	1016	66	4408	108	97
# Repeated states	1017	0	-	1500	0	-	3805	0	-
CPU time (sec.)	3.11	1.03	67	5.43	1.74	68	23.12	0.53	98
Memory (KB)	1204	843	30	1687	1132	33	2226	462	79

Columns tagged with *W* show results with the hash table. The columns labelled with *I* show the percentage improvement both in memory and CPU time obtained if hash tables are used.

All the examples are variations of a producer-consumer system targeted to multi-media applications such as video decoding. The processes communicate through FIFO buffers. System 1 is composed of two processes with a producing/consuming rate (*pcr*) equal to two. System 2 incorporates a controller to better manage the synchronization among processes, and the *pcr* is equal to 48 in this case. System 3 has the same *pcr* but includes some additional processes that perform filtering functions on the data the producer sends to the consumer. System 4 is equivalent to System 3 but including two conditionals in the SCC, each one with two equivalent branches. Finally, System 5 is identical to System 4 except that its *pcr* is equal to 70.

On the other hand, MPEG is a real world example of an MPEG algorithm composed of five processes.

In System 1 the memory overhead introduced by the hash table is larger than the reduction obtained by avoiding the state repetition. This is so because the small size of this example produces few states to explore. However, the rest of examples show that as the number of conditionals increases and as the difference between the producing and the consuming rates raises, the benefits in CPU time and consumed memory obtained when using hash tables soar up.

In the case of the MPEG example the improvements achieved when using the hash table are spectacular. This is mainly due to the presence of many quasi-equivalent paths.

A noteworthy aspect is that due to the employment of search heuristics to select the next ECS to fire, the schedules obtained when using hash tables can be different from those generated in their absence. As can be deduced from the former tables, the number of different states in the schedule tends to be lower when the hash table is used. Since the reachability graph needs to be traversed by the code generation phase, this means that the obtained schedule could potentially be more efficient and help to produce code of better quality.

## 6. CONCLUSIONS

In this paper we have addressed the problem of reducing the solution space explored during quasi-static scheduling in a software synthesis methodology for one-processor, embedded, reactive systems.

We have studied different ways to compress the state space to reduce the size of the schedule. We have shown that the structure of the specification PN can be examined to identify sequences of traces that could be fired at once during the scheduling with the help of marking equations. Further, we have shown that a scheme based on the use of efficient hash tables provides an effective way to eliminate state repetitions.

The experimental results we have obtained demonstrate the usefulness of our technique, yielding outstanding reductions in the CPU time and memory consumed by the scheduling algorithm for large-scale, real world problems.

## ACKNOWLEDGEMENTS

This work is partly supported by MCYT projects TIC2000-0583-C02 and HI2001-0033.

## NOTE

<sup>1</sup> This requirement does not impose restrictions because any PN with self-loops can be transformed into self-loop-free PN by inserting dummy transitions.

## REFERENCES

1. A. V. Aho, R. Sethi and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.
2. Cortadella, J. et al. "Task Generation and Compile-Time Scheduling for Mixed Data-Control Embedded Software." In *Design Automation Conference*, pp. 489–494, 2000.

3. Eckerle, Jurgen and Lais. "New Methods for Sequential Hashing with Supertrace." *Technical Report*, Institut fur Informatik, Universitat Freiburg, Germany, 1998.
4. J. Esparza, S. Römer and W. Vogler. "An Improvement of McMillan's Unfolding Algorithm." In *TACAS'96*, 1996.
5. P. Godefroid. *Partial Order Methods for the Verification of Concurrent Systems. An Approach to the State-Explosion Problem*. Ph.D. thesis, Universite de Liege, 1995.
6. R. Jain. "A comparison of Hashing Schemes for Address Lookup in Computer Networks." *IEEE Trans. On Communications*, 4(3): 1570-1573, 1992.
7. T. Murata. "Petri Nets: Properties, Analysis and Applications." *Proceedings of the IEEE*, Vol. 77, No. 4, pp. 541-580, 1989.
8. K. Schmidt. "Integrating Low Level Symmetries into Reachability Analysis." In *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 315-330, Springer Verlag, 2000.
9. SPIN. <http://spinroot.com/spin/whatispin.html>, 2003.
10. U. Stern and D. L. Dill. "Combining State Space Caching and Hash Compaction." In *GI/ITG/GME Workshop*, 1996.
11. K. Strehl et al. "FunState - An Internal Design Representation for Codesign." *IEEE Trans. on VLSI Systems*, Vol. 9, No. 4, pp. 524-544, August 2001.

# SIMULATION TRACE VERIFICATION FOR QUANTITATIVE CONSTRAINTS

Xi Chen<sup>1</sup>, Harry Hsieh<sup>1</sup>, Felice Balarin<sup>2</sup> and Yosinori Watanabe<sup>2</sup>

<sup>1</sup> *University of California, Riverside, CA, USA*; <sup>2</sup> *Cadence Berkeley Laboratories, Berkeley, CA, USA*

**Abstract.** System design methodology is poised to become the next big enabler for highly sophisticated electronic products. Design verification continues to be a major challenge and simulation will remain an important tool for making sure that implementations perform as they should. In this paper we present algorithms to automatically generate C++ checkers from any formula written in the formal quantitative constraint language, Logic Of Constraints (LOC). The executable can then be used to analyze the simulation traces for constraint violation and output debugging information. Different checkers can be generated for fast analysis under different memory limitations. LOC is particularly suitable for specification of system level quantitative constraints where relative coordination of instances of events, not lower level interaction, is of paramount concern. We illustrate the usefulness and efficiency of our automatic trace verification methodology with case studies on large simulation traces from various system level designs.

**Key words:** quantitative constraints, trace analysis, logic of constraints

## 1. INTRODUCTION

The increasing complexity of embedded systems today demands more sophisticated design and test methodologies. Systems are becoming more integrated as more and more functionality and features are required for the product to succeed in the marketplace. Embedded system architecture likewise has become more heterogeneous as it is becoming more economically feasible to have various computational resources (e.g. microprocessor, digital signal processor, reconfigurable logics) all utilized on a single board module or a single chip. Designing at the Register Transfer Level (RTL) or sequential C-code level, as is done by embedded hardware and software developers today, is no longer efficient. The next major productivity gain will come in the form of system level design. The specification of the functionality and the architecture should be done at a high level of abstraction, and the design procedures will be in the form of refining the abstract functionality and the abstract architecture, and of mapping the functionality onto the architecture through automatic tools or manual means with tools support [1, 2]. High level design procedures allow the designer to tailor their architecture to the functionality at hand or to modify their functionality to suit the available architectures (see

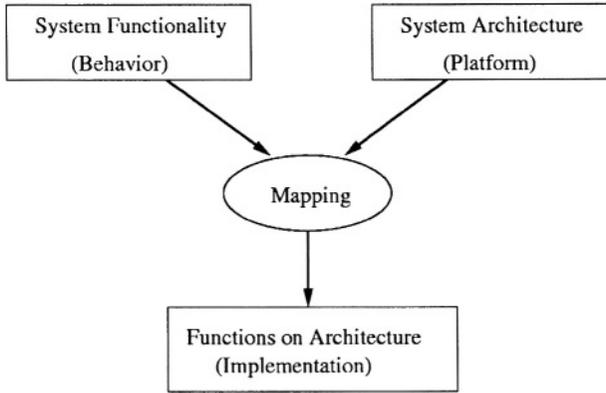


Figure 21-1. System design methodology.

Figure 21-1). Significant advantages in flexibility of the design, as compared to today's fixed architecture and *a priori* partitioning approach, can result in significant advantages in the performance and cost of the product.

In order to make the practice of designing from high-level system specification a reality, verification methods must accompany every step in the design flow from high level abstract specification to low level implementation. Specification at the system level makes formal verification possible [3]. Designers can prove the property of a specification by writing down the property they want to check in some logic (e.g. Linear Temporal Logic (LTL) [4], Computational Tree Logic (CTL) [5]) and use a formal verification tool (e.g. Spin Model checker [6, 7], Formal-Check [8], SMV [9]) to run the verification. At the lower level, however, the complexity can quickly overwhelm the automatic tools and the simulation quickly becomes the workhorse for verification.

The advantage of simulation is in its simplicity. While the coverage achieved by simulation is limited by the number of simulation vectors, simulation is still the standard vehicle for design analysis in practical designs. One problem of simulation-based property analysis is that it is not always straightforward to evaluate the simulation traces and deduce the absence or presence of an error. In this paper, we propose an efficient automatic approach to analyze simulation traces and check whether they satisfy quantitative properties specified by denotational logic formulas. The property to be verified is written in Logic of Constraints (LOC) [10], a logic particularly suitable for specifying constraints at the abstract system level, where coordination of executions, not the low level interaction, is of paramount concern. We then automatically generate a C++ trace checker from the quantitative LOC formula. The checker analyzes the traces and reports any violations of the LOC formula. Like any other simulation-based approach, the checker can only disprove the LOC formula (if a violation is found), but it can never prove it

conclusively, as that would require analyzing infinitely many traces. The automatic checker generation is parameterized, so it can be customized for fast analysis for specific verification environment. We illustrate the concept and demonstrate the usefulness of our approach through case studies on two system level designs. We regard our approach as similar in spirit to symbolic simulation [11], where only particular system trajectory is formally verified (see Figure 21-2). The automatic trace analyzer can be used in concert with model checker and symbolic simulator. It can perform logic verification on a single trace where the other approaches failed due to excessive memory and space requirement.

In the next section, we review the definition of LOC and compare it with other forms of logic and constraint specification. In section 3, we discuss the algorithm for building a trace checker for any given LOC formula. We demonstrate the usefulness and efficiency with two verification case studies in section 4. Finally, in section 5, we conclude and provide some future directions.

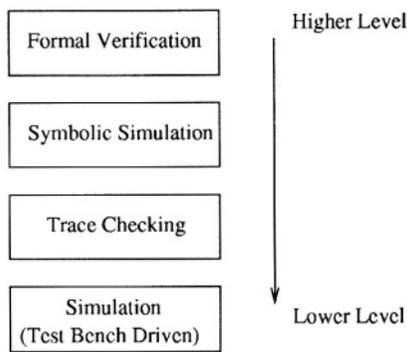


Figure 21-2. System verification approaches.

## 2. LOGIC OF CONSTRAINTS (LOC)

Logic Of Constraints [10] is a formalism designed to reason about simulation traces. It consists of all the terms and operators allowed in sentential logic, with additions that make it possible to specify system level quantitative constraints without compromising the ease of analysis. The basic components of an LOC formula are:

- **event names:** An input, output, or intermediate signal in the system. Examples of event names are “in”, “out”, “Stimuli”, and “Display”;
- **instances of events:** An instance of an event denotes one of its occurrence in the simulation trace. Each instance is tagged with a positive integer index, strictly ordered and starting from “0”. For example, “Stimuli[0]” denotes the first instance of the event “Stimuli” and “Stimuli[1]” denotes the second instance of the event;

- **index and index variable:** There can be only one index variable  $i$ , a positive integer. An index of an event can be any arithmetic operations on  $i$  and the annotations. Examples of the use of index variables are “ $Stimuli[i]$ ”, “ $Display[i-5]$ ”;
- **annotation:** Each instance of the event may be associated with one or more annotations. Annotations can be used to denote the time, power, or area related to the event occurrence. For example, “ $t(Display[i-5])$ ” denotes the “ $t$ ” annotation (probably time) of the “ $i-5$ ”th instance of the “ $Display$ ” event. It is also possible to use annotations to denote relationships between different instances of different event. An example of such a relationship is causality. “ $t(in[cause(out[i])])$ ” denotes the “ $t$ ” annotation of an instance of “ $in$ ” which in turn is given by the “ $cause$ ” annotation of the “ $i$ ”th instance of “ $out$ ”.

LOC can be used to specify some very common real-time constraints:

- **rate**, e.g. “a new  $Display$  will be produced every 10 time units”:
 
$$t(Display[i+1]) - t(Display[i]) = 10 \quad (1)$$
- **latency**, e.g. “ $Display$  is generated no more than 45 time units after  $Stimuli$ ”:
 
$$t(Display[i]) - t(Stimuli[i]) \leq 45 \quad (2)$$
- **jitter**, e.g. “every  $Display$  is no more than 15 time units away from the corresponding tick of the real-time clock with period 15”:
 
$$|t(Display[i]) - i * 10| \leq 15 \quad (3)$$
- **throughput**, e.g. “at least 100  $Display$  events will be produced in any period of 1001 time units”:
 
$$t(Display[i+100]) - t(Display[i]) \leq 1001 \quad (4)$$
- **burstiness**, e.g. “no more than 1000  $Display$  events will arrive in any period of 9999 time units”:
 
$$t(Display[i+1000]) - t(Display[i]) > 9999 \quad (5)$$

As pointed out in [10], the latency constraints above is truly a latency constraint only if the  $Stimuli$  and  $Display$  are kept synchronized. Generally, we will need an additional annotation that denotes which instance of  $Display$  is “caused” by which instance of the  $Stimuli$ . If the  $cause$  annotation is available, the latency constraints can be more accurately written as:

$$t(Display[i]) - t(Stimuli[cause(Display[i])]) \leq 45 \quad (6)$$

and such an LOC formula can easily be analyzed through the simulation checker presented in the next section. However, it is the responsibility of the designer, the program, or the simulator to generate such an annotation.

By adding additional index variables and quantifiers, LOC can be extended to be at least as expressive as S1S [12] and Linear Temporal Logic. There is no inherent problem in generating simulation monitor for them. However,

the efficiency of the checker will suffer greatly as memory recycling becomes impossible (as will be discussed in the next section). In similar fashion, LOC differs from existing constraint languages (e.g. Rosetta [13], Design Constraints Description Language [14], and Object Constraint Language [15]) in that it allows only limited freedom in specification to make the analysis tractable. The constructs of LOC are precisely chosen so system-level constraints can be specified and efficiently analyzed.

### 3. THE LOC CHECKER

We analyze simulation traces for LOC constraint violation. The methodology for verification with automatically generated LOC checker is illustrated in Figure 21-3. From the LOC formula and the trace format specification, an automatic tool is used to generate a C++ LOC checker. The checker is compiled into an executable that will take in simulation traces and report any constraint violation. To help the designer to find the point of error easily, the error report will include the value of index  $i$  which violates the constraint and the value of each annotation in the formula (see Figure 21-4). The checker is designed to keep checking and reporting any violation until stopped by the user or if the trace terminates.

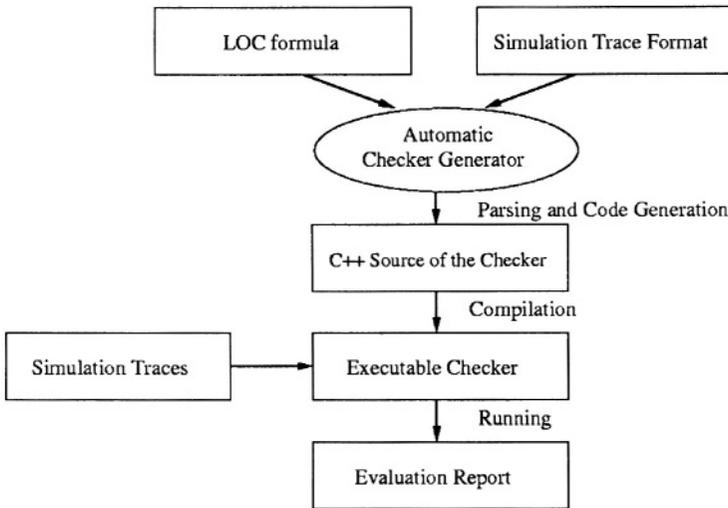


Figure 21-3. Trace analysis methodology.

The algorithm progresses based on index variable  $i$ . Each LOC formula instance is checked sequentially with the value of  $i$  being 0, 1, 2, . . . , etc. A formula instance is a formula with  $i$  evaluated to some fix positive integer number. The basic algorithm used in the checker is given as follows:

**Algorithm of LOC Checker:**

```

i = 0;
memory_used = 0;

Main {
  while(trace not end){
    if(memory_used < MEM_LIMIT){
      read one line of trace;
      store useful annotations;
      check_formula();
    }
    else{
      while(annotations for current
            formula instance is not
            available && trace not end)
        scan the trace for annotation;
      check_formula();
    }
  }
}

check_formula {
  while (can evaluate formula instance i) {
    evaluate formula instance i;
    i++;
    memory recycling;
  }
}

```

```

cs220@chimera $ checker latency.trace
Reading from trace file "latency.trace" ...

Formula t(Display[i]) - t(Stimuli[i]) <= 25 is violated
at trace line# 278:  Display: -6 at time 87

where i = 23
t (Display[i]) = 87
t (Stimuli[i]) = 60
          ⋮

```

*Figure 21-4.* Example of error report.

The time complexity of the algorithm is linear to the size of the trace. The memory usage, however, may become prohibitively high if we try to keep the entire trace in the memory for analysis. As the trace file is scanned in, the proposed checker attempts to store only the useful annotations and in addition, evaluate as many formula instances as possible and remove from memory parts of the trace that are no longer needed (memory recycling). The algorithm tries to read and store the trace only once. However, after the memory usage reaches the preset limit, the algorithm will not store the annotation information any more. Instead, it scans the rest of the trace looking for needed events and annotations for evaluating the current formula instance (current  $i$ ). After freeing some memory space, the algorithm resumes the reading and storing of annotation from the same location. The analysis time will certainly be impacted in this case (see Table 21-3). However, it will also allow the checker to be as efficient as possible, given the memory limitation of the analysis environment.

For many LOC formulas (e.g. constraints 1–5), the algorithm uses a fixed amount of memory no matter how long the traces are (see Table 21-2). Memory efficiency of the algorithm comes from being able to free stored annotations as their associated formula instances are evaluated (memory recycling). This ability is directly related to the choice made in designing LOC. From the LOC formula, we often know what annotation data will not be useful any more once all the formula instance with  $i$  less than a certain number are all evaluated. For example, let's say we have an LOC formula:

$$t(input[i+10]) - t(output[i+5]) < 300 \quad (7)$$

and the current value of  $i$  is 100. Because the value of  $i$  increases monotonically, we know that event *input*'s annotation  $t$  with index less than 111 and event *output*'s annotation  $t$  with index less than 106 will not be useful in the future and their memory space can be released safely. Each time the LOC formula is evaluated with a new value of  $i$ , the memory recycling procedure is invoked, which ensures minimum memory usage.

#### 4. CASE STUDIES

In this section, we apply the methodology discussed in the previous section to two very different design examples. The first is a Synchronous Data Flow (SDF) [16] design called Expression originally specified in Ptolemy and is part of the standard Ptolemy II [17] distribution. The Expression design is respecified and simulated with SystemC simulator [18]. The second is a Finite Impulse Response (FIR) filter written in SystemC and is actually part of the standard SystemC distribution. We use the generated trace checker to verify a wide variety of constraints.

#### 4.1. Expression

Figure 21-5 shows a SDF design. The data generators SLOW and FAST generate data at different rates, and the EXPR process takes one input from each, performs some operations (in this case, multiplication) and outputs the result to DISPLAY. SDF designs have the property that different scheduling will result in the same behavior. A snapshot of the simulation trace is shown in Figure 21-6.

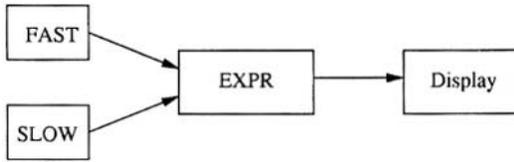


Figure 21-5. Expression design example.

The following LOC formula must be satisfied for any correct simulation of the given SDF design:

$$SLOW[i] * FAST[i] = DISPLAY[i] \quad (8)$$

We use the automatically generated checker to show that the traces from SystemC simulation adhere to the property. This is certainly not easy to infer from manually inspecting the trace files, which may contain millions of lines. As expected, the analysis time is linear to the size of the trace file and the maximum memory usage is constant regardless of the trace file size (see Table 21-1). The platform for experiment is a dual 1.5 GHz Athlon system with 1 GB of memory.

```

FAST output data: 0.314
SLOW output data: 0.0314
FAST output data: 0.628
SLOW output data: 0.0628
DISPLAY the result: 0.0098596
FAST output data: 0.942
SLOW output data: 0.0942
DISPLAY the result: 0.0394384
⋮
  
```

Figure 21-6. Expression simulation trace.

Table 21-1. Results of Constraint (8) on EXPR.

Lines of traces	$10^4$	$10^5$	$10^6$	$10^7$
Time used (s)	< 1	1	12	130
Memory usage	8 KB	8 KB	8 KB	8 KB

## 4.2. FIR filter

Figure 21-7 shows a 16-tap FIR filter that reads in samples when the input is valid and writes out the result when output is ready. The filter design is divided into a control FSM and a data path. The test bench feeds sampled data of arbitrary length and the output is displayed with the simulator.

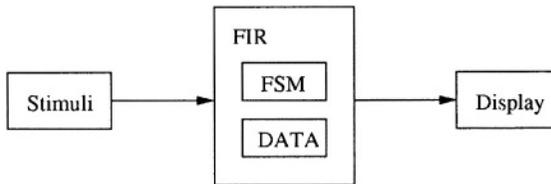


Figure 21-7. FIR design example.

We utilize our automatic trace checker generator and verify the properties specified in constraints (1)–(5). The same trace files are used for all the analysis. The time and memory requirements are shown in Table 21-2. We can see that the time required for analysis grows linearly with the size of the trace file, and the maximum memory requirement is formula dependent but stays fairly constant. Using LOC for verification of common real-time constraints is indeed very efficient.

Table 21-2. Result of constraints (1–5) on FIR.

Lines of traces		$10^4$	$10^5$	$10^6$	$10^7$
Constraint (1)	Time(s)	< 1	1	8	89
	Memory	28 B	28 B	28 B	28 B
Constraint (2)	Time(s)	< 1	1	12	120
	Memory	28 B	28 B	28 B	28 B
Constraint (3)	Time(s)	< 1	1	7	80
	Memory	24 B	24 B	24 B	24 B
Constraint (4)	Time(s)	< 1	1	7	77
	Memory	0.4 KB	0.4 KB	0.4 KB	0.4 KB
Constraint (5)	Time(s)	< 1	1	7	79
	Memory	4 KB	4 KB	4 KB	4 KB

We also verify constraint (6) using the simulation analyzer approach. Table 21-3 shows that the simulation time grows linearly with the size of the trace file. However, due to the use of an annotation in an index expression, memory can no longer be recycled with the algorithm in the previous section and we see that it also grows linearly with the size of the trace file. Indeed, since we will not know what annotation will be needed in the future, we can never remove any information from memory. If the memory is a limiting factor in the simulation environment, the analysis speed must be sacrificed to allow the verification to continue. This is shown in Table 21-3 where the memory usage is limited to 50 KB. We see that the analysis takes more time when the memory limitation has been reached. Information about trace pattern can be used to dramatically reduce the running time under memory constraints. Aggressive memory minimization techniques and data structures can also be used to further reduce time and memory requirements. For most LOC formulas, however, the memory space can be recycled and the memory requirements are small.

Table 21-3. Result of constraint (6) on FIR.

Lines of traces		$2 \times 10^4$	$3 \times 10^4$	$4 \times 10^4$	$5 \times 10^4$
Unlimited memory	Time (s)	< 1	< 1	< 1	1
	Mem (KB)	40	60	80	100
Memory limit (50 KB)	Time (s)	< 1	61	656	1869
	Mem (KB)	40	50	50	50

## 5. CONCLUSION

In this paper we have presented a methodology for system-level verification through automatic trace analysis. We have demonstrated how we take any formula written in the formal quantitative constraint language, Logic Of Constraints, and automatically generate a trace checker that can efficiently analyze the simulation traces for constraint violations. The analyzer is fast even under memory limitation. We have applied the methodology to many case studies and demonstrate that automatic LOC trace analysis can be very useful.

We are currently considering a few future enhancements and novel applications. One such application we are considering is to integrate the LOC analyzer with a simulator that is capable of non-deterministic simulation, non-determinism being crucial for design at high level of abstraction. We will use the checker to check for constraint violations, and once a violation is found, the simulation could roll back and look for another non-determinism resolution that will not violate the constraint.

## ACKNOWLEDGEMENTS

We gratefully acknowledge the preliminary work by Artur Kedzierski who did experiments on LOC formula parsing and checker generation. We also would like to thank Lingling Jin who wrote and debug the Metropolis Meta-Model source code for the EXPR example.

## REFERENCES

1. K. Keutzer, S. Malik, A. R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. "System Level Design: Orthogonalization of Concerns and Platform-Based Design." *IEEE Transactions on Computer-Aided Design*, Vol. 19, No. 12, December 2000.
2. F. Balarin, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, M. Sgroi, and Y. Watanabe. "Modeling and Designing Heterogeneous Systems." *Technical Report 2001/01 Cadence Berkeley Laboratories*, November 2001.
3. X. Chen, F. Chen, H. Hsieh, F. Balarin, and Y. Watanabe. "Formal Verification of Embedded System Designs at Multiple Levels of Abstraction". *Proceedings of International Workshop on High Level Design Validation and Test – HLDVT02*, September 2002.
4. P. Godefroid and G. J. Holzmann. "On the Verification of Temporal Properties". *Proceedings of IFIP/WG6.1 Symposium on Protocols Specification, Testing, and Verification*, June 1993.
5. T. Hafer and W. Thomas. "Computational Tree Logic and Path Quantifiers in the Monadic Theory of the Binary Tree." *Proceedings of International Colloquium on Automata, Languages, and Programming*, July 1987.
6. Gerard J. Holzmann. "The Model Checker Spin." *IEEE Transactions on Software Engineering*, Vol. 23, No. 5, pp. 279–258, May 1997.
7. Spin manual, "<http://netlib.bell-labs.com/netlib/spin/whatispin.html>".
8. FormalCheck, "<http://www.cadence.com/products/formalcheck.html>".
9. K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
10. F. Balarin, Y. Watanabe, J. Burch, L. Lavagno, R. Passerone, and A. Sangiovanni-Vincentelli. "Constraints Specification at Higher Levels of Abstraction." *Proceedings of International Workshop on High Level Design Validation and Test – HLDVT01*, November 2001.
11. C. Blank, H. Eveking, J. Levihn, and G. Ritter. "Symbolic Simulation Techniques: State-of-the-Art and Applications." *Proceedings of International Workshop on High-Level Design Validation and Test – HLDVT01*, November 2001.
12. A. Aziz, F. Balarin, R. K. Brayton and A. Sangiovanni-Vincentelli. "Sequential Synthesis Using SIS." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 19, No. 10, pp. 1149–1162, October 2000.
13. P. Alexander, C. Kong, and D. Barton. "Rosetta usage guide". <http://www.sldi.org>, 2001.
14. Quick reference guide for the Design Constraints Description Language, <http://www.eda.org/dcwg>, 2000.
15. Object Constraint Language specification, <http://www.omg.org>, 1997.
16. E. Lee and D. Messerschmitt. "Synchronous Data Flow." *Proceedings of IEEE*, pp. 55–64, September 1987.
17. Ptolemy home page, "<http://www.ptolemy.eecs.berkeley.edu>".
18. SystemC home page, "<http://www.systemc.org>".

*This page intentionally left blank*

PART VI:

ENERGY AWARE SOFTWARE TECHNIQUES

*This page intentionally left blank*

## Chapter 22

# EFFICIENT POWER/PERFORMANCE ANALYSIS OF EMBEDDED AND GENERAL PURPOSE SOFTWARE APPLICATIONS

*A Pre-Characterization Free Approach*

Venkata Syam P. Rapaka, and Diana Marculescu  
*Carnegie Mellon University*

**Abstract.** This chapter presents a novel approach for an efficient, yet accurate estimation technique for power consumption and performance of embedded and general-purpose applications. Our approach is adaptive in nature and is based on detecting sections of code characterized by high temporal locality (also called *hotspots*) in the execution profile of the benchmark being executed on a target processor. The technique itself is *architecture* and *input independent* and can be used for both embedded, as well as for general-purpose processors. We have implemented a hybrid simulation engine, which can significantly shorten the simulation time by using on-the-fly profiling for critical sections of the code and by reusing this information during power/performance estimation for the rest of the code. By using this strategy, we were able to achieve up to 20× better accuracy compared to a flat, non-adaptive sampling scheme and a simulation speed-up of up to 11.84× with a maximum error of 1.03% for performance and 1.92% for total energy on a wide variety of media and general-purpose applications.

**Key words:** power estimation, simulation speedup, application hotspots

### 1. INTRODUCTION

Embedded or portable computer systems play an increasingly important role in today's quest for achieving true ubiquitous computing. Since power consumption and performance have a direct impact on the success of not only embedded, but also high performance processors, designers need efficient and accurate tools to evaluate the efficacy of their software and architectural innovations.

To estimate power consumption and performance, a designer can make a choice from a variety of simulators at various levels of abstraction, ranging from transistor or layout-level [1] to architectural [2, 3] and instruction-level [4–7]. The lowest level simulators provide the most detailed and accurate statistics, while the higher-level simulators trade off accuracy for simulation speed and portability. Although high-level simulators offer high speedup when compared to low-level simulators, they are still time consuming and may take days to simulate very large, practical benchmarks. At the same time,

acceptable ranges of accuracy are subject to change when refining the design in various stages of the design cycle. Hence, it is desirable to speed-up existing simulators at various levels of abstraction without compromising on their accuracy.

In this chapter, we describe our strategy for accelerating simulation speed, without significant loss in accuracy. Such a strategy has the additional benefit of being able to adapt itself to the behavior of the benchmark being simulated. Hence, it can predict the power consumption and performance statistics *without* complete detailed analysis of the execution profile of the application being executed and *without* any pre-characterization of the benchmark being simulated. Our strategy is generic and can be adopted by any simulator, at any level of abstraction, to accelerate the simulation process.

### 1.1. Prior work

At software or architecture-level, various schemes and strategies have been proposed for speeding-up the power estimation process. Techniques like macro-modeling [8, 9], function level power estimation [10] and energy caching [9] are some of the proposed strategies used for accelerating power simulators, while keeping the accuracy within acceptable range. Macro-modeling and functional level power estimation provide speedup at the cost of time consuming pre-characterization of programs. Energy caching is based on the energy and delay characteristics of a code segment, and can be used only when these statistics are uniform across the entire execution of the code segment.

A *two-level* simulation approach has been described in [16] for estimating energy and performance with sufficient accuracy. The technique uses a flat, *fixed-window* sampling scheme; coupled with a program phase detection technique, which decides, on-the-fly when to use a detailed vs. a non-detailed mode of simulation. However, the approach does not adapt the *sampling window size* to the application profile to achieve better accuracy, nor does it try to detect finer-grain changes in program phases that could be exploited for better speed-up. In this chapter, we introduce a *hybrid simulation engine*, which is able to fully adapt to the application behavior and provide up to 20× better accuracy than the fixed-window sampling technique presented previously. Our work complements existing techniques for gate and RT-level power estimation based on sequence compaction [11] by recognizing the effect of fine and coarse grain temporal dependencies, present in common software applications.

### 1.2. Chapter overview and contributions

Our main goal is to accelerate existing simulators, by predicting power and performance values accurately. This scheme can be applied to simulators at various levels of abstraction to reduce the simulation time without compro-

mising accuracy. Without loss of generality, to validate our strategy, as a baseline simulator, we have chosen *Wattch* [2], a framework for architectural-level power analysis and optimizations. *Wattch* has been implemented on top of *SimpleScalar* [12] tool set and is based on a suite of parameterizable power models. Based on these power models, *Wattch* can estimate power consumed by the various hardware structures based on per-cycle resource usage counts, generated through cycle-accurate simulation. *Wattch* has considerable speedup (1000×) when compared to circuit-level power estimation tools, and yet can estimate results within 10% of the results generated by Spice. But even with this speedup, it can take very long time to simulate most benchmarks of practical interest. It is thus, desirable to further reduce the simulation time without trading off accuracy.

*Wattch* uses per cycle statistics generated by SimpleScalar to estimate the power consumed by various components of the architecture being simulated. As in most other cycle-accurate tools, to get sufficiently accurate statistics, one must perform a *detailed* simulation of the benchmark program, at the cost of increased simulation time. In both *Wattch* and *SimpleScalar*, the program can also be executed in a *fast* mode, in which case the program will be executed correctly, but without cycle accurate information.

Our strategy involves using a hybrid simulator, which is capable of switching between the detailed and fast modes of simulation. The rationale for using such a simulation strategy stems from the inherent behavior of most programs of practical interest. In fact, most benchmark programs are made up of tightly coupled regions of code or *hotspots* [13], in which the program behaves in a predictable manner, by executing sections of code with high temporal locality. Once the program enters a hotspot, one can identify critical sections of the code that should be simulated in detailed mode, record per cycle information like *Instructions Per Cycle* (IPC) and *Energy Per Cycle* (EPC), and complete functional simulation of the hotspot by switching into fast mode. While this strategy has been used before in the context of a fixed-size sampling window [16], we identify the shortcomings associated with such a scheme and propose a truly application-adaptive sampling scheme with up to 20x better accuracy. Our scheme provides up to 11.8× speed-up compared to the baseline, cycle-accurate simulation engine, while keeping accuracy within 2% on average for both performance and power consumption. In addition, the proposed approach offers superior accuracy for fine-grain, per module energy estimates (less than 2% compared to up to 18% estimation error), as well as energy and performance run-time profiles that closely follow the actual, detailed profile for benchmarks under consideration.

### 1.3. Organization of the chapter

The rest of this chapter is organized as follows: Section 2 discusses hotspots and the behavior of code inside hotspots. We describe our proposed approach for identifying program regularity and present our strategy in greater detail

in Section 3. Practical considerations are discussed in Section 4. We present our experimental results and discuss them in Section 5. Section 6 concludes the chapter with some final remarks.

## 2. PROGRAM BEHAVIOR

As it is well known, most common applications exhibit sections of code with high temporal locality. Such characteristics define the so-called *hotspots*, which are collections of tightly coupled basic blocks, executing together most of the time. When an application program enters a hotspot, it executes only the basic blocks belonging to that hotspot, and only rarely steps out of this set of basic blocks. Two typical hotspots are shown in Figure 22-1. In this case, the code executes the basic blocks of hotspot A for a significant portion of the total execution time, before it starts executing those of hotspot B. The program maintains a very high temporal locality once it enters a hotspot, and, due to high temporal locality, it behaves in a *predictable* manner while running inside the hotspot.

### 2.1. Hotspot properties

Hotspots are typically tight loops or series of instructions that are executed repetitively for the entire duration of the hotspot. This repetitive behavior is reflected in how performance and power-related statistics behave. In fact, inside a hotspot, all functional units along with other hardware blocks are accessed in a specific repetitive pattern. This is true even for architectures supporting *out-of-order* execution as the dynamic schedule for the same set

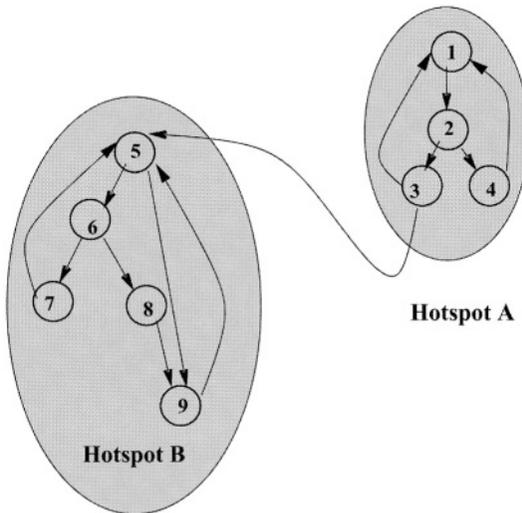


Figure 22-1. An example of two hotspots [16].

of instructions almost always results in the same scheduled trace of instructions inside of the given hotspot.

Hotspots create regular patterns of execution profiles during the course of execution of a benchmark. Due to these execution patterns, it is *not essential* to do a cycle accurate detailed simulation for the entire duration of each of the hotspots. Thus, one can use the IPC and EPC values of a sampling window to predict the future behavior of the program. We exploit this feature of the program execution behavior in order to accelerate micro-architectural simulation. In Section 3, we describe how metrics of interest can be obtained via sampling inside detected hotspots.

## 2.2. Hotspot detection

Hotspots are mainly characterized by the behavior of the branches inside the hotspot. A hotspot can be detected by keeping track of the branches being encountered during the execution of the program. To keep track of branches, we use a cache-like data structure called the Branch Behavior Buffer (*BBB*) [13]. Each branch has an entry in the *BBB*, consisting of an Execution Counter and a one-bit Candidate Flag (*CF*). The execution counter is incremented each time the branch is taken, and once the counter exceeds a certain threshold (512, in our case), the branch in question is marked as a candidate branch by setting the *CF* bit for that branch. The simulator also maintains a saturating counter called the hotspot detection counter (*HDC*), which keeps track of candidate branches. Initially, the counter is set to a maximum value (64 K in our case); each time a candidate branch is taken, the counter is decremented by a value  $D$ , and each time a non-candidate branch is taken, it is incremented by a value  $I$ . When the *HDC* decrements down to zero, we are in a hotspot. For our implementation we chose  $D$  as 2 and  $I$  as 1, such that exiting the hotspot is twice as slow as entering it (this is done to prevent false exits from a hotspot). The details of the hotspot detection scheme can be found in [13], [16].

## 3. HYBRID SIMULATION

As described previously, common programs exhibit high temporal locality in various sections of their code and behave in a predictable manner inside hotspots. Our strategy is to employ a two-level hybrid simulation paradigm, which can perform architectural simulation at two different levels of abstraction and with different speed/accuracy characteristics.

- A *low-level* simulation environment, which can perform cycle accurate simulation and provide accurate metrics associated with the program execution.
- A *high level* simulation environment, which can perform correct functional simulation without providing cycle accurate metrics.

Our chosen strategy is to achieve considerable speedup by exploiting the predictable behavior inside the hotspots. We employ the hotspot detection strategy described in [13] for determining the entry and exit points for a hotspot. We use the low-level simulation engine for the code outside a hotspot and for a fraction of the code executed inside a hotspot (also called the *sampling window*) and use high-level simulation for the remainder of the hotspot. We use the metrics acquired during the detailed simulation of the sampling window to estimate the metrics for the entire hotspot. To estimate the metrics for the entire hotspot, we use the metrics acquired during the detailed simulation of the sampling window. The time spent by a program inside a hotspot is dependent on the program itself and the specific input being used, but we have observed that the average fraction of time spent inside detected hotspots is 92%. Thus, accelerating simulation of the code inside the hotspots should provide considerable speedup for the entire benchmark.

For estimating the statistics associated to a hotspot, it is imperative to select a suitable sampling window. We will describe our proposed sampling techniques in the remainder of this section.

### 3.1. Flat sampling

This scheme corresponds to the sampling strategy employed in [16] and is illustrated in more detail in Figure 22-2(a). Once the program enters a hotspot, a fixed window of 128 K instructions is selected as the sampling window. The metrics collected in this window are used for estimating the metrics of the whole hotspot. This window is selected after skipping a warm-up window of 100 K instructions.

### 3.2. Convergence based sampling

The flat scheme blindly chooses a fixed window size and assumes that such a window will be enough for achieving convergence for power and performance inside all hotspots, across various applications. However, such an assumption is far from being valid. To account for these different behaviors, a convergence-based sampling scheme is proposed. In such a scheme, the simulator starts off in detailed mode and switched to fast mode only upon convergence. To check for convergence, a sampling window  $w$  is employed. Convergence is declared only if the metrics sampled in a number of consecutive windows of  $w$  instructions are within a threshold of  $p$  (the precision for convergence). If convergence is not satisfied, the window size  $w$  is increased and the process is repeated. There are two possible ways of increasing the window size  $w$ :

- Exponentially increasing window size. In this case, the current window size  $w$  is *doubled* if the convergence condition was not satisfied (possible window sizes are  $w, 2w, 4w, 8w, 16w, \dots$ ).

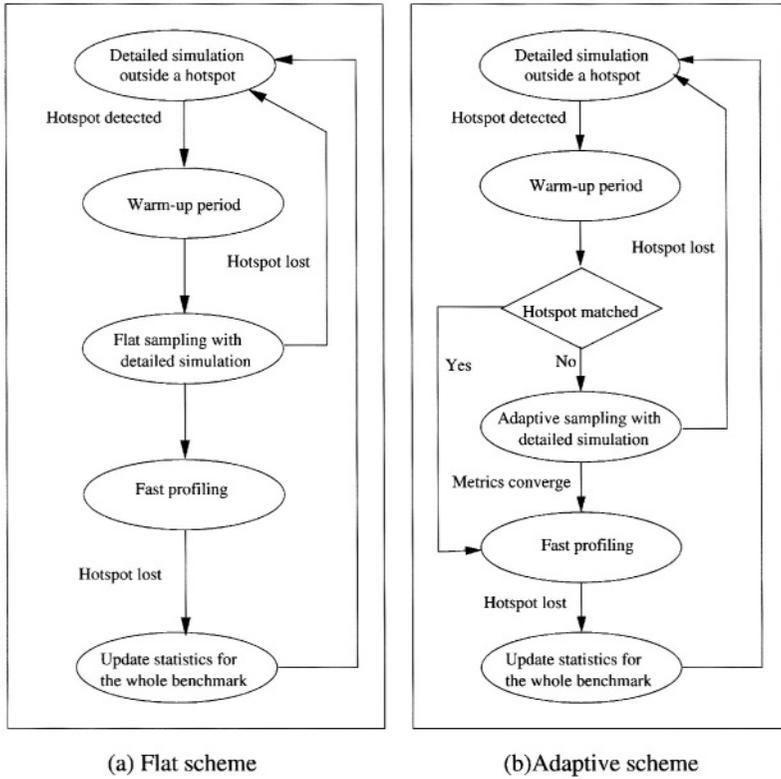


Figure 22-2. Hybrid simulation.

- Linearly increasing window sizes. In this case, the current window size is *incremented* by a fixed size of  $w$  if the convergence condition was not satisfied (possible window sizes are  $w, 2w, 3w, 4w, 5w, \dots$ ).

In our case, convergence is declared when metrics are maintained within a threshold  $p$  for 3 consecutive windows of size  $w$ . The exponential approach attains large window sizes in smaller number of iterations, so it starts with a smaller window size  $w$  of 2.5 K, while the linear approach starts with 128 K. Both the approaches can be used with three different threshold values for checking convergence (0.001, 0.0005 and 0.0001). The overall hybrid simulation strategy is illustrated in Figure 22-2(b).

As it can be expected, there is a trade-off between speed and accuracy in both cases. While an exponentially increasing window may suit some applications better, it may happen that the sampling window size is increasing too fast and fine-grain program phase changes may be missed. At the same time, a linearly increasing window size may prove inappropriate in cases where convergence is achieved only for large window sizes. To find a good compromise between the two, we have developed an adaptive sampling mech-

anism, which tries to identify fine-grain program phases, also called *micro-hotspots*.

### 3.3. Adaptive sampling

While the hotspot detection scheme makes sure that tightly coupled basic blocks that are executed together most of the time are detected, it does not ensure that the sequencing information is also maintained. For example, for the control flow graph in Figure 22-3(a), the set of basic blocks {A, B, C, D, E, F, G} is identified as being part of a hotspot. In effect, this means that the *occurrence probability* of any of these blocks is sufficiently high (related to the execution counter value in Section 2.2). However, second (or higher) order effects related to the sequencing information for these basic blocks are completely ignored. In fact, these effects determine whether a slower or faster increasing sampling window size should be used. For example, if during the execution of the hotspot, the basic blocks are executed as  $((AB^nC)^mADEGADFGADEG)^*$ ,<sup>1</sup> the sampling window size  $w$  should be directly related to the values of  $m$  and  $n$ . The *adaptive sampling scheme* tries to match the window size with the run-time sequencing information of the basic blocks. Such an application-adaptive window is called a *microhotspot*.

For the example shown in Figure 22-3(a), a possible series of candidate branches being executed in this typical hotspot is shown in Figure 22-3(b), where each letter represents the basic block corresponding to a candidate branch. In this example, the trace ABBCABBCADEGADFADEG represents the repeating microhotspot. We detect the microhotspot by keeping track of

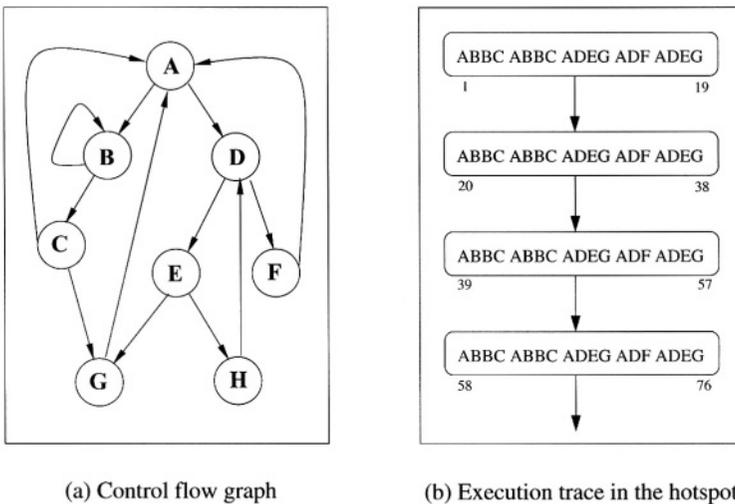


Figure 22-3. Microhotspot detection.

the most recent occurrence of every candidate branch. Whenever the difference between two consecutive occurrences of a candidate branch is larger than the current window size, a potential microhotspot is declared. The window size is changed to this new difference and the simulator checks this window for convergence. If the same branch is occurred again and the metrics of interest (EPC, IPC) have not converged yet, the window size is stretched to accommodate the new occurrence of the candidate branch. This process continues by checking for microhotspots for other candidate branches until convergence is achieved or the hotspot ends.

In practice, the simulator starts with an exponential convergence scheme with  $p = 0.0001$  and  $w = 2.5$  K. It first tries to achieve convergence using the exponential scheme until it encounters the first potential microhotspot, defined as the set of instructions between two occurrences of a candidate branch. Then the simulator continues with microhotspot detection and stops doubling the current window size. Once metrics of interest converge, the simulator switches to the fast mode of profiling. The detailed diagram for the adaptive sampling scheme is shown in Figure 22-2(b).

Identifying microhotspots has not only the advantage of being able to select the right size for the sampling window, but offers additional potential for speeding-up the simulation, as described next.

#### 4. PRACTICAL CONSIDERATIONS

To further speed-up the simulation, we also maintain a *monitor table* to reuse the information about earlier visited hotspots. This monitor table is similar to the one described in [13], and can be used to cache information and for determining whether the current hotspot has been visited earlier. The monitor table consists of entries corresponding to a unique hotspot. Each unique hotspot is identified by a unique number (*HSP\_id*) and it has its own characteristic *signature*. This signature is made up of the top seven most frequently executed branches after the hotspot period. The signature consists of the addresses of these branches, along with their corresponding frequencies. This entry also contains the necessary information required for estimating the statistics for a matched hotspot. It includes the average IPC and EPC values for the entire hotspot, along with the average per component EPC values. These are required to accurately determine per component energy values for the entire benchmark. These IPC and EPC values are the recorded values before the simulator switches to fast mode of profiling and after adaptive sampling converges.

Whenever the program enters a hotspot, the simulator tries to match the current hotspot with one of the entries in the monitor table both in terms of branch addresses and occurrence probability. At the end of the warm-up period, the simulator stores information about the most frequent branches of the present hotspot and tries to match the current hotspot with one of the

hotspots from the monitor table. This is done by comparing each of the *top five* branches of the current hotspot with the signature of each hotspot entry in the monitor table (we compare only the top five branches of the hotspot as we have observed that the frequencies of the candidate branches are very close in the sampling period and the least frequent two branches may change with different execution contexts of the hotspot).

The monitor table entries are used in conjunction with the hybrid simulation mechanism shown in Figure 22-2(b). The simulation starts in the detailed mode and continues to monitor branch behavior. Once a hotspot is detected, the simulator tries to match the current hotspot with one of the entries of the monitor table after the initial warm-up period. If the current hotspot is not matched, then the simulator tries to find an appropriate sampling window, which can be used for the estimation of various values. Once this window is determined through adaptive sampling, the simulator switches to the fast mode of profiling after recording the necessary values for estimation of power and performance. The simulator keeps track of the branch behavior and once the hotspot is lost, it reverts back to the detailed mode of simulation after updating the global power and performance metrics. If the hotspot is lost before adaptive sampling finds the required sampling window, the simulator waits until a new hotspot is detected and starts all over again. If the simulator finds a matching hotspot it directly switches over to the fast mode of profiling. The various parameters of the matching hotspot are used for updating the metrics after the current hotspot ends. If the current hotspot does not match any of the existing entries, it is added as a new entry into the monitor table after the hotspot is sampled partially or completely. This entry is created irrespective of whether the metrics had converged during adaptive sampling. Thus, hotspots that do not achieve convergence of metrics during adaptive sampling can be simulated in fast mode if they are encountered again.

The exponential and linear convergence based sampling schemes are similar and only differ in the way they check for convergence of metrics.

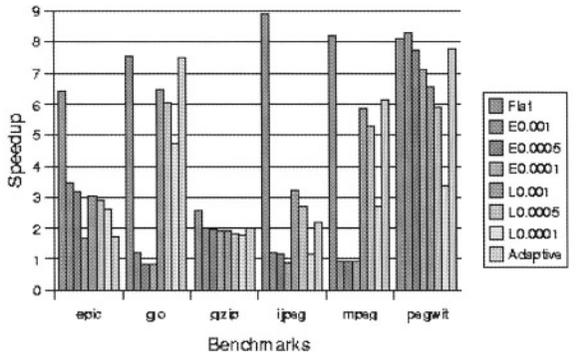
## 5. EXPERIMENTAL RESULTS

We have tested our hybrid simulation strategy on various benchmarks for both single instruction issue, in-order processors and for 4-way superscalar, out-of-order processors. We have compared the performance of the four schemes of hybrid simulation described in this chapter (flat sampling, linear and exponential window sampling, and adaptive sampling with a monitor table). Both configurations for the in-order, single instruction issue and 4-way, out-of-order superscalar processors assume a 512 K direct-mapped I-cache and a 1024 K 4-way set-associative D-cache. In case of the superscalar machine, the width of the pipeline and the number of ALUs is assumed to be four. In addition, a 32-entry Register Update Unit (RUU) and a 16-entry Load Store Queue (LSQ) are considered in this case. In both cases, we have used

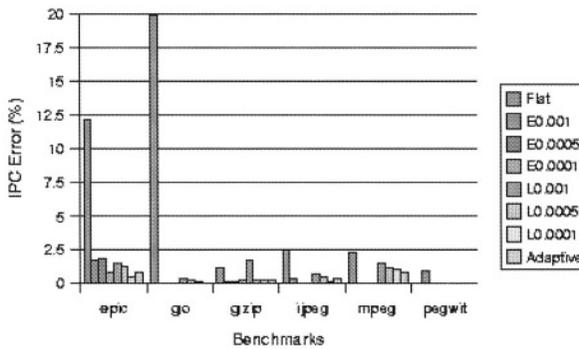
benchmarks from SpecInt95 (*jpeg*, *go*), SpecInt2000 (*gzip*) and MediaBench (*mpeg*, *pegwit*, and *epic*).

To assess the viability of our proposed hybrid simulation approach, the following set of experiments has been considered:

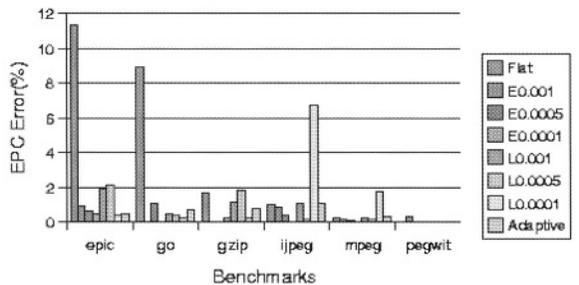
- \* The speed-up and accuracy for energy (EPC) and performance (IPC) metrics for the set of benchmarks under consideration.



(a) Speedup



(b) IPC Error

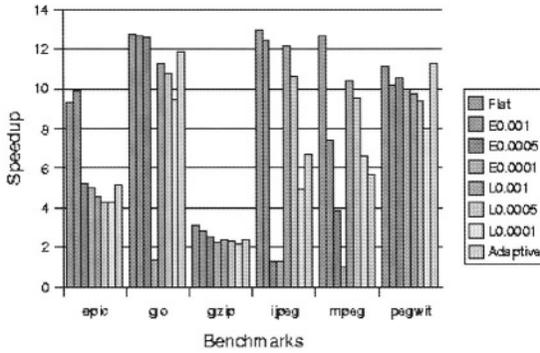


(c) EPC Error

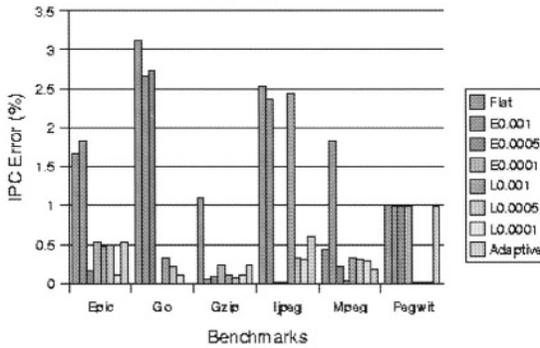
Figure 22-4. Results for 4-way, out-of-order processors.

- \* A comparative, detailed study of the flat and adaptive sampling schemes in terms of per module accuracy.
- \* A comparative study of all sampling schemes in terms of predicting the run-time profile of a given benchmark.

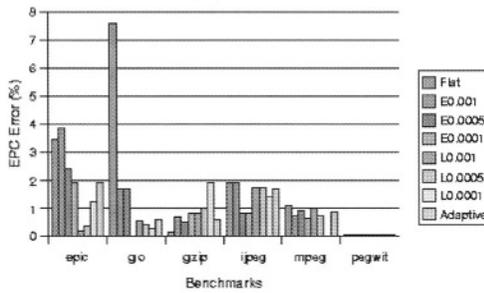
We show in Figures 22-4 and 22-5 our results for the accuracy and speed-up achieved for all sampling schemes proposed. The linear and exponential



(a) Speedup



(b) IPC Error



(c) EPC Error

Figure 22-5. Results for single instruction issue, in-order processors.

convergence schemes have been considered for three different precision thresholds: 0.001, 0.0005, and 0.0001 (E0.001, E0.0005, and E0.0001 for exponential and similar for the linear case). As it can be seen, the flat sampling case offers the highest speed-up, but at the expense of a very large error in some cases (e.g., 20% for IPC estimation in case of *go* or 11% error for EPC in case of *epic* when run on a 4-way, superscalar processor). While the exponential and linear convergence cases outperform the adaptive sampling scheme in some cases in terms of accuracy, they have unpredictable behavior, depending on the benchmark. The adaptive sampling scheme is the only one that offers consistent low error (less than 2% for both energy and performance estimates), irrespective of the type of processor being considered. The same cannot be said about the linear or exponential sampling schemes, showing their lack of adaptability to the application being run.

For the worst case achieved in the case of flat sampling scheme (i.e., benchmark *go* in case of single issue, in-order processors and *epic* in case of 4-way superscalar processors), we also show in Table 22-1 the estimation error for the energy cost per module. As it can be seen, adaptive sampling case (denoted by *adp* in Table 22-1) consistently produces results that are less than 2% away from the exact, cycle-accurate values, while the flat sampling (denoted by *flat* in Table 22-1) scheme can generate up to 18.25% error in some cases.

Finally, we have analyzed how the sampling schemes proposed in this chapter track the actual run-time profile of the application being run. We show in Figure 22-6 the estimated EPC values for the adaptive and flat sampling cases, compared to the original results provided by *Wattch*. As it can be seen, the adaptive case preserves the actual run-time profile much better (within 2% of the original), while the flat sampling results can be off by as much as 12%.

Table 22-1. Per component power errors.

Component	go flat	go adp	epic flat	epic adp
Rename	N/A	N/A	12.40%	0.64%
Bpred	18.25%	0.00%	6.05%	0.31%
Window	N/A	N/A	12.34%	0.81%
LSQ	N/A	N/A	7.91%	0.44%
Regfile	6.93%	0.00%	8.56%	0.58%
Icache	2.75%	0.00%	9.39%	0.53%
Dcache	3.21%	0.00%	11.58%	0.49%
Dcache2	2.36%	0.00%	2.44%	0.07%
ALU	0.05%	0.00%	7.63%	0.50%
Resultbus	9.80%	0.00%	12.00%	0.83%
Clock	18.25%	1.52%	14.25%	0.41%

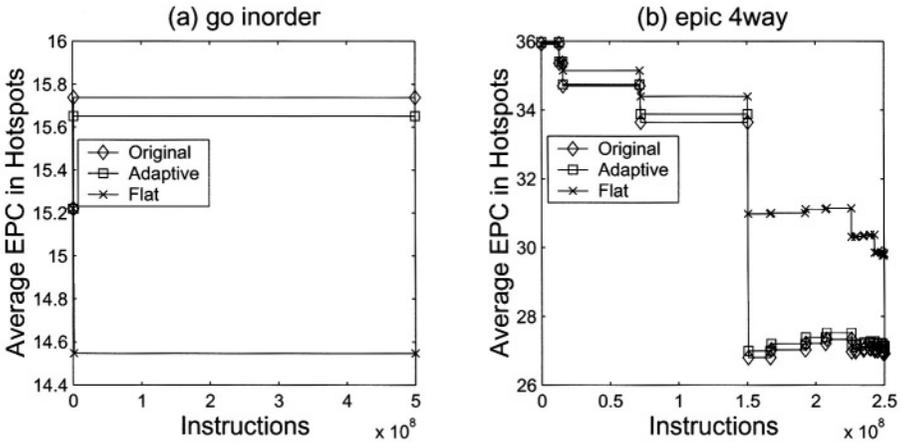


Figure 22-6. Comparison of the two schemes.

## 6. CONCLUSION

In this chapter we have presented a hybrid simulation strategy, which can accelerate the microarchitectural simulation without trading off accuracy. The strategy is based on the detection of hotspots and determining the exact sampling window size, which can be used for estimation of various metrics. We have also presented how these features can be exploited for simulation acceleration along with using a monitor table for reusing know information. By using these techniques we have been able to obtain substantial speedup, with negligible errors in IPC and EPC values. The proposed adaptive sampling scheme not only offers superior accuracy when compared to a simpler, flat sampling scheme, but also provides per module estimation error of less than 2% and faithfully tracks the run-time profile of the application under consideration.

## ACKNOWLEDGEMENT

This research has been supported in part by NSF Career Award CCR-008479.

## NOTE

<sup>1</sup> Classic notations from formal language theory have been used.

## REFERENCES

1. C. X. Huang, B. Zhang, A. C. Deng and B. Swirski. "The Design and Implementation of Powermill." In *Proceedings of International Workshop on Low Power Design*, pp. 105–110, April 1995.
2. D. Brooks, V. Tiwari and M. Martonosi. "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations." In *Proceedings of International Symposium on Computer Architecture*, pp. 83–94, Vancouver, BC, Canada, June 2000.
3. W. Ye, N. Vijaykrishnan, M. Kandemir and M. J. Irwin. "The Design and Use of SimplePower: A Cycle-Accurate Energy Estimation Tool." In *Proceedings of ACM/IEEE Design Automation Conference*, Los Angeles, CA, USA, June 2000.
4. C. Brandolese, W. Fornaciari, F. Salice and D. Sciuto. "An Instruction-Level Functionality-Based Energy Estimation Model for 32-bit Microprocessors." In *Proceedings of Design Automation Conference*, pp. 346–351, June 2000.
5. J. Russell and M. Jacome. "Software Power Estimation and Optimization for High-Performance 32-bit Embedded Processors," in *Proceedings of International Conference on Computer Design*, pp. 328–333, October 1998.
6. A. Sama, M. Balakrishnan and J. F. M. Theeuwens. "Speeding up Power Estimation of Embedded Software." In *Proceedings of International Symposium on Low Power Electronics and Design*, Rapallo, Italy, 2000.
7. V. Tiwari, S. Malik and A. Wolfe. "Power Analysis of Embedded Software: A First Step Towards Software Power Minimization." In *IEEE Transactions on VLSI Systems*, Vol. 2, No. 4, pp. 437–445, December 1994.
8. T. K. Tan, A. Raghunathan, G. Lakshminarayana and N. K. Jha. "High-Level Software Energy Macro-Modelling." In *Proceedings of ACM/IEEE Design Automation Conference*, Las Vegas, Nevada, USA, June 2001.
9. M. Lajolo, A. Raghunathan and S. Dey, "Efficient Power Co-Estimation Techniques for System-on-Chip Design," In *Proc. Design & Test Europe*, pp.27-34, March 2000.
10. G. Qu, N. Kawabe, K. Usami and M. Potkonjak. "Function-Level Power Estimation Methodology for Microprocessors." In *Proceedings of ACM/IEEE Design Automation Conference*, Los Angeles, CA, USA, June 2000.
11. R. Marculescu, D. Marculescu and M. Pedram, "Sequence Compaction for Power Estimation: Theory and Practice." In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 18, No. 7, pp. 973–993, July 1999.
12. D. Burger and T. M. Austin. "The SimpleScalar Tool Set, Version 2.0." In *Computer Architecture News*, pp. 13-25, June 1997.
13. Matthew C. Merten, Andrew R. Trick, Christopher N. George, John C. Gyllenhaal, and Wen-mei W. Hwu. "A Hardware-Driven Profiling Scheme for Identifying Program Hot Spots to Support Runtime Optimization." In *Proceedings of International Symposium on Computer Architecture*, pp. 136–147, May, 1999.
14. M. Sami, D. Sciuto, C. Silvano and V. Zaccaria, "Instruction-Level Power Estimation for Embedded VLIW Cores." In *Proceedings of International Workshop of Hardware/Software Codesign*, pp. 34-37, March 2000.
15. A. Sinha and A. P. Chandrakasan, "JouleTrack – A Web Based Tool for Software Energy Profiling," In *Proceedings of ACM/IEEE Design Automation Conference*, Las Vegas, Nevada, USA, June 2001.
16. D. Marculescu and A. Iyer, "Application-Driven Processor Design Exploration for Power-Performance Trade-off Analysis," In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, San Jose, USA, Nov. 2001.

*This page intentionally left blank*

## Chapter 23

# DYNAMIC PARALLELIZATION OF ARRAY BASED ON-CHIP MULTIPROCESSOR APPLICATIONS

M. Kandemir<sup>1</sup>, W. Zhang<sup>1</sup> and M. Karakoy<sup>2</sup>

<sup>1</sup> *CSE Department, The Pennsylvania State University, University Park, PA 16802, USA;*

<sup>2</sup> *Department of Computing, Imperial College, London SW7 2AZ, UK;*

*E-mail: {kandemir,wzhang}@cse.psu.edu, m.karakoy@ic.ac.uk*

**Abstract.** Chip multiprocessing (or multiprocessor system-on-a-chip) is a technique that combines two or more processor cores on a single piece of silicon to enhance computing performance. An important problem to be addressed in executing applications on an on-chip multiprocessor environment is to select the most suitable number of processors to use for a given objective function (e.g., minimizing execution time or energy-delay product) under multiple constraints. Previous research proposed an ILP-based solution to this problem that is based on exhaustive evaluation of each nest under all possible processor sizes. In this study, we take a different approach and propose a pure runtime strategy for determining the best number of processors to use at runtime. Our experiments show that the proposed approach is successful in practice.

**Key words:** code parallelization, array-based applications, on-chip multiprocessing

## 1. INTRODUCTION

Researchers agree that chip multiprocessing is the next big thing in CPU design [4]. The performance improvements obtained through better process technology, better micro-architectures and better compilers seem to saturate; in this respect, on-chip multiprocessing provides an important alternative for obtaining better performance/energy behavior than what is achievable using current superscalar and VLIW architectures.

An important problem to be addressed in executing applications on an on-chip multiprocessor environment is to select the most suitable number of processors to use. This is because in most cases using all available processors to execute a given code segment may not be the best choice. There might be several reasons for that. First, in cases where loop bounds are very small, parallelization may not be a good idea at the first place as creating individual threads of control and synchronizing them may itself take significant amount of time, offsetting the benefits from parallelization. Second, in many cases, data dependences impose some restrictions on parallel execution of loop iterations; in such cases, the best results may be obtained by using a specific

number of processors. Third, data communication/synchronization between concurrently executing processors can easily offset the benefits coming from parallel execution.

As a result, optimizing compiler community invested some effort on determining the most suitable number of processors to use in executing loop nests (e.g., [7]). In the area of embedded computing, the situation is even more challenging. This is because, unlike traditional high-end computing, in embedded computing one might have different parameters (i.e., objective functions) to optimize. For example, a strategy may try to optimize energy consumption under an execution time bound. Another strategy may try to reduce the size of the generated code under both energy and performance constraints. Consequently, selecting the most appropriate number of processors to use becomes a much more challenging problem. On top of this, if the application being optimized consists of multiple loop nests, each loop nest can demand a different number of processors to generate the best result. Note that if we use fewer number of processors (than available) to execute a program fragment, the unused processors can be turned off to save energy.

Previous research (e.g., [6]) proposed a solution to this problem that is based on exhaustive evaluation of each nest under all possible processor sizes. Then, an integer linear programming (ILP) based approach was used to determine the most suitable number of processors for each nest under a given objective function and multiple energy/performance constraints. This approach has three major drawbacks. First, exhaustively evaluating each alternative processor size for each loop can be very time consuming. Second, since all evaluations are performed statically, it is impossible to take runtime-specific constraints into account (e.g., a variable whose value is known only at runtime). Third, it is not portable across different on-chip multiprocessor platforms. Specifically, since all evaluations are done under specific system parameters, moving to a different hardware would necessitate repeating the entire process.

In this study, we take a different approach and propose a pure runtime strategy for determining the best number of processors to use at runtime. The idea is, for each loop nest, to use the first couple of iterations of the loop to determine the best number of processors to use, and when this number is found, execute the remaining iterations using this size. This approach spends some extra cycles and energy at runtime to determine the best number of processors to use; however, this overhead is expected to be compensated for when the remaining iterations are executed. This is particularly true for many image/video applications in embedded domain where multiple loops with large iteration counts operate on large images/video sequences. The main advantage of this approach is that it requires little help from compiler and it can take runtime parameters into account.

The remainder of this study discusses our approach in detail and presents experimental data demonstrating its effectiveness. Our experiments clearly show that the proposed approach is successful in practice and adapts well to

runtime conditions. Our results also indicate that the extra cycles/energy expended in determining the best number of processor are more than compensated for by the speedup obtained in executing the remaining iterations. We are aware of previous research that used runtime adaptability for scaling on-chip memory/cache sizes (e.g., [1]) and issue widths (e.g., [5]) among other processor resources. However, to the best of our knowledge, this is the first study that employs runtime resource adaptability for on-chip multiprocessors. Recently, there have been several efforts for obtaining accurate energy behavior for on-chip multiprocessing and communication (e.g., see [3] and the references therein). These studies are complementary to the approach discussed here, and our work can benefit from accurate energy estimations for multiprocessor architectures.

Section 1.2 presents our on-chip multiprocessor architecture and gives the outline of our execution and parallelization strategy. Section 1.3 explains our runtime parallelization approach in detail. Section 1.4 introduces our simulation environment and presents experimental data. Section 1.5 presents our concluding remarks.

## 2. ON-CHIP MULTIPROCESSOR AND CODE PARALLELIZATION

Figure 23-1 shows the important components of the architecture assumed in this research. Each processor is equipped with data and instruction caches and can operate independently; i.e., it does not need to synchronize its execution with those of other processors unless it is necessary. There is also a global (shared) memory through which all data communication is performed. Also, processors can use the shared bus to synchronize with each other when such a synchronization is required. In addition to the components shown in this figure, the on-chip multiprocessor also accommodates special-purpose circuitry, clocking circuitry, and I/O devices. In this work, we focus on processors, data and instruction caches, and the shared memory.

The scope of our work is array-intensive embedded applications. Such applications frequently occur in image and video processing. An important characteristic of these applications is that they are loop-based; that is, they are composed of a series of loop nests operating on large arrays of signals. In many cases, their access patterns can be statically analyzed by a compiler and modified for improved data locality and parallelism. An important

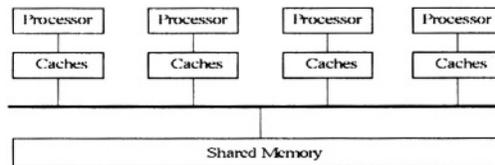


Figure 23-1. Relevant parts of our on-chip multiprocessor.

advantage of on-chip multiprocessing from the software perspective is that such an architecture is very well suited for high-level parallelism; that is, the parallelism that can be exploited at the source level (loop level) using an optimizing compiler. In contrast, the parallelism that can be exploited by single processor superscalar and VLIW machines are low level (instruction level). Previous compiler research from scientific community reveals that array-intensive applications can be best optimized using loop-level parallelization techniques [9]. Therefore, we believe that array-intensive applications can get the most benefit from an on-chip multiprocessor.

An array-intensive embedded application can be executed on this architecture by parallelizing its loops. Specifically, each loop is parallelized such that its iterations are distributed across processors. An effective parallelization strategy should minimize the inter-processor data communication and synchronization.

There are different ways of parallelizing a given loop nest (see Wolfe's book [9]). A parallelization strategy can be oriented to exploiting the highest degree of parallelism (i.e., parallelizing as many loops as possible in a given nest), achieving load balance (i.e., minimizing the idle processor time), achieving good data locality (i.e., making effective use of data cache), or a combination of these. Since we are focusing on embedded applications, the objective functions that we consider are different from those considered in general purpose parallelization. Specifically, we focus on the following type of compilation strategies:

$$\text{minimize } f(E_1, E_2, \dots, E_k, X) \text{ under } g_i(E_1, E_2, \dots, E_k, X)$$

Here,  $k$  is the number of components we consider (e.g., processor, caches, memory).  $E_j$  is the energy consumption for the  $j$ th component and  $X$  is the execution time.  $f(\cdot)$  is the objective function for the compilation and each  $g_i(\cdot)$  where  $1 \leq i \leq C$  denotes a constraint to be satisfied by the generated output code. Classical compilation objectives such as minimizing execution time or minimizing total energy consumption can easily be fit into this generic compilation strategy. Table 23-1 shows how several compilation strategies can be expressed using this approach. Since we parallelize each loop nest in isolation, we apply the compilation strategy given above to each nest separately.

Table 23-1. Different compilation strategies (note that this is not an exhaustive list).

Minimize execution time	$\min(X)$
Minimize total energy	$\min(\sum E_j)$
Minimize energy of component $i$ under performance constraint	$\min(E_i)$ under $X \leq X_{max}$
Minimize execution time under energy constraint for component $i$	$\min(X)$ under $E_i = E_{max}$
Minimize energy-delay product	$\min(X \sum E_j)$

### 3. RUNTIME PARALLELIZATION

#### 3.1. Approach

Let  $I$  be the set of iterations for a given nest that we want to parallelize. We use  $I' \in I$  (a subset of  $I$ ) for determining the number of processors to use in executing the iterations in set  $I - I'$ . The set  $I'$ , called the training set, should be very small in size as compared to the iteration set,  $I$ . Suppose that we have  $K$  different processor sizes. In this case, the training set is divided into  $K$  subsets, each of which containing  $I'/K$  iterations (assuming that  $K$  divides  $I'$  evenly).

Let  $f_k$  denote the processor size used for the  $k$ th trial, where  $1 = k = K$ . We use  $T_{f_k}(J)$  to express the execution time of executing a set of iterations denoted by  $J$  using  $f_k$ . Now, the original execution time of a loop with iteration set  $I$  using a single processor can be expressed as  $T_1(I)$ . Applying our training-based runtime strategy gives an execution time of

$$T_{all} = \sum_{k=1}^K T_{f_k}(I'/K) + T_{f_{kbest}}(I - I')$$

In this expression, the first component gives the training time and the second component gives the execution time of the remaining iterations with the best number of processors selected by the training phase (denoted  $f_{kbest}$ ). Consequently,  $T_{f_{kbest}}(I) - T_{all}$  is the extra time incurred by our approach compared to the best execution time possible. If successful, our approach reduces this difference to minimum.

Our approach can also be used for objectives other than minimizing execution time. For example, we can try to minimize energy consumption or energy-delay product. As an example, let  $E_{f_k}(J)$  be the energy consumption of executing a set of iterations denoted by  $J$  using  $f_k$  processors. In this case,  $E_1(I)$  gives the energy consumption of the nest with iteration set  $I$  when a single processor is used. On the other hand, applying our strategy gives an energy consumption of

$$E_{all} = \sum_{k=1}^K T_{f_k}(I'/K) + T_{f_{kbest}}(I - I')$$

The second component in this expression gives the energy consumption of the remaining iterations with the best number of processors (denoted  $f_{kbest}$ ) found in the training phase. Note that in general  $f_{kbest}$  can be different from  $f_{kbest}$ .

Figure 23-2 illustrates this training period based loop parallelization strategy. It should also be noted that we are making an important assumption here. We are assuming that the best processor size does not change during the execution of the entire loop. Our experience with array-intensive embedded applications indicates that in majority of the cases, this assumption is valid. However, there exist also cases where it is not. In such cases, our approach can be slightly modified as follows. Instead of having only a single training

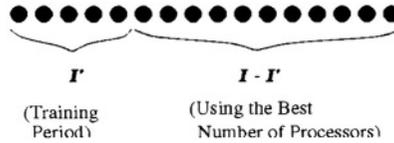


Figure 23-2. Parallelization based on training. Each dot represents an iteration.

period at the beginning of the loop, we can have multiple training periods interspersed across loop execution. This allows the compiler to tune the number of processors at regular intervals, taking into account the dynamic variations during loop execution. The idea is discussed in more detail in Section 1.3.0.

An important parameter in our approach is the size of the training set,  $I'$ . Obviously, if we have  $K$  different processor sizes, then the size of  $I'$  should be at least  $K$ . In general, larger the size of  $I'$ , better estimates we can derive (as we can capture the impact of cross-loop data dependences). However, as the size of  $I'$  gets larger, so does the time spent in training (note that we do not want to spend too many cycles/too much energy during the training period). Therefore, there should be an optimum size for  $I'$ , and this size depends strongly on the loop nest being optimized.

### 3.2. Hardware support

In order to evaluate a given number of processors during the training period, we need some help from the hardware. Typically, the constraints that we focus on involve both performance and energy. Consequently, we need to be able to get a quick estimation of both execution cycles and energy consumption at runtime (during execution). To do this, we assume that the hardware provides a set of performance counters.

In our approach, we employ these performance counters for two different purposes. First, they are used to estimate the performance for a given number of processors. For example, by sampling the counter that holds the number of cycles at the end of each trial (during the training period), our approach can compare the performances of different processor sizes. Second, we exploit these counters to estimate the energy consumption on different components of the architecture. More specifically, we obtain from these counters the number of accesses (for a given processor size) to the components of interest, and multiply these numbers by the corresponding per access energy costs. For instance, by obtaining the number of L1 hits and misses and using per access and per miss energy costs, the compiler calculates the energy spent in L1 hits and misses. To sum up, we adopt an activity based energy calculation strategy using the hardware counters available in the architecture.

### 3.3. Compiler support

In order to experiment with different processor sizes (in the training period), we need to generate different versions of each loop at compile time. Our compiler takes the input code, constraints, and the objective function as input and generates the transformed code as output.

### 3.4. Overheads

When our approach is employed, there are several overheads that will be incurred at runtime. In this subsection, we discuss these overheads. Later in Section 1.4, we quantify these overheads for our benchmark programs.

#### 3.4.1. Sampling counters

The architectures that provide performance counters also provide special instructions to read (sample) and initialize them. We assume the existence of such instructions that can be invoked from both C and assembly codes. Executing these instructions consumes both energy and execution cycles in processor datapath, instruction cache, and memory.

#### 3.4.2. Performance calculations

As mentioned earlier, after trying each processor size, we need to compute the objective function and constraints. Note that this calculation needs to be done at runtime. To reduce the overhead of these computations, we first calculate constraints since if any of the constraints is not satisfied we do not need to compute the objective function. After all trials have been done, we compare the values of the objective functions (across all processor sizes experimented) and select the best number of processors. Our implementation also accounts for energy and execution cycle costs of these runtime calculations.

#### 3.4.3. Activating/deactivating processors

During the training phase we execute loop iterations using different number of processors. Since re-activating a processor which is not in active state takes some amount of time as well as energy, we start trying different number of processors with the highest number of processors (i.e., the largest possible processor size). This helps us avoid processor re-activation during the training period. However, when we exit the training period and move to start executing the remaining iterations with the best number of processors, we might need to re-activate some processors. An alternative approach would be not turning off processors during the training period. In this way, no processor re-activation is necessary; the downside is some extra energy consumption. It

should be noted, however, when we move from one nest to another, we might still need to re-activate some processors as different nests might demand different processor sizes for the best results.

#### 3.4.4. Locality issues

Training periods might have another negative impact on performance too. Since the contents of data caches are mainly determined by the number of processors used and their access patterns, frequently changing the number of processors used for executing the loop can distort data cache locality. For example, at the end of the training period, one of the caches (the one whose corresponding processor is used to measure the performance and energy behavior in the case of one processor) will keep most of the current working set. If the remaining iterations need to reuse these data, they need to be transferred to their caches, during which data locality may be poor.

### 3.5. Conservative training

So far, we have assumed that there is only a single training period for each nest during which all processor sizes are tried. In some applications, however, the data access pattern and performance behavior can change during the loop execution. If this happens, then the best number of processors determined using the training period may not be valid anymore. Our solution is to employ multiple training sessions. In other words, from time to time (i.e., at regular intervals), we run a training session and determine the number of processors to use until the next training session arrives. This strategy is called the conservative training (as we conservatively assume that the best number of processors can change during the execution of the nest).

If minimizing execution time is our objective, under conservative training, the total execution time of a given nest is

$$T_{all} = \sum_{m=1}^M \left[ \sum_{k=1}^K T_{f_{m,k}}(I'_{m,k}) + T_{f_{m,best}}(I''_m) \right]$$

Here, we assumed a total of  $M$  training periods. In this formulation,  $f_{m,k}$  is the number of processors tried in the  $k$ th trial of the  $m$ th training period (where  $1 \leq m \leq M$ ).  $I'_{m,k}$  is the set of iterations used in the  $k$ th trial of the  $m$ th training period and  $I''_m$  is the set of iterations executed with the best number of processors (denoted  $f_{m,best}$ ) determined by the  $m$ th training period. It should be observed that

$$\sum_{m=1}^M \sum_{k=1}^K I'_{m,k} + \sum_{m=1}^M I''_m = I'$$

where  $I$  is the set of iterations in the nest in question. Similar formulations can be given for energy consumption and energy-delay product as well. When we are using conservative training, there is an optimization the compiler can

apply. If two successive training periods generate the same number (as the best number of processors to use), we can optimistically assume that the loop access pattern is stabilized (and the best number of processors will not change anymore) and execute the remaining loop iterations using that number. Since a straightforward application of conservative training can have a significant energy and performance overhead, this optimization should be applied with care.

### 3.6. Exploiting history information

In many array-intensive applications from the embedded image/video processing domain, a given nest is visited multiple times. Consider an example scenario where  $L$  different nests are accessed within an outermost loop (e.g., a timing loop that iterates a fixed number of iterations and/or until a condition is satisfied). In most of these cases, the best processor size determined for a given nest in one visit is still valid in subsequent visits. That is, we may not need to run the training period in these visits. In other words, by utilizing the past history information, we can eliminate most of the overheads due to running the training periods.

## 4. EXPERIMENTS

### 4.1. Benchmarks and simulation platform

To evaluate our runtime parallelization strategy, we performed experiments using a custom experimental platform. Our platform has two major components: an optimizing compiler and a cycle-accurate simulator. Our compiler takes a sequential program, compilation constraints, an objective function, and generates a transformed program with explicit parallel loops. The simulator takes as input this transformed code and simulates parallel execution. For each processor, four components are simulated: processor, instruction cache, data cache, and the shared memory. When a processor is not used in executing a loop nest, we shut off that processor and its data and instruction caches to save leakage energy. To obtain the dynamic energy consumption in a processor, we used SimplePower, a cycle-accurate energy simulator [10]. SimplePower simulates a simple, five-stage pipelined architecture and captures the switching activity on a cycle-by-cycle basis. Its accuracy has been validated to be within around 9% of a commercial embedded processor. To obtain the dynamic energy consumptions in instruction cache, data cache, and the shared memory, we use the CACTI framework [8]. We assumed that the leakage energy per cycle of an entire cache is equal to the dynamic energy consumed per access to the same cache. This assumption tries to capture the anticipated importance of leakage energy in the future as leakage becomes the dominant part of energy consumption for 0.10 micron (and below) technolo-

gies for the typical internal junction temperatures in a chip [2]. We assumed a 20 msec resynchronization latency (a conservative estimate) to bring a turned off processor (and its caches) into the full operational mode. We also assumed a simple synchronization mechanism where each processor updates a bit in a globally shared and protected location. When a processor updates its bit, it waits for the other processors to update their bits. When all processor update their bits, all processors continue with their parallel execution. Our energy calculation also includes the energy consumed by the processors during synchronization.

For each processor, both data and instruction caches are 8 KB, 2-way set-associative with 32 byte blocks and an access latency of 1 cycle. The on-chip shared memory is assumed to be 1 MB with a 12 cycle access latency. All our energy values are obtained using the 0.1-micron process technology. The cache hit and miss statistics are collected at runtime using the performance counters in a Sun Spare machine, where all our simulations have been performed.

We used a set of eight benchmarks to quantify the benefits due to our runtime parallelization strategy, *Img* is an image convolution application. *Cholesky* is a Cholesky decomposition program. *Atr* is a network address translation application. *SP* computes the all-nodes shortest paths on a given graph. *Encr* has two modules. The first module generates a cryptographically-secure digital signature for each outgoing packet in a network architecture. The second one checks the authenticity of a digital signature attached to an incoming message. *Hyper* simulates the communication activity in a distributed-memory parallel architecture. *wood* implements a color-based visual surface inspection method. Finally, *Usonic* is a feature-based object estimation algorithm. The important characteristics of these benchmarks are given in Table 23-2. The last two columns give the execution cycles and the energy consumptions (in the processor core, caches, and the shared memory) when our applications are executed using a single processor.

Table 23-2. Benchmark codes used in the experiments and their important characteristics.

Benchmark	Input	Cycles	Energy
<i>Img</i>	263.1 KB	16526121	18.51 mJ
<i>Cholegky</i>	526.0 KB	42233738	53.34 mJ
<i>Atr</i>	382.4 KB	19689554	24.88 mJ
<i>SP</i>	688.4 KB	268932537	91.11 mJ
<i>Encr</i>	443.6 KB	28334189	72.55 mJ
<i>Hyper</i>	451.1 KB	25754072	66.80 mJ
<i>Wood</i>	727.6 KB	362069719	90.04 mJ
<i>Usonic</i>	292.9 KB	9517606	22.93 mJ

### 4.2. Results

As discussed earlier, our approach can accommodate different compilation strategies. In most of the results we report here, we focus on energy-delay product. What we mean by “energy” here is sum of the energy consumptions in the CPU, instruction and data caches, and the shared memory. Also, what we mean by “delay” is the execution cycles taken by the application. Figure 23-3 gives the energy-delay products for four different versions. The first version (denoted  $|P| = 8$ ) is the result obtained when 8 processors are used for each nest in the application. The second version (denoted Best  $|P|$  (Application-Wide)) is the result obtained when the best number of processors is used for each application. Note, however, that all nests execute using the same processor size. The third version (denoted Best  $|P|$  (Nest-Based)) uses the best number of processors for each nest and the fourth version (denoted Runtime) is the strategy discussed in this study. The difference between the last two versions is that the third version runs the entire nest using the best number of processors (for that nest); that is, it does not have a training period (thus, it represents the best scenario). All the bars in Figure 23-3 represent energy-delay values given as a fraction of the energy-delay product of the base case where each nest is executed using a single processor (see the last column of Table 23-2). In obtaining the results with our version, for each nest, the size of the iterations used in training was 10% of the total number of iterations of the nest. These results indicate that average reductions provided by the  $|P| = 8$ , Best  $|P|$  (Application-Wide), Best  $|P|$  (Nest-Based), and Runtime versions are 31.25%, 43.87%, 63.87%, and 59.50%, respectively, indicating that the extra overheads incurred by our approach over the third version are not too much. That is, our runtime strategy performs very well in practice.

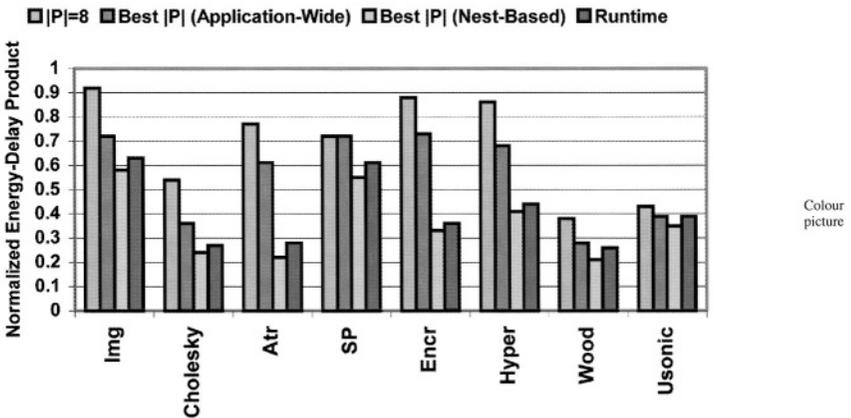


Figure 23-3. Normalized energy-delay products.

We also measured the overhead breakdown for our approach. The overheads are divided into three portions: the overheads due to sampling the counters, the overheads due to the calculations for computing the objective function, checking constraints, and selecting the best number of processors (denoted Calculations in the figure), and the overheads due to processor reactivation (i.e., transitioning a processor from the inactive state to the active state). We observed that the contributions of these overheads are 4.11%, 81.66%, and 14.23%, respectively. In other words, most of the overhead is due to the (objective function + constraint evaluation) calculations performed at runtime. The last component of our overheads (i.e., locality-related one) is difficult to isolate and is incorporated as part of the execution time/energy.

The results reported so far have been obtained by setting the number of iterations used in training to 10% of the total number of iterations in the nest. To study the impact of the size of the training set, we performed a set of experiments where we modified the number of iterations used in the training period. The results shown in Figure 23-4 indicate that for each benchmark there exists a value (for the training iterations) that generates the best result. Working with very small size can magnify the small variations between iterations and might prevent us from detecting the best number of processors accurately. At the other extreme, using a very large size makes sure that we find the best number of processors; however, we also waste too much time/energy during the training phase itself.

To investigate the influence of conservative training (Section 1.3.0) and history based training (Section 1.3.0), we focus on two applications: *Atr* and *Hyper* (both of these applications have nine nests). In *Atr*, in some nests, there are some significant variations (load imbalances) between loop iterations. Therefore, it is a suitable candidate for conservative training. The first two bars in Figure 23-5 show, for each nest, the normalized energy-delay product due to our strategy without and with conservative training, respectively. We see that the 3rd, 4th, 5th, and 9th nests take advantage of conservative training,

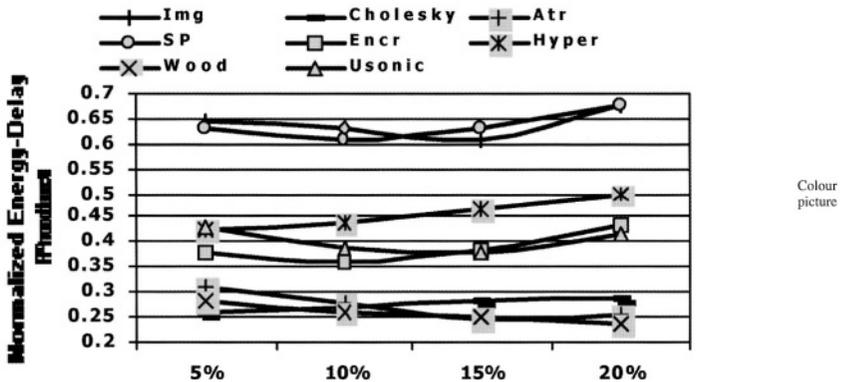


Figure 23-4. Impact of the number of iterations used in training period.

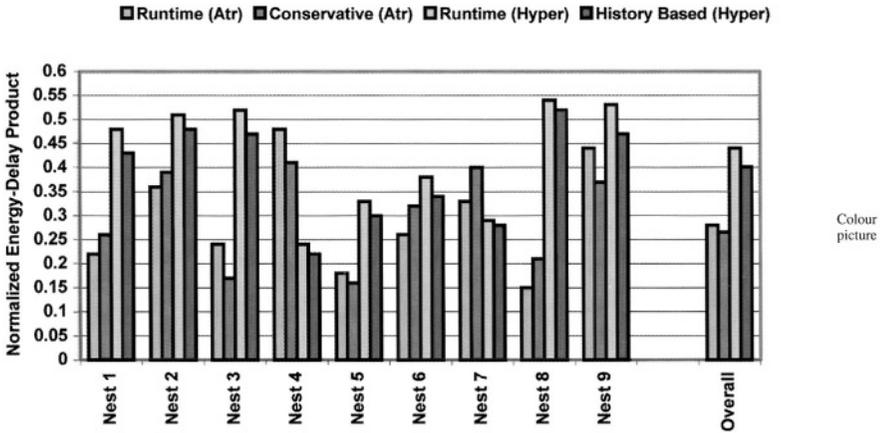


Figure 23-5. Influence of conservative training (Atr) and history based training (Hyper).

whereas the remaining nests do not. Since the third nest of this application is the one with the largest execution time, it dominates the overall energy-delay product, leading to some gain when the entire application is considered. Hyper, on the other hand, benefits from the history based training. The last two bars Figure 23-5 show, for each nest, the normalized energy-delay product due to our strategy without and with past history information. Recall from Section 1.3.0 that when the history information is used, we can eliminate the training overheads in successive visits. The results shown in the figure clearly indicate that all nests in this application take advantage of history information, resulting in a 9% improvement in the energy-delay product as compared to our strategy without history information.

### 5. CONCLUDING REMARKS

On-chip multiprocessing is an attempt to speedup applications by exploiting inherent parallelism in them. In this study, we have made three major contributions. First, we have presented a runtime loop parallelization strategy for on-chip multiprocessors. This strategy uses the initial iterations of a given loop to determine the best number of processors to employ in executing the remaining iterations. Second, we have quantified benefits of our strategy using a simulation environment and showed how its behavior can be improved by being more aggressive in determining the best number of processors. Third, we have demonstrated that the overheads associated with our approach can be reduced using past history information about loop executions.

**REFERENCES**

1. R. Balasubramonian et al. "Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures." In *Proceedings of 33rd International Symposium on Micro-architecture*, pp. 245–257, December 2000.
2. A. Chandrakasan et al. "Design of High-Performance Microprocessor Circuits." *IEEE Press*, 2001.
3. DAC'02 Sessions: "Design Methodologies Meet Network Applications" and "System on Chip Design", New Orleans, LA, June 2002.
4. <http://industry.java.sun.com/javaneWS/stories/story2/0,1072,32080,00.html>
5. A. Iyer and D. Marculescu. "Power-Aware Micro-architecture Resource Scaling." In *Proceedings of IEEE Design, Automation, and Test in Europe Conference*, Munich, Germany, March 2001.
6. I. Kadayif et al. "An Integer Linear Programming Based Approach for Parallelizing Applications in On-Chip Multiprocessors." In *Proceedings of Design Automation Conference*, New Orleans, LA, June 2002.
7. K. S. McKinley. *Automatic and Interactive Parallelization*, PhD Dissertation, Rice University, Houston, TX, April 1992.
8. S. Wilton and N. P. Jouppi. "CACTI: An Enhanced Cycle Access and Cycle Time Model." *IEEE Journal of Solid-State Circuits*, pp. 677–687, 1996.
9. M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1996.
10. W. Ye et al. "The Design and Use of SimplePower: A Cycle-Accurate Energy Estimation Tool." In *Proceedings of the 37th Design Automation Conference*, Los Angeles, CA, June, 2000.

## Chapter 24

# SDRAM-ENERGY-AWARE MEMORY ALLOCATION FOR DYNAMIC MULTI-MEDIA APPLICATIONS ON MULTI-PROCESSOR PLATFORMS

P. Marchal<sup>1</sup>, J. I. Gomez<sup>2</sup>, D. Bruni<sup>3</sup>, L. Benini<sup>3</sup>, L. Piñuel<sup>2</sup>, F. Catthoor<sup>1</sup>, and H. Corporaal<sup>4</sup>

<sup>1</sup> IMEC and K.U.Leuven-ESAT, Leuven, Belgium; <sup>2</sup> DACYA U.C.M., Spain; <sup>3</sup> D.E.I.S. University of Bologna, Italy; <sup>4</sup> IMEC and T.U. Eindhoven, Nederland

**Abstract.** Heterogeneous multi-processors platforms are an interesting option to satisfy the computational performance of future multi-media applications. Energy-aware data management is crucial to obtain reasonably low energy consumption on the platform. In this paper we show that the assignment of data of dynamically created/deleted tasks to the shared main memory has a large impact on platform's energy consumption. We introduce two dynamic memory allocators, which optimize the data assignment in the context of shared multi-banked memories.

**Key words:** dynamic multi-media applications, data-assignment, data locality, SDRAM, multi-processor

## 1. INTRODUCTION

In the near future, the silicon market will be driven by low-cost, portable consumer devices, which integrate multi-media and wireless technology. Applications running on these devices require an enormous computational performance (1–40GOPS) at low energy consumption (0.1–2W). Heterogeneous multi-processor platforms potentially offer enough computational performance. However, energy-aware memory management techniques are indispensable to obtain sufficiently low energy consumption. In this paper, we focus on the energy consumption of large off-chip multi-banked memories (e.g. SDRAMs) used as main memory on the platform. They contribute significantly to the system's energy consumption [1]. Their energy consumption depends largely on how data is assigned to the memory banks. Due to the interaction of the user with the system, which data needs to be allocated is only known at run-time. Therefore, fully design-time based solutions as proposed earlier in the compiler and system synthesis cannot solve the problem (see Section 4). Run-time memory management solutions as present in nowadays operating systems are too inefficient in terms of cost optimization (especially energy consumption). We present two SDRAM-energy-aware

memory allocators for dynamic multi-tasked applications. Both allocators reduce the energy consumption compared with the best known approach so far.

## 2. PLATFORM AND SDRAM ENERGY MODEL

On our platform, each processor is connected to a local memory and interacts with shared off-chip SDRAM modules. The SDRAMs are present on the platform because their energy cost per bit to store large data structures is lower than of SRAMs. They are used to store large infrequently accessed data structures. As a consequence, they can be shared among processors to reduce the static energy cost without a large performance penalty.

A simplified view of a typical multi-banked SDRAM architecture is shown in Figure 24-1. Fetching or storing data in an SDRAM involves three memory operations. An activation operation selects the appropriate bank and moves a page/row to the page buffer of the corresponding bank. After a page is opened, a read/write operation moves data to/from the output pins of the SDRAM. Only one bank can use the output pins at the time. When the next read/write accesses hit in the same page, the memory controller does not need to activate the page again (a page hit). However, when another page is needed (a page miss), pre-charging the bank is needed first. Only thereafter the new page can be activated and the data can be read.

SDRAMs nowadays support several energy states: standby mode (STBY), clock-suspend mode (CS) and power down (PWDN). We model the timing behavior of the SDRAM memory with a state-machine similar to [10]. The timing and energy parameters of the different state transitions have been derived from Micron 64 Mb mobile SDRAM [3]. The energy consumption of the SDRAM is computed with the following formula:

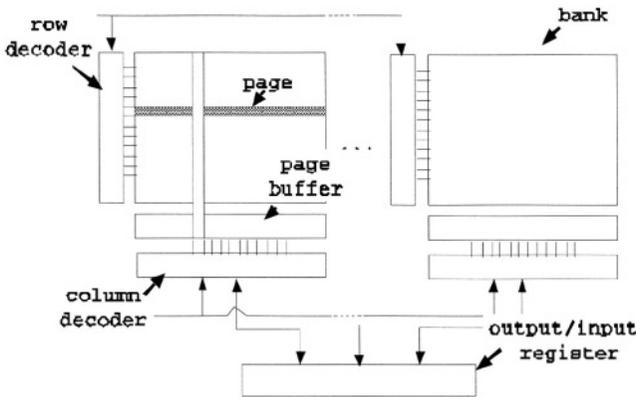


Figure 24-1. A multi-banked SDRAM.

$$E = \sum_{\forall i=1}^{N_{banks}} (E_{static}^i + E_{dynamic}^i)$$

$$E_{static}^i = P_{cs} * t_{cs}^i + P_{stby} * t_{pwn}^i + P_{pwn} * t_{pwn}^i$$

$$E_{dynamic}^i = N_{cs} * E_{pa} + N_{rw}^i * E_{wr}$$

where:

- $t_{cs/stby/pwn}^i$ : time, in CS/STBY/PWDNmode
- $P_{cs/stby/pwn}$ : power in CS/STBY/PWDNmode
- $E_{pa}$ : energy of a precharge/activation
- $E_{rw}$ : energy of a read/write
- $N_{pa}^i$ : number of precharge and activations in Bank  $i$
- $N_{rw}^i$ : number of reads and writes in Bank  $i$

The static energy consumption depends on which energy states are used during execution. An energy state manager [8] controls when the banks should transition to another energy state. As soon as the bank idles for more than one cycle, the manager switches the bank to the CS-mode. When the bank is needed again it is switched back to STDBY-mode within a cycle. Finally, we switch off the bank as soon as it remains idle for longer than a million cycles.<sup>1</sup> The dynamic energy consumption depends on which operations are needed to fetch/store the data from/into the memory. The energy parameters are presented in Table 24-1. The remaining parameters are obtained by simulation (see Section 6).

Table 24-1. Energy consumption parameters.

$E_{precharge/activate}$	14 nJ/miss
$E_{read/write}$	2 nJ/access
$P_{stby}$	50 mW
$P_{cs}$ with self refresh	10.8 mW
$P_{pwn}$	0 mW

### 3. MOTIVATION AND PROBLEM FORMULATION

According to our experiments on a multi-processor architecture, the dynamic energy contributes up to 68% of the energy of an SDRAM. Compared to a single-processor architectures, the SDRAM banks are more frequently used, and thus less static energy is burned waiting between consecutive accesses. Moreover, even though in future technology leakage energy is likely to increase, many techniques are under development by DRAM manufactures to reduce the leakage energy during inactive modes [5]. Therefore, in this paper we focus on data assignment techniques to reduce the dynamic energy. At the same time, our techniques reduce the time of the SDRAM banks spent in the active state, thereby thus reducing the leakage loss in that state which

is more difficult to control [5]. We motivate with an example how a good data assignment can reduce the number of page-misses, thereby saving dynamic energy. The example consists of two parallel executing tasks, *Convolve* and *Cmp* (see Table 24-2).

Page-misses occur when e.g. *kernel* and *imgin* of *Convolve* are assigned to the same bank. Each consecutive access evicts the open-page of the previous one, causing a page-miss ( $2K^2$ -misses in total). Similarly, when data of different tasks are mapped to the same bank, (e.g. *kernel* of *Convolve* and *img1* of *Cmp*), each access to the bank potentially causes a page-miss. The number of page-misses depends on how the accesses to the data structures are interleaved. When *imgin* and *imgout* are mapped in the same bank, an access to *imgout* is only scheduled after  $K^2$  accesses to *imgin*. Therefore, the frequency at which accesses to both data structures interfere is much lower than for *kernel* and *imgin*. The energy benefit of storing a data structure alone in a bank depends on how much spatial locality exists in the access pattern to the data structure. E.g. *kernel* is a small data structure (it can be mapped on a single page) which is characterized by a large spatial locality. When *kernel* is stored in a bank alone, only one page-miss occurs for all accesses to it. The assignment problem is complicated by the dynamic behavior of modern multi-media applications. Only at run-time it is known which tasks are executing in parallel and which data needs to be allocated to the memory. A fully static assignment of the data structures to the memory banks is thus impossible. Dynamic memory allocators are a potential solution. However, existing allocators do not take the specific behavior of SDRAM memories into account to reduce the number of page-misses. To solve above issues we introduce two dynamic memory allocators, which reduce the number of page-misses.

Table 24-2. Extracts from *Convolve* and *Cmp*.

Convolve	Cmp
for (i=0; i<K; i++)	for (i+0; i<row; i++)
for (j=0; j<K; j++)	for (j=0; j<col; j++)
... = <i>imgin</i> [i][j]*	tmp1 = <i>img1</i> [I][j]
<i>imgout</i> [r*col][j] = ...	

#### 4. RELATED WORK

In the embedded system community the authors of [12] and [13] have presented assignment algorithms to improve the performance of SDRAM memories. Both algorithms distribute data over different banks such that the number of page-misses is minimized. Their optimizations rely on the fact that single threaded applications are analyzable at design-time. This is not possible for dynamic multi-media applications.

In [11], techniques are presented to reduce the static energy consumption of SDRAMs in embedded systems. The strategy of this paper consists of clustering data structures with a large temporal affinity in the same memory bank. As a consequence the idle periods of banks are grouped, thereby creating more opportunities to transition more banks in a deeper low-power mode for a longer time. Several researchers (see e.g. [7], [8] and [11]) present a hardware memory controller to exploit the SDRAM energy modes. The authors of [15] present a data migration strategy. It detects at run-time which data structures are accessed together, then it moves arrays with a high temporal affinity to the same banks. The research on techniques to reduce the static energy consumption of SDRAMs is very relevant and promising. Complementary to the static energy reduction techniques, we seek to reduce the dynamic energy contribution.

The most scalable and fastest multi-processor virtual memory managers (e.g. [9] and [16]) use a combination of private heaps combined with a shared pool to avoid memory fragmentation. They rely on hardware based memory management units to map virtual memory pages to the banks. These hardware units are unaware of the underlying memory architecture. Therefore, in the best-case<sup>2</sup> memory managers distribute the data across all memory banks (random allocation). In the worst-case, all data is concentrated in a single memory bank. We will compare our approach to both extremes in Section 6.

## 5. BANK AWARE ALLOCATION ALGORITHMS

We first present a best effort memory allocator (**BE**). The allocator can map data of different tasks to the same bank in order to minimize the number of page-misses. Hence, accesses from different tasks can interleave at run-time, causing unpredictable page-misses. At design-time, we do not exactly know how much the page-misses will increase the execution time of the tasks. As a consequence, the best effort allocator cannot be used when hard real-time constraints need to be guaranteed and little slack is available. The goal of the guaranteed performance allocator (**GP**) is to minimize the number of page-misses while still guaranteeing the real-time constraints.

### 5.1. Best effort memory allocator

The BE (see below) consists of a design-time and a run-time phase. The design-time phase bounds the exploration space of the run-time manager reducing its time and energy penalty.

```

1: Design-Time
2: for all T ∈ Task do
3:   for all ds (= data structure) ∈ T do
4:     Compute local selfishness :
           
$$S_{ds}^{local} = N_{ds}^{access} \frac{\tau_{ds}^{misses}}{\tau_{ds}^{accesses}}$$

5:     Add  $S_{ds}^{local}$  to  $Tab_{info}$ 
6:   end for
7: end for
8: Run-Time:
9: Initialize selfishness of all available banks
           
$$S_{bank} = 0$$

10: Initialize ordered queue Q
11: for all T ∈ ActiveTasks do
12:   for all ds T ∈ do
13:     Insert ds in Q according decreasing  $S_{ds}^{local}$ 
14:   end for
15: end for
16: while Q is not empty do
17:   ds = head of Q
18:   Insert ds in bank with smallest selfishness
19:   Update selfishness of the bank:
           
$$S_{bank} = S_{ds}^{local}$$

20: end while

```

At design-time we characterize the data structures of each task with a heuristic parameter: *selfishness* (line 4:  $S_{ds}^{local}$ ). When accesses to a selfish data structure are not interleaved with accesses to other data structures in the same bank, page-misses are avoided. Selfishness of a data structure is calculated by dividing the average time between page-misses ( $\tau_{ds}^{misses}$ ) with the average time between accesses ( $\tau_{ds}^{accesses}$ ). This ratio expresses the available spatial locality and can be either measured or calculated at design-time. We weigh it with the importance of the data structure by multiplying it with the number of accesses to the data structure ( $N_{ds}^{accesses}$ ). Finally, we add extra data structures to the source code for the design-time info needed at run-time (line 5:  $Tab_{info}$ ). At run-time, when it is known which tasks are activated at the start of a new frame and thus which data needs to be allocated, the algorithm assigns the live data to the memory banks.<sup>3</sup> The algorithm distributes the data among the banks such that the selfishness of all the banks is balanced. The selfishness of a bank ( $S_{bank}$ ) is the sum of the selfishness of all data structures in the bank. The algorithm ranks the data structures according to decreasing selfishness (line: 11–15) and then greedily assigns the data to the banks starting from the most selfish one (lines: 15–20). Each data struc-

ture is put in the least selfish bank. This strategy puts the most selfish data structures in separate banks and clusters the remaining ones.

## 5.2. Guaranteed performance allocation

Time guarantees are only possible when all page-misses can be predicted, but that can be difficult due to the interference between accesses of different tasks. An obvious way to avoid page-interference is to assign the data of simultaneously active tasks to independent banks. The following two degrees of freedom then remain: how to partition the banks among the tasks and how to assign the data of each task to its partition. The sensitivity of the number of page-misses to the number of banks varies from task to task. Some tasks benefit more than others from having extra banks assigned to it. Our guaranteed performance approach allocates more banks to those tasks, which benefit most.

At design-time we generate a data assignment for every task and for any possible number of banks. The resulting assignments for each task can be presented in a Pareto curve, which trades off the energy consumption of the task in function of the number of banks. The Pareto curves can be created with the best-effort approach based on selfishness.

At run-time we distribute the available banks of the platform among the active tasks using the Pareto curves. We select a point on the Pareto curve of each task such that the energy consumption of all tasks is minimized and that the total number of banks for all tasks is less or equals the available number of banks on the platform. We reuse for this purpose a greedy heuristic which we have developed in the context of task-scheduling [6].

## 6. EXPERIMENTAL RESULTS

The processing elements and their performance are simulated using an adapted ARMulator [14]. This simulator dumps a memory access trace for each task in the parallel application. We input the memory traces in an energy evaluation script. It combines the memory traces in a cycle-accurate way according to the issue-time of each access, the task schedule and the memory hierarchy. The output is the total execution time of each task<sup>4</sup> and the SDRAM energy consumption. To evaluate the effectiveness of our assignment techniques, we have generated several representative task-sets, derived from MediaBench [2] (*Cmp*, *Convolve*,<sup>5</sup> *Dct* and *Rawcaudio*). We have also added *Rgb2Yuv*, *Quick24to8*, *Fir*, *Lzw* and *Rinjadel*. In Table 24-4. we show how their energy consumption in function of the number of memory banks.

We verify the quality of our heuristics (best-effort (**BE**) and guaranteed performance (**GP**) against a Monte-Carlo approximation of the best-possible assignment (**MA**)). The results of the latter were obtained by measuring 100 different data assignments. We compare our memory allocators against two

Table 24-4. Energy Consumption of the Benchmark Tasks @ 100 Mhz..

Tasks	Nds	NrBanks					
		1	2	3	4	5	6
Cmp	3	4331	3293	993	–	–	–
Fir	3	1770	457	489	–	–	–
Lzw	3	6482	7004	7655	–	–	–
Rawcaudio	4	4202	2998	3234	4061	–	–
Convolve	4	19222	8173	8474	8474	–	–
Rinjadel	4	7073	6491	6870	7277	–	–
Rgb2Yuv	5	528	796	1064	1333	1593	–
Dct	6	2552	2845	2118	2540	3015	3485
Quick24to8	10	6930	5597	4215	4417	4620	4824

existing policies. The first reference policy, random allocation (**RA**) randomly distributes the data structures across the memory banks similar to architecture-unaware allocators (see Section 4). In the second reference we do not share the SDRAMs among the processors. Each processor owns an equal number of memory banks. On each processor a local memory allocator manages the private banks (sequential allocation (**SA**)). In Figure 24-2 we visualize the energy consumption of the different allocators for the *Convolve* and *Cmp* task-set. Similar results for other tasks-sets are presented in Table 24-5.

The energy consumption of all allocators, except for SA, first decreases when the number of banks is increased. The allocators distribute the data across more banks, thereby reducing the number of page-misses and the

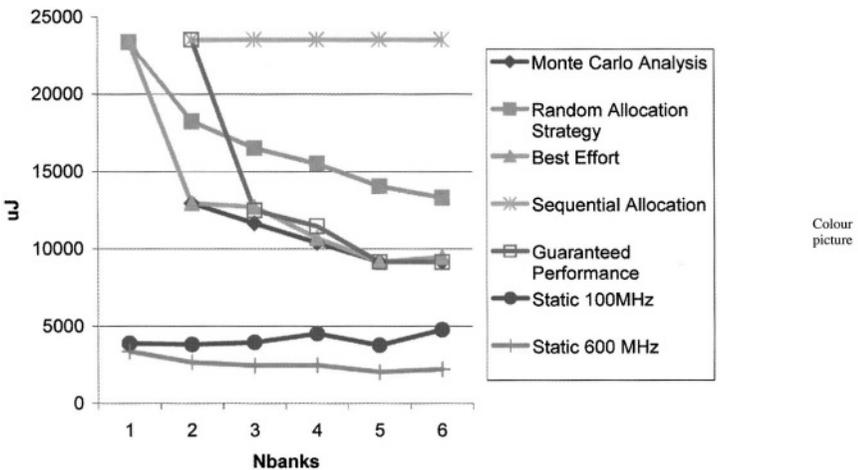


Figure 24-2. Comparison of allocation strategies for Convolve and Cmp

dynamic energy. At the same time, the static energy consumption slightly reduces since less misses results in a shorter execution time (see the static energy of the example in Figure 24-2 executed at 600 MHz.).

However, when extra banks do not significantly reduce the page-miss rate anymore, the dynamic energy savings become smaller than the extra static energy needed to keep the banks in CS/STBY-mode. The total energy consumption increases then again. A nice illustration of this is *Quick24to8* in Table 24-5. The total energy consumption decreases up to three banks and then increases again due to the extra static energy. Also, in Figure 24-2. the total energy consumption increases again when more than five banks are used. From these examples, we see that an optimal number of active banks exist. The optimal number of banks depends on the ratio of static versus dynamic energy. When the banks become more active (e.g. because more tasks are activated or the processor frequency is increased), the dynamic energy becomes more important than the static energy and the optimal number of

Please check no. of figures and no. of tables

Table 24-5. Energy comparison of several allocation strategies.

Tasks (100 Mhz)	Nds	$N_{bank}$ ( $\mu$ J)					
		1	2	3	4	5	6
Fir+Conv(SA)	7	–	20993	20993	20993	20993	20993
Fir+Conv(MA)	7	20958	9858	9269	8641	8641	8641
Fir+Conv(RA)	7	20958	15712	13834	12381	12436	10990
Fir+Conv(GP)	7	–	20993	9943	8641	8641	8641
Fir+Conv(BE)	7	20958	10250	9269	9515	8942	8964
R2Y+Cmp(SA)	8	–	4859	4859	4859	4859	4859
R2Y+Cmp(MA)	8	4986	3832	1877	1521	1521	1521
R2Y+Cmp(RA)	8	4986	4362	3733	3407	3368	3209
R2Y+Cmp(GP)	8	–	4859	3821	1521	1521	1521
R2Y+Cmp(BE)	8	4986	4041	2031	1553	1821	2089
R2Y+Cmp+Conv(SA)	12	–	–	24082	24082	24082	24082
R2Y+Cmp+Conv(MA)	12	23531	13872	12456	11392	10109	10060
R2Y+Cmp+Conv(RA)	12	23531	13872	12456	11392	10109	10060
R2Y+Cmp+Conv(GP)	12	–	–	24082	13034	11987	9695
R2Y+Cmp+Conv(BE)	12	23531	13769	13468	11515	9907	10206
R2Y+Cmp+Conv+Dct(SA)	18	–	–	–	26647	26647	26647
R2Y+Cmp+Conv+Dct(MA)	18	26195	16684	15165	13665	13132	12514
R2Y+Cmp+Conv+Dct(RA)	18	26195	21896	19332	17947	17517	17696
R2Y+Cmp+Conv+Dct(GP)	18	–	–	–	26647	15598	14551
R2Y+Cmp+Conv+Dct(BE)	18	26195	16212	16143	14224	13405	12758
Fir+Conv(SA)	7	–	20305	20305	20305	20305	20305
Fir+Conv(MA)	7	20480	9261	7582	6600	6494	6473
Fir+Conv(RA)	7	20480	14938	11664	11421	10115	9196
Fir+Conv(GP)	7	–	20305	8395	7842	6494	6473
Fir+Conv(BE)	7	20480	9526	8219	7623	6494	6473

banks increases. E.g. in Figure 24-2. the optimal number of banks increases from five to six when the processor frequency changes from 100 MHz to 600 MHz.

SA performs poorly when the energy consumption is dominated by the dynamic energy. It cannot exploit idle banks owned by other processors to reduce the number of page-misses. The difference between SA and MA (an approximation of the best-possible assignment) is large (more than 300), indicating that sharing SDRAM memories is an interesting option for heterogeneous multi-processor platforms. It increases the exploration space such that better assignments can be found. When the banks are not too heavily used, even no performance penalty is present (see below).

We also observe in Figure 24-2. that existing commercial multi-processor memory allocators (RA) perform badly compared to MA. This suggests that large headroom for improvement exists. When only one bank is available, obviously all memory allocation algorithms produce the same results. With an increasing number of banks the gap between RA and MA first widens as a result of the larger assignment freedom (up to 55% for *Rgb2Yuv* and *Cmp* with four banks). However, the performance of the RA improves with an increasing number of banks: the chances increase that RA distributes the data structures across the banks, which significantly reduces the energy consumption. Therefore, when the number of banks becomes large the gap between RA and MA becomes smaller again (50% for *Rgb2Yuv* and *Cmp* with six banks).

For higher processor frequencies the static energy consumption decreases and the potential gains become larger. E.g. for *Convolve* and *Cmp* the gap increases from 26% to 34%.

Figure 24-2 shows how BE outperforms RA. Results in Table 24-6. suggest an improvement up to 50% (see *Rgb2Yuv* and *Cmp* with four banks). Moreover, BE often comes close to the MA results. The difference between BE and MA is always less than 23%. When the number of tasks in the application becomes large (see task-set in Table 24-6. which consists of 10 tasks), we note a small energy loss of BE compared to RA for the first task-set and eight banks are used. In this case 50 data structures are allocated in

Table 24-6. Energy comparison of several allocation strategies.

Tasks(100Mhz)	Nds	Nbank(uJ)		
		8	16	32
2Quick+2R2Y+2Rin+Cmp+2Lzw(SA)	50	–	48788	48788
2Quick+2R2Y+2Rin+Cmp+2Lzw(RA)	50	49525	41872	45158
2Quick+2R2Y+2Rin+Cmp+2Lzw(MA)	50	47437	37963	41610
2Quick+2R2Y+2Rin+Cmp+2Lzw(GP)	50	–	38856	34236
2Quick+2R2Y+2Rin+Cmp+2Lzw(BE)	50	51215	40783	35371
2Quick+2R2Y+2Rin+Cmp+2Lzw(Estatic)	50	10227	9045	8489

a small amount of banks. As a result, a limited freedom exists to reduce the number of page-misses, i.e. the energy gap between maximum and minimum energy consumption is small. Note that currently BE cannot detect the optimal number of banks. When the banks are not actively used, its energy consumption increases (compare e.g. *Cmp* and *Convolve* for five and six banks), but it remains lower than existing dynamic allocation policies.

GP performs equally well for a sufficiently large number of banks. The main advantage of this technique is that the execution times of the different tasks can be predicted and guaranteed. Moreover, it will never use more than the optimal number of banks, but its performance breaks down when only few banks are available per task. In this case, it maps similar to SA all data structures of each task in a single (or few) banks. It then consumes more energy than RA (29% for *Convolve* and *Cmp* with two banks).

This data-assignment technique combined with task scheduling can significantly improve the performance of a task-set. Or, for a given deadline, the joined approach can be used to reduce the energy consumption of the application [4].

The run-time overhead of both BE and GP is limited: usually, large time-intervals exist between successive calls to the run-time manager, which allows to relax its time- and energy-overhead over a large period. Also, in the context of run-time task scheduling [6], we have shown that the time and energy overhead of a comparable run-time scheduler is low.

## 7. CONCLUSIONS AND FUTURE WORK

Low-power design is a key issue for future dynamic multi-media applications mapped on multi-processor platforms. On these architectures off-chip SDRAM memories are big energy consumers. This paper presents two dynamic memory allocators for bank assignment: a best-effort and a guaranteed performance allocator. Experimental results obtained with a multi-processor simulator are very promising. The allocators significantly reduce the energy consumption of SDRAMs compared to existing dynamic memory managers.

## NOTES

<sup>1</sup> Similar to [12].

<sup>2</sup> For dynamic energy consumption

<sup>3</sup> We currently assume that tasks can only be started/deleted at predefined points in the program. However, this is not a severe limitation for most modern multi-media applications (see [6]).

<sup>4</sup> Including both the processor and memory delay.

<sup>5</sup> *Convolve* and *Cmp* are parts of *Edge\_detection*.

**REFERENCES**

1. M. Viredaz and D. Wallach. "Power Evaluation of a Handheld Computer: A Case Study." *WRL Compaq*, May, 2001
2. C. Lee et al. "MediaBench: A Tool for Evaluation and Synthesizing Multimedia and Communication Systems." *Micro*, 1997
3. "Calculating Memory System Power For DDR." *Micron*, Technical Report TN-46-03
4. J. I. Gomez et al. "Scenario-based SDRAM-Energy-Aware Scheduling for Dynamic Multimedia Applications on Multi-Processor Platforms." *WASP*, Istanbul, Turkey, 2002.
5. K. Itoh. "Low-Voltage Memories for Power-Aware Systems." *Proc. Islped.*, Monterey CA, August 2002, pp. 1–6
6. P. Yang et al. "Managing Dynamic Concurrent Tasks in Embedded Real-Time Multimedia Systems." *Proc. Isss*, October, 2002
7. A. Lebeck et al. "Power Aware Page Allocation." *Proceedings of 9th Asplos*, Boston MA, November 2000
8. X. Fan et al. "Controller Policies for DRAM Power Management." *Proc. Islped*, New Orleans, LO, 2001, pp. 129–134.
9. M. Shalan and V. Mooney III. "A Dynamic Memory Management Unit for Embedded Real-Time Systems-on-a-Chip." *Cases*, San Jose, CA, November 2000, pp. 180–186.
10. Y. Joo, Y. Choi and H. Shim. "Energy Exploration and Reduction of SDRAM Memory Systems." *Proceedings of 39th Dac*, New Orleans, LO, 2002, pp. 892–897.
11. V. Delaluz et al. "Hardware and Software Techniques for Controlling DRAM Power Modes." *IEEE Trans. Computers*, Vol. 50, No. 11, pp. 1154–1173, November 2001.
12. P. Panda. "Memory Bank Customization and Assignment in Behavioral Synthesis." *Proc. Iccad*, October 1999, pp. 477–481.
13. H. Chang and Y. Lin, "Array Allocation Taking into Account SDRAM Characteristics." *Proc. ASP-Dac*, 2000, pp. 447–502.
14. ARM, [www.arm.com](http://www.arm.com)
15. V. Delaluz, M. Kandemir and I. Kolcu. "Automatic Data Migration for Reducing Energy Consumption in Multi-Bank Memory Systems." *Proceedings of 39th DAC*, 2002, pp. 213–218.
16. E. Berger et al. "Hoard: A Scalable Memory Allocator for Multithreaded Applications." In *Proceedings of 8th Asplos*, October 1998.

PART VII:

SAFE AUTOMOTIVE SOFTWARE DEVELOPMENT

*This page intentionally left blank*

# SAFE AUTOMOTIVE SOFTWARE DEVELOPMENT

Ken Tindell<sup>1</sup>, Hermann Kopetz<sup>2</sup>, Fabian Wolf<sup>3</sup> and Rolf Ernst<sup>4</sup>

<sup>1</sup> *LiveDevices, York, UK (1), Technische Universität Wien, Austria (2), Volkswagen AG, Wolfsburg, Germany (3), Technische Universität Braunschweig, Germany (4)*

**Abstract.** Automotive systems engineering has made significant progress in using formal methods to design safe hardware-software systems. The architectures and design methods could become a model for safe and cost-efficient embedded software development as a whole. This paper gives several examples from the leading edge of industrial automotive applications.

**Key words:** embedded software, safety critical, automotive, real-time systems

## 1. INTRODUCTION

*R. Ernst, Technische Universität Braunschweig*

Automotive software is for a large part safety critical requiring safe software development. Complex distributed software functions and software function integration challenge traditional simulation based verification approaches. On the other hand, software functions have to match the high fault tolerance and fail safe requirements of automotive designs. To cope with this challenge, advanced research in automotive software has developed formal approaches to safe automotive design which could become a model for safe and cost-efficient embedded software development as a whole. The special session which is summarized in this paper includes contributions from most renowned experts in the field. The following section will outline the problems when integrating software IP from several sources on one electronic control unit and gives an example of a hardware-software solution to observe and enforce real-time behavior. Section 3 looks at the next level, distributed systems. Only at this level, the high safety requirements of autonomous vehicle functions such as X-by-Wire can be met. The section introduces the concepts of fault-containment regions and error containment and introduces architectures and corresponding hardware-software control techniques to isolate defective parts. The presented time-triggered architecture (TTA) and the time-triggered communication protocol are used throughout the automotive and aircraft industries. The last section gives an example of a practical application of formal methods to software integration for combustion engine control.

## 2. THE NEED FOR A PROTECTED OS IN HIGH-INTEGRITY AUTOMOTIVE SYSTEMS

*K. Tindell, LiveDevices, York*

### 2.1. Motivation

As ECU (Electronic Control Unit) software becomes more complex there is a need to run more than one major subsystem inside a single ECU (such as a mixture of high- and low-integrity software in an X-by-Wire system). This need gives rise to the demand for an ECU operating system that can provide protection between software components. The OS can isolate a fault in a software component and prevent that fault from propagating to another subsystem in the same ECU. The need for such a protected OS has been recognized by the automotive industry. The vehicle manufacture initiative of the HIS group (Herstellerinitiative Software) of Audi, BMW, DaimlerChrysler, Porsche and VW, lays out requirements for a protected OS, the two most interesting of which are:

Protection must apply between applications (i.e. groups of tasks) as well as tasks. This is because a single application, developed by a single organization, will typically be composed of a set of tasks. Tasks need protection from each other, but they must also share data (typically via shared memory for performance reasons).

The OS must be designed to tolerate malicious software. This means that a programmer with the deliberate intent of doing damage cannot cause another application in the ECU to fail. Although there are not major concerns about deliberate sabotage by a programmer, this requirement is here because software failures can be as intricate and subtle as if they were deliberately programmed. Thus the requirement to protect against all possible failures results in a more secure system.

Not only does protection need to be applied in the functional domain (e.g. controlled access to memory, I/O) but it also needs to be applied in the timing domain (e.g. tasks must not run for too long or too frequently). This timing requirement is necessary because the new approaches to ECU development are based on schedulability analysis - engineering mathematics that determine the worst-case response times of activities (tasks, interrupt handlers) in a real-time system. Such analysis is necessary for a complex ECU with an OS because the timing determinism of a rigid statically scheduled system is no longer present.

Protection in the timing domain entails that the data used by the schedulability analysis (e.g. task execution times) is enforced at run-time so that the timing analysis always remains valid. For example, if a task runs for too long then the OS needs to step in and stop it in order that lower priority (and potentially more critical) tasks that are in another application are not held out for longer than calculated in the schedulability analysis.

A final and very important set of requirements are imposed by the

automotive industry on any use of electronics: component costs must be as low as possible. This means that the choice of hardware is very limited. Today only Texas Instruments with the TMS470 and Infineon with the Tricore TC17xx provide silicon to automotive requirements that also contains the necessary memory protection hardware.

## **2.2. The APOS research project**

The Advanced Technology Group (ATG) of LiveDevices has been conducting a research project to look at these issues. It has developed a prototype OS called 'APOS' (ATG Protected OS) to meet the HIS requirements and to investigate the issues surrounding a protected OS (e.g. examining the additional overheads due to protection, the issues of OSEK OS compliance).

The following sections discuss the results of this project in more detail.

## **2.3. Architecture**

The APOS kernel runs on the Tricore TC1775 from Infineon. It supports supervisor- and user-mode execution, which prevents normal tasks from executing privileged instructions and restricts these tasks to accessing memory to four pre-defined regions (private RAM, private ROM, application RAM, application ROM). The private RAM is used to store the task stack.

Access to the kernel is via API calls that are implemented with TRAP instructions (as for a conventional OS).

API calls check for permissions before completing. There are permissions assigned off-line to each task in the system that indicates which OSEK OS objects (tasks, resources, alarms, counters, etc.) that the task may access. Some critical API calls (such as the call to shutdown the OS and re-start the ECU) have special permissions checking so that only certain trusted tasks can make these calls.

The complete ECU is constructed from a set of applications (in turn composed of a set of tasks) by the ECU integrator. This is a critical and trusted role since one of the jobs of the system integrator is to assign permissions and levels of trust to different tasks. Another job of the system integrator is to perform the timing analysis of the complete system to ensure that all the tasks in all the applications meet their defined timing requirements (i.e. deadlines). Only the system integrator can do this since the schedulability analysis requires complete information about the timing behaviors of the application tasks.

Communication between tasks within the same application is typically via shared memory. But communication between applications is via OSEK COM intra-ECU messaging.

## 2.4. Execution time monitoring

Every task in APOS is assigned an execution time budget. A task begins running with a virtual stopwatch set to this time. The stopwatch then counts down while the task is running. If the task is pre-empted by another task (or interrupt handler) then the stopwatch is stopped, to be resumed when the task continues running. If the stopwatch reaches zero then the APOS kernel terminates the task. Thus a task that runs for too long is killed before it can disrupt the execution of other lower priority tasks (most especially those that might be in another application).

Execution times are measured in hardware clock ticks and stored as 32-bit integers. The TC1775 unit used in development is clocked at 20 MHz; thus no task execution time budget in the system can be longer than 3.6 minutes. Since a task rarely runs for longer than a few milliseconds, this is adequate. One exception is the idle task: this task never terminates, and so will run for longer than 3.6 minutes. The APOS implementation addresses this problem by providing a special “refresh budget” API call. Only the idle task is permitted to make this call.

APOS also keeps track of the longest execution time observed for each task. This aids the developer in setting appropriate execution time limits in the case where the developer does not have access to tools that can calculate statically a worst-case execution time figure. The measurement feature can also guide the testing strategy to ensure that the software component tests do exercise the worst-case paths in the task code.

In addition to patrolling the total execution time of a task, the kernel also patrols execution times of the task while holding a resource. In OSEK OS a resource is actually a semaphore locked and unlocked according to the rules of the priority ceiling protocol. This is done because the maximum blocking time of a higher priority task can be calculated. However, the time for which a task holds a given resource needs to be bounded (this time is used in the schedulability analysis) and hence enforced at run-time. The APOS kernel does this: when a task obtains an OSEK OS resource another virtual stopwatch is started and counts down. If the task fails to release the resource within the defined time then it is killed and the resource forcibly released. Unfortunately, this may lead to application data corruption (a typical use for a resource is to guard access to application shared data). However, since a failed task can in any case corrupt application shared data, it is important that the application be written to tolerate this failure (perhaps by having a soft restart option).

## 2.5. Execution pattern monitoring

The schedulability analysis performed by the system integrator not only uses the worst-case execution time figures for each task (and resource access patterns), but also the pattern in which the task (or interrupt handler) is

invoked. The OS must enforce the execution patterns since any task or interrupt handler exceeding the bounds implied by the pattern may cause another task in another application to fail (and hence violating the strong protection requirements). There is therefore a strong coupling between enforcement and analysis: there is no point analyzing something that cannot be enforced, and vice versa.

Simple schedulability analysis requires a minimum time between any two invocations of a task or interrupt handler (i.e. the period in the case of periodic invocations). But in real systems there are often tasks that run with more complex patterns, particularly when responding to I/O devices. For example, an interrupt handler servicing a CAN network controller may be invoked in a ‘bursty’ fashion, with the short-term periodicity dictated by the CAN baud rate and the long-term periodicity a complex composite of CAN frame periodicities. Making the assumption that the interrupt handler simply runs at the maximum rate is highly pessimistic. Although the analysis can be extended to account for the more complex behavior, enforcing such arbitrary patterns efficiently in the OS is impossible. An alternative approach is taken by using two deferred servers for each sporadic task or interrupt handler. This approach provides two invocation budgets for each sporadic task and interrupt handler. If either budget is exhausted then no further invocations are permitted (for an interrupt handler the interrupts are disabled at source). The budgets are replenished periodically (typically with short- and long-term rates).

## **2.6. Performance and OSEK**

Early figures for performance of the OS compared to a conventional OSEK OS indicate that the CPU overheads due to the OS are about 30–50% higher and the RAM overheads are about 100% higher. Given the very low overheads of an OSEK OS and the benefits of sharing the hardware across several applications, this is quite acceptable. Furthermore, no significant OSEK OS compatibility issues have been discovered.

## **2.7. Summary**

In the near future automotive systems will require the ability to put several applications on one ECU. There are many technical demands for this but all are soluble within the general requirements of the automotive industry for low component cost.

### 3. ARCHITECTURE OF SAFETY-CRITICAL DISTRIBUTED REAL-TIME SYSTEMS

*H. Kopetz, Technische Universität Wien*

Computer technology is increasingly applied to assist or replace humans in the control of safety-critical processes, i.e., processes where some failures can lead to significant financial or human loss. Examples of such processes are *by-wire-systems* in the aerospace or automotive field or *process-control systems* in industry. In such a computer application, the computer system must support the safety, i.e., the probability of loss caused by a failure of the computer system must be very much lower than the benefits gained by computer control.

Safety is a system issue and as such must consider the system as a whole. It is an emergent property of systems, not a component property [1], p. 151. The safety case is an accumulation of evidence about the quality of components and their interaction patterns in order to convince an expert (a certification authority) that the probability of an accident is below an acceptable level. The safety case determines the criticality of the different components for achieving the system function. For example, if in a drive-by-wire application the computer system provides only assistance to the driver by advising the driver to take specific control actions, the criticality of the computer system is much lower than in a case where the computer system performs the control actions (e.g., braking) autonomously without a possible intervention by the driver. In the latter case, the safety of the car as a whole depends on the proper operation of the computer system. This contribution is concerned with the architecture of safety-critical distributed real-time systems, where the proper operation of the computer system is critical for the safety of the system as a whole.

A computer architecture establishes a framework and a blueprint for the design of a class of computing systems that share a common set of characteristics. It sets up the computing infrastructure for the implementation of applications and provides mechanisms and guidelines to partition a large application into nearly autonomous subsystems along small and well-defined interfaces in order to control the complexity of the evolving artifact [2]. In the literature, a failure rate of better than  $10^{-9}$  critical failures per hour is demanded in ultra-dependable computer applications [3]. Today (and in the foreseeable future) such a high level of dependability cannot be achieved at the component level. If it is assumed that a component – a single-chip computer – can fail in an arbitrary failure mode with a probability of  $10^{-6}$  failures per hour then it follows that the required safety at the system level can only be achieved by redundancy at the architecture level.

In order to be able to estimate the reliability at the system level, the experimentally observed reliability of the components must provide the input to a reliability model that captures the interactions among the components and calculates the system reliability. In order to make the reliability calculation

tractable, the architecture must ensure the independent failure of the components. This most important independence assumption requires fault containment and error containment at the architecture level. Fault containment is concerned with limiting the immediate impact of a fault to well-defined region of the system, the fault containment region (FCR). In a distributed computer system a node as a whole can be considered to form an FCR. Error containment is concerned with assuring that the consequences of the faults, the errors, cannot propagate to other components and mutilate their internal state. In a safety-critical computer system an error containment region requires at least two fault containment regions.

Any design of a safety-critical computer system architecture must start with a precise specification of the fault hypothesis. The fault hypothesis partitions the system into fault-containment regions, states their assumed failure modes and the associated probabilities, and establishes the error-propagation boundaries. The fault hypothesis provides the input for the reliability model in order to calculate the reliability at the system level. Later, after the system has been built, it must be validated that the assumptions which are contained in the fault hypothesis are realistic.

In the second part of the presentation it will be demonstrated how these general principles of architecture design are realized in a specific example, the Time-Triggered Architecture (TTA) [4]. The TTA provides a computing infrastructure for the design and implementation of dependable distributed embedded systems. A large real-time application is decomposed into nearly autonomous clusters and nodes and a fault-tolerant global time base of known precision is generated at every node. In the TTA this global time is used to precisely specify the interfaces among the nodes, to simplify the communication and agreement protocols, to perform prompt error detection, and to guarantee the timeliness of real-time applications. The TTA supports a two-phased design methodology, architecture design and component design. During the architecture design phase the interactions among the distributed components and the interfaces of the components are fully specified in the value domain and in the temporal domain. In the succeeding component implementation phase the components are built, taking these interface specifications as constraints. This two-phased design methodology is a prerequisite for the composability of applications implemented in the TTA and for the reuse of pre-validated components within the TTA. In this second part we present the architecture model of the TTA, explain the design rationale, discuss the time-triggered communication protocols TTP/C and TTP/A, and illustrate how component independence is achieved such that transparent fault-tolerance can be implemented in the TTA.

## **4. CERTIFIABLE SOFTWARE INTEGRATION FOR POWER TRAIN CONTROL**

*F. Wolf, Volkswagen AG, Wolfsburg*

### **4.1. Motivation**

Sophisticated electronic control is the key to increased efficiency of today's automotive system functions, to the development of novel services integrating different automotive subsystems through networked control units, and to a high level of configurability. The main goals are to optimize system performance and reliability, and to lower cost. A modern automotive control unit is thus a specialized programmable platform and system functionality is implemented mostly in software.

The software of an automotive control unit is typically separated into three layers. The lowest layer are system functions, in particular the real-time operating system, and basic I/O. Here, OSEK is an established automotive operating system standard. The next higher level is the so-called 'basic software'. It consists of functions that are already specific to the role of the control unit, such as fuel injection in case of an engine control unit. The highest level are vehicle functions, e.g. adaptive cruise control, implemented on several control units. Vehicle functions are an opportunity for automotive product differentiation, while control units, operating systems and basic functions differentiate the suppliers. Automotive manufacturers thus invest in vehicle functions to create added value.

The automotive software design process is separated into the design of vehicle software functions (e.g. control algorithms), and integration of those functions on the automotive platform. Functional software correctness can be largely mastered through a well-defined development process, including sophisticated test strategies. However, operating system configuration and non-functional system properties, in particular timing and memory consumption are the dominant issues during software integration.

### **4.2. Automotive software development**

While automotive engineers are experts on vehicle function design, test and calibration (using graphical tools such as ASCET-SD or Matlab/Simulink, hardware-in-the-loop simulation etc.), they have traditionally not been concerned with software implementation and integration. Software implementation and integration is usually left to the control unit supplier who is given the full specification of a vehicle function to implement the function from scratch. Some automotive manufacturers are more protective and implement part of the functions in-house but this does not solve the software integration problem.

This approach has obvious disadvantages. The automotive manufacturer has to expose his vehicle function knowledge to the control unit supplier who

also supplies the manufacturer's competitors. It is hard to protect intellectual property in such an environment. Re-implementation of vehicle functions results in design cycles of several weeks. This inhibits design-space exploration and optimized software integration. Often the function returned by the system integrator does not fully match the required behavior resulting in additional iterations. From the automotive manufacturer's perspective, a software integration flow is preferable where the vehicle function does not have to be exposed to the supplier and where integration for rapid design-space exploration is possible. This can only be supported in a scenario where software functions are exchanged and integrated using object codes.

The crucial requirement here is that the integrated software must meet the stringent safety requirements for an automotive system in a certifiable way. These requirements generally state that it must be guaranteed that a system function (apart from functional correctness) satisfies real-time constraints and does not consume more memory than its budget. This is very different from the telecom domain where quality of service measures have been established. However, timeliness of system functions is difficult to prove with current techniques, and the problem is aggravated if software parts are provided by different suppliers. An important aspect is that even little additional memory or a faster system hardware that can guarantee the non-functional system correctness may simply be too costly for a high-volume product like an automotive control unit.

### **4.3. Certification of multi-source systems**

The focus is on a methodology and the resulting flow of information that should be established between car manufacturer, system supplier, OSEK supplier and system integrator to avoid the mentioned disadvantages and enable certifiable software integration. The information flow that is needed by the certification authority should be defined via formal agreements. Part of the agreements provides a standardized description of the OSEK configuration, process communication variables and scheduling parameters, such that intellectual property is protected and any system integrator can build an executable engine control. However, the novel key agreements should guarantee real-time performance and memory budgets of the integrated software functions. This requires suitable models for timing and resource usage of all functions involved: vehicle functions, basic software functions and system functions.

Commercial tool suites can determine process-level as well as system-level timing by simulation with selected test patterns. This approach lacks the possibility to explore corner-case situations that are not covered in the tests. It is known from real-time systems design that reliable system timing can only be achieved if properties of each function are described using conservative min-max intervals. Therefore, the enabling requirements for certifiable software integration are to obtain such conservative process-level intervals for

all functions involved, including operating system primitives, and to apply suitable system-level analysis techniques to determine all relevant system timing.

Different methodologies for the determination of conservative process-level timing intervals exist. The main problem is the absence of mature, industry-strength tool suites that support certifiable timing analysis for complex automotive systems. The same applies to system-level timing analysis tool suites. So for today's automotive systems, a combination of simulation-based timing analysis with careful test pattern selections and formal approaches where applicable is feasible.

#### 4.4. Conclusion

The need for distributed development of automotive software requires a certifiable integration process between car manufacturer and suppliers. Key aspects, i.e., software timing and memory consumption of the operating system, of the software functions provided by the control unit supplier as well as of the software functions provided by the car manufacturer have been identified. These need to be guaranteed for certifiable software integration.

#### REFERENCES

1. N. G. Leveson. *Safeware, System Safety and Computers*. Reading, MA: Addison Wesley Company, 1995.
2. H. A. Simon. *Science of the Artificial*. MIT Press, Cambridge, 1981.
3. N. Suri, C. J. Walter, and M. M. Hugue (eds). *Advances in Ultra-Dependable Systems*. IEEE Press, 1995.
4. H. Kopetz and G. Bauer. "The Time-Triggered Architecture." *Proceedings of the IEEE*, 2003. 91 (January 2003).

PART VIII:

EMBEDDED SYSTEM ARCHITECTURE

*This page intentionally left blank*

## Chapter 26

# EXPLORING HIGH BANDWIDTH PIPELINED CACHE ARCHITECTURE FOR SCALED TECHNOLOGY

Amit Agarwal, Kaushik Roy and T. N. Vijaykumar  
*Purdue University, West Lafayette, IN 47906, USA*

**Abstract.** In this article we propose a design technique to pipeline cache memories for high bandwidth applications. With the scaling of technology cache access latencies are multiple clock cycles. The proposed pipelined cache architecture can be accessed every clock cycle and thereby, enhances bandwidth and overall processor performance. The proposed architecture utilizes the idea of banking to reduce bit-line and word-line delay, making word-line to sense amplifier delay to fit into a single clock cycle. Experimental results show that optimal banking allows the cache to be split into multiple stages whose delays are equal to clock cycle time. The proposed design is fully scalable and can be applied to future technology generations. Power, delay and area estimates show that on average, the proposed pipelined cache improves MOPS (millions of operations per unit time per unit area per unit energy) by 40–50% compared to current cache architectures.

**Key words:** pipelined cache, bandwidth, simultaneous multithreading

### 1. INTRODUCTION

The phenomenal increase in microprocessor performance places significant demands on the memory system. Computer architects are now exploring thread-level parallelism to exploit the continuing improvements in CMOS technology for higher performance. Simultaneous Multithreading (SMT) [7] has been proposed to improve system throughput by overlapping multiple (either multi-programmed or explicitly parallel) threads in a wide-issue processor. SMT enables several threads to be executed simultaneously on a single processor, placing a substantial bandwidth demand on the cache hierarchy, especially on L1 caches. SMT's performance is limited by the rate at which the L1 caches can supply data.

One way to build a high-bandwidth cache is to reduce the cache access time, which is determined by cache size and set-associativity [3]. To reduce the access time, the cache needs to be smaller in size and less set associative. However, decreasing the size or lowering the associativity has negative impact on the cache miss rate. The cache miss rate determines the amount of time the CPU is stalled waiting for data to be fetched from the main memory.

Hence, there is a need for L1 cache design that is large and provides high bandwidth.

The main issue with large cache is that the bit-line delay does not scale well with technology as compared to the clock cycle time [5]. Clock speed is getting doubled every technology generation making cache access latency to be more than one cycle. More than one cycle for cache access keeps the cache busy for those many cycles and no other memory operation can proceed without completing the current access. The instructions dependent on these memory operations also get stalled. This multi-cycle-access latency reduces the bandwidth of cache and hurts processor performance.

One of the techniques to increase the bandwidth is pipelining. Pipelining divides the cache latency into multiple stages so that multiple accesses can be made simultaneously. The problem in pipelining the cache is that data voltage level in the bit-lines remains at a fraction of supply voltage making it difficult to latch the bit-line data. We explore the idea of banking to make pipelining feasible in caches. We propose a scheme that can be utilized to divide the cache access latency into several stages as required by the clock cycle time and bandwidth requirement (as technology scales). The technique is also helpful in designing large-sized caches to decrease the miss rate while providing high bandwidth.

Decoder contributes around 15–20% to overall cache hit time delay. Nogami et al. [1] proposed a technique to hide the full or partial decoder delay by putting the decoder in a pipeline stage. However, even in the best case such a decoder-pipelined cache is limited by the word-line driver to data-out delay. Because this delay is 75–80% of the total cache delay, the decoder-pipelined cache has an imbalanced pipeline stage, which degrades the bandwidth.

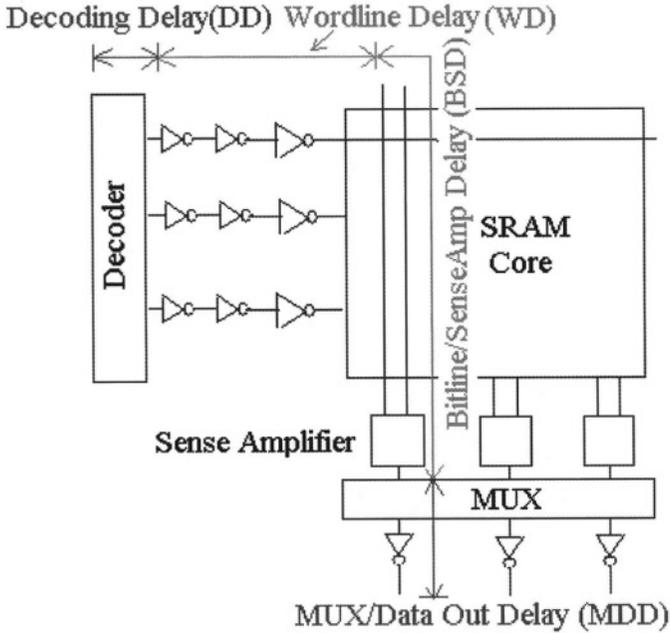
## 2. BANKING OF CACHE

The cache access time can be divided into four parts (Figure 26-1):

- a) Decoding Delay (DD);
- b) Word-line Delay (WD);
- c) Bit-line to Sense Amplifier Delay (BSD);
- d) Mux to Data Out Delay (MDD).

One of the main hit time reduction techniques for large caches is to bank the cache. The memory is partitioned into  $M$  smaller banks. An extra address word called bank address selects one of the  $M$  banks to be read or written. The technique reduces the word-line and bit-line (capacitance) and in turn reduces the WD and BSD. The bank address can be used to disable sense amplifiers and row and column decoders of un-accessed memory banks to reduce power dissipation.

The memory can be divided into smaller banks. Based on such banking, two parameters can be defined:  $N_{dwl}$  and  $N_{dbl}$  [2]. The parameter  $N_{dwl}$

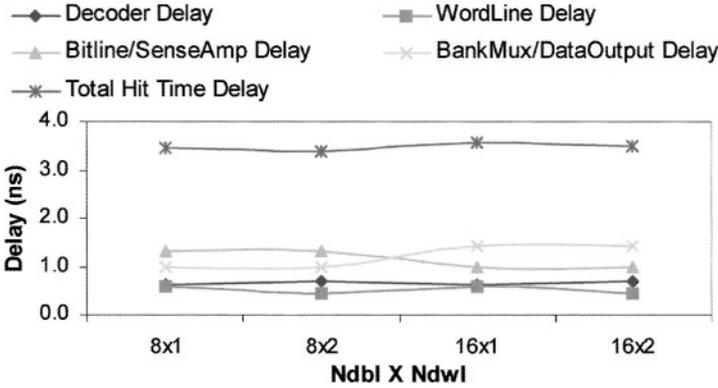


Colour picture

Figure 26-1. Cache access delay.

indicates the number of times the array has been split with vertical cut lines (creating more, but shorter word-lines), while  $N_{dbl}$  indicates the number of times the array is split with horizontal cut lines (creating shorter bit-lines). The total number of banks is  $N_{dwl} \times N_{dbl}$ . However, reduced hit time by increasing these parameters comes with extra area, energy and delay overhead. Increasing  $N_{dbl}$  increases the number of sense amplifiers, while increasing  $N_{dwl}$  translates into more word-line drivers and bigger decoder due to increase in the number of word-lines [4]. Most importantly, except in the case of a direct mapped cache (where  $N_{dbl}$  and  $N_{dwl}$  are both equal to one), a multiplexer is required to select the data from the appropriate bank. Increasing  $N_{dbl}$  increases the size of multiplexer, which in turn increases the critical hit time delay and energy consumption.

Figure 26-2 shows the variation of different components of hit time with respect to  $N_{dbl}$  and  $N_{dwl}$ . A 64 K and a 128 K, 2-way caches were simulated using TSMC 0.25  $\mu$  technology. Increasing  $N_{dbl}$  decreases BSD due to shorter bit-lines but it also increases MDD due to larger number of banks. More banking requires larger column multiplexer to select proper data. There is an optimum partitioning beyond which the increase in delay due to column multiplexer supercedes the delay gain due to reduced bit-lines. Increasing  $N_{dwl}$  does not make any difference in delay for 64 K cache for current technology generations. In a 64 K cache, a single word-line needs to have 2 cache blocks (64 byte) to maintain a good aspect ratio of the cache. Because



Colour picture

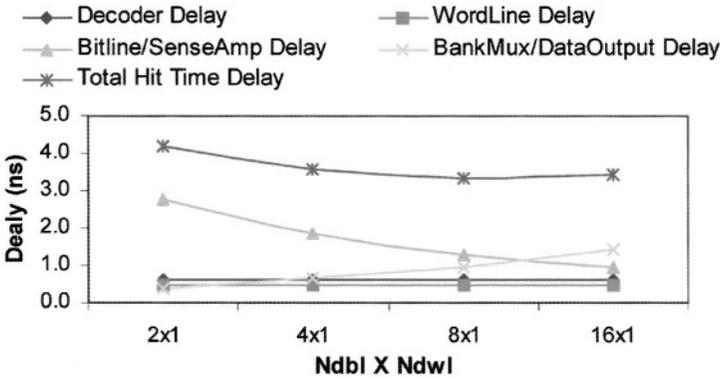


Figure 26-2. Delay analysis for different Ndbl | Ndwl. a) 64K Cache; b) 128K Cache

the cache under consideration in Figure 26-1 is a 2-way associative cache, which requires reading of two cache blocks simultaneously, changing Ndwl does not change the capacitance driven by word-line driver. For a 128 K cache, 4 cache blocks can be put in a single word-line. Dividing word-line into two parts decreases WDD, but increases DD due to the increased number of word-lines to be decoded. Several variations of the given architecture are possible. Variation includes positioning of sense amplifiers, the partitioning of the word and bit-lines, and the logic styles of the decoder used.

### 3. PROPOSED PIPELINED CACHE ARCHITECTURE

Pipelining is a popular design technique often used to increase the bandwidth of the datapaths. It reduces the cycle time of a block that may otherwise be governed by critical delay of the block. Unfortunately, unlike functional units, caches do not lend themselves to be pipelined to arbitrary depth. The key impediment to pipelining the cache into more stages is that the bit-line delay

cannot be pipelined because the signals on the bit-lines are weak, and not digital; latching can be done only after the sense amplifiers convert the bit-line signals from analog to digital. A bit-line is loaded by both multiple word-line cell capacitances and the bit-line wire capacitance and resistance. Consequently, the bit-line delay depends on the bank size, which is optimized for latency. In a latency-optimized cache, further dividing the cache (other than the decoder) into the two natural stages of bit-line + senseamps and multiplexor + data driver creates a substantially uneven split; the WD + BSD is a factor of two to four longer than the MDD step (Figure 26-2). Thus, the uneven split renders ineffective pipelining.

Partitioning the cache into multiple banks decreases BSD but increases MDD. Beyond a certain point more partitioning increases the multiplexer delay and it dominates the decrease in bit-line delay. It is interesting to observe that banking makes MDD delay to catch up the WD + BSD delay. Placing a latch in between divides the word-line to data-out delay into two approximately comparable sections. This technique has advantage of having more banks (low bit-line delay) than a conventional design permits. The increase in multiplexer delay due to aggressive banking is hidden in a pipeline stage. The clock cycle is governed by the word-line to sense amplifier delay (WD + BSD) and can be made smaller by aggressive banking. The technique can be used to aggressively bank the cache to get minimum possible WD + BSD which is limited by WD. For scaled technologies it might be necessary to bank the cache more so that BSD fits in a single clock cycle. Aggressive banking makes MDD to go beyond the point where it can no longer fit into a single cycle. For dealing with this problem, multiplexer can be designed as a tree and can itself be pipelined into multiple stages as required.

Figure 26-3 shows a block diagram of a fully pipelined cache. The cache is divided into three parts:

- a) Decoder;
- b) Word-line Driver to Sense Amplifier via Bit-lines;
- c) Bank Multiplexer to Data Output Drivers.

Cache is banked optimally so that word-line driver to sense amplifier delay is comparable to a clock cycle time. Bank multiplexer is further divided into more pipeline stages as required by clock cycle time requirement.

### 3.1. Sense amplifier latches

Sense amplifiers are duplicated based on the number of horizontal cuts,  $N_{dbl}$ . A latch is placed at both input and output path of every sense amplifier to pipeline both read and write delay. Both input and output data is routed through bank multiplexer and data output drivers. The output of the bank decoder is also used to drive the sense enable signal of sense amplifiers. Only the accessed bank's sense amplifiers are activated and take part in sensing the data while the rest of them remain idle and do not increase dynamic energy.



latches to synchronize the pipeline operation. To reduce the redundant clock power, clock gating was used. No extra control logic is required for clock gating. The outputs of bank decoder themselves act as control signals to gate the clock to un-accessed latches.

### **3.2. Decoder latches**

All bank, row, and column decoders' outputs are also latched. Increasing the number of vertical cuts (N<sub>dwl</sub>) increases the number of word-lines to decode. This requires a bigger row decoder and more number of latches. At the same time increasing the number of banks also increases the size of bank decoder and hence, the number of latches. However, at most only two lines switch in a single read/write operation. These two lines are: the one that got selected in previous cycle and the one that is going to be selected in current cycle. Hence, not all latches contribute to dynamic energy dissipation.

### **3.3. Multiplexer latches**

To pipeline the multiplexer, sets of latches are placed at appropriate positions so that the multiplexer can be divided into multiple stages. The required number of latches depends on the aggressiveness of banking and how deep the latches are placed in the multiplexer tree. Similar to sense amplifier latches bank decoder output can be used to gate the clock in unused multiplexer latches.

## **4. RESULTS**

In this section we evaluate the performance of conventional unpipelined cache with respect to technology scaling. We show the effectiveness of the proposed design in maintaining the pipeline stage delay within the clock cycle time bound. We extracted HSPICE netlist from the layout of a 64 K cache using TSMC 0.25  $\mu$  technology. We scaled down the netlist for 0.18, 0.13, 0.10 and 0.07  $\mu$  technology and used BPTM [6] (Berkeley Predictive Technology Model) for simulating the cache for scaled technology.

### **4.1. Cache access latency vs CPU cycle**

Current technology scaling trend shows that CPU clock frequency is getting doubled each generation. However, cache access latency does not scale that well because of long bit-lines. Figure 26-4 shows the difference between clock frequency and conventional unpipelined cache access frequency for different technology generations. Here we assume that for 0.25  $\mu$  technology the cache is accessed in one clock cycle. As technology is scaled, the clock cycle frequency is doubled. The cache is banked optimally to get minimum possible

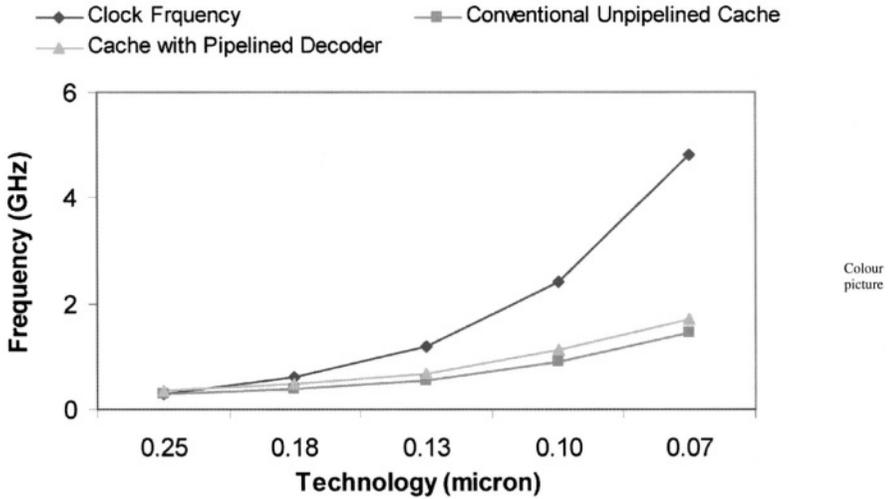


Figure 26-4. Cache access frequency vs. clock cycle frequency.

cache delay. Figure 26-4 shows that as technology scales down, the gap between clock frequency and cache access frequency widens. Multiple clock cycles are required to access the cache (Table 26-1). Hence, cache cannot be accessed in every clock cycle. This reduces the bandwidth of cache and leads to processor stalls and loss in overall performance.

The third column of Table 26-1 shows the number of clock cycles required to access the cache with pipelined decoder [1]. For the design under consideration, the cache has two pipeline stages. First stage (decoder) is accessed in one clock cycle while second stage is accessed in the number of cycles given in Table 26-1. Putting decoder in pipeline reduces the clock cycle time requirement and enhances the bandwidth. However, the cache is still not capable of limiting the overall cache delay in pipeline stages to single clock cycle (Figure 26-4).

The proposed pipeline technique is implemented in 64 K cache for different technology generations. To scale the bit-lines, the cache is divided into multiple banks. The overall cache delay is divided into three parts: decoder delay (DD), word-line to sense amplifier delay (WD + BSD) and multiplexer

Table 26-1. The number of clock cycle vs. technology.

Technology ( $\mu$ )	Unpipelined conventional cache	Cache with pipelined decoder
0.25	1	1
0.18	2	2
0.13	3	2
0.10	3	3
0.07	4	3

to data output delay (MDD). Figure 26-5 shows the variation of these delays with banking for different technology generations. The delay for a particular technology is normalized with respect to the clock cycle time for that

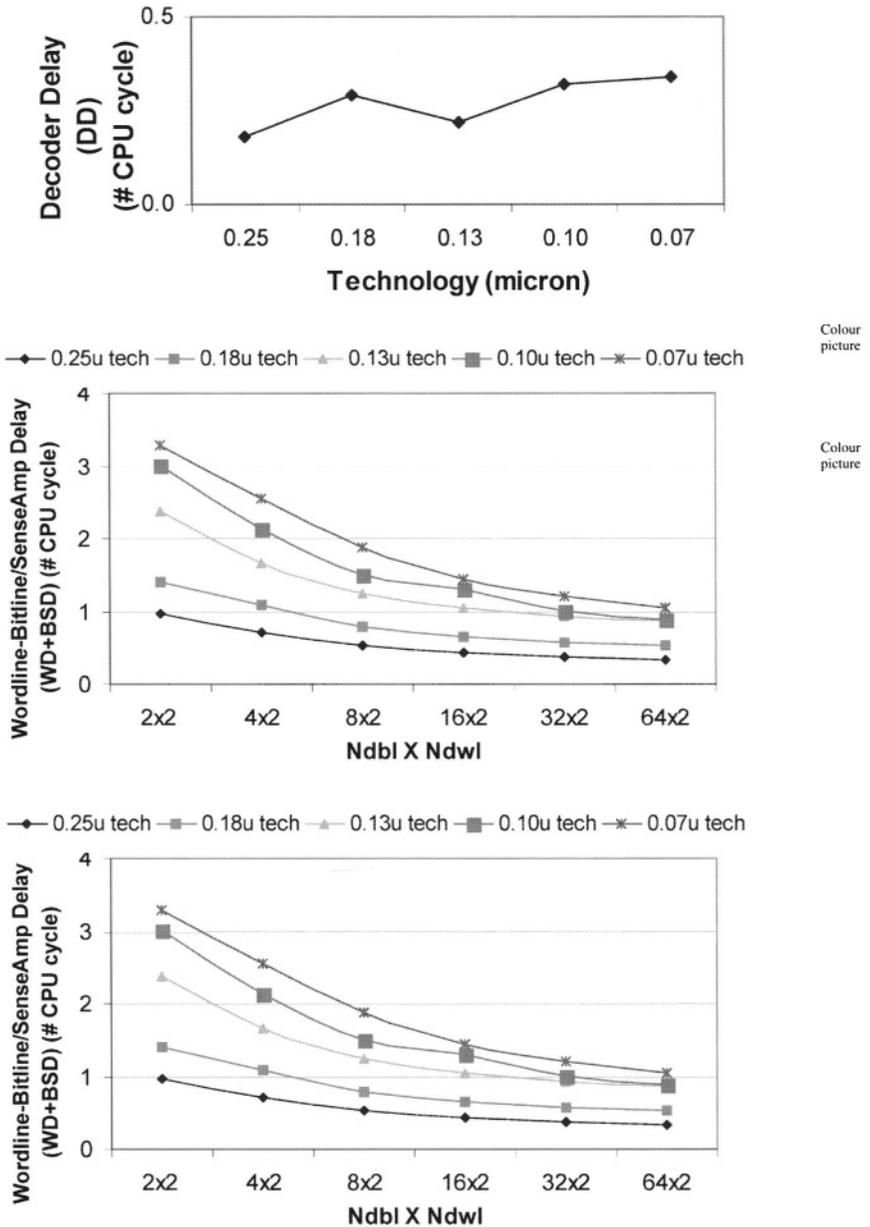


Figure 26-5. Delay analysis. a) Decoder delay; b) Word-line to sense amplifier delay via bit-line; c) Mux to Data output delay.

technology as given in Figure 26-4. To account for latch overhead, latch setup time and propagation delay are added to the WD + BSD and MDD. This overhead is around 10% of the clock cycle time for a given technology. DD remains within the bound of one clock cycle irrespective of technology scaling. Since  $N_{dwl}$  is kept constant, banking the cache does not affect DD. WD + BSD does not scale well with technology. Scaling down the technology requires more clock cycles for a given bank configuration. For example, the number of cycles required for  $4 \times 2$  bank configuration is one for  $0.25 \mu$  and  $0.18 \mu$  technology, whereas for  $0.13 \mu$  and  $0.10 \mu$  it is two cycles, and for  $0.07 \mu$  technology it is three cycles (Figure 26-4).

More banking makes the bit-line delay to go down. The goal behind banking is to get an optimal configuration for which the word-line driver to sense amplifier delay (with latch overhead) is equal a single CPU cycle. Overbanking will increase energy and area overhead without any performance improvement. Figure 26-5(b) shows that for a  $0.25 \mu$  technology, the bit-line delay remains within the limit of one cycle for all bank configurations. Scaling the technology requires more banking to achieve one cycle bit-line delay, e.g. for  $0.18 \mu$ , optimal banking is  $8 \times 2$ . Similarly for  $0.13 \mu$ ,  $0.10 \mu$ , and  $0.07 \mu$  technologies, optimal banking configurations are  $16 \times 2$ ,  $32 \times 2$ , and  $64 \times 2$ , respectively.

For optimal banking, WD + BSD can be confined to a single clock cycle; however, it increases MDD. Since MDD consists of wire delay due to routing of the banks, it also does not scale effectively with technology. In the proposed scheme, the MDD is determined by the optimal banking configuration for a bit-line delay. Analyzing MDD (Figure 26-5(c)) for the optimal configurations shows that for  $0.25 \mu$ ,  $0.18 \mu$ , and  $0.13 \mu$  technology, MDD remains within one clock cycle. For  $0.10 \mu$  and  $0.07 \mu$  technologies, optimal bank configurations, decided by bit-line delay, are  $32 \times 2$  and  $64 \times 2$  respectively, for which required MDD goes beyond one clock cycle. To make the cache fully pipelined with access frequency equivalent to clock frequency, the multiplexer can be divided into multiple pipeline stages. Table 26-2 shows the optimal banking requirement and the number of pipeline stages required to make the cache fully pipelined with access frequency of one clock cycle.

The design technique has its own disadvantages of having extra energy and area overhead. Table 26-3 shows the area and energy overhead associated with

Table 26-2. Number of cache pipeline stages.

Technology ( $\mu$ )	Banks ( $N_{dbl} \times N_{dwl}$ )	Decoder stage	WL-sense amp stage	Mux to dataout stages	Total cache pipeline stages
0.18	$8 \times 2$	1	1	1	3
0.13	$16 \times 2$	1	1	2	3
0.10	$32 \times 2$	1	1	2	4
0.07	$64 \times 2$	1	1	2	4

Table 26-3. Performance, energy and area estimates.

Tech ( $\mu$ )	% incr area	% incr read energy	% incr write energy	% incr in MOPOS with respect to conventional cache			% incr in MOPS with respect to cache with pipelined decoder		
				BBR	BBW	ARW	BBR	BBW	ARW
0.18	3.3	3.1	3.7	41.1	16.5	-6.5	15.9	3.9	-19.1
0.13	7.5	11.2	12.4	85.0	48.9	20.7	50.5	32.0	3.6
0.10	15.3	23.1	25.4	85.4	49.3	21.0	51.8	32.4	4.2
0.07	31.3	54.4	60.6	68.1	42.9	13.4	47.7	32.2	3.0

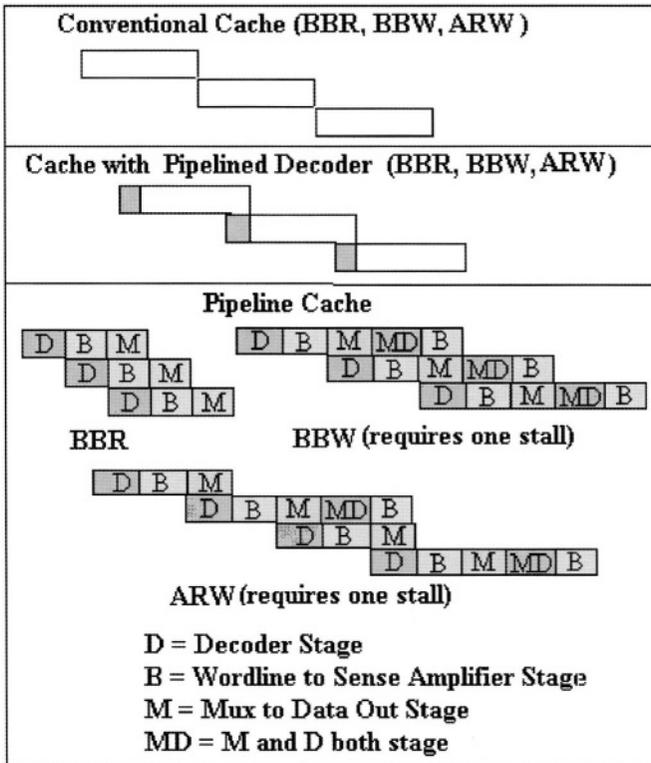
pipelining for different technology generations. The energy estimate considers the energy overhead due to extra latches, multiplexers, clock and decoders. It also accounts for the decrease in precharge energy as the bit-line capacitance is reduced. Increase in area due to extra precharge circuitry, latches, decoders, sense amplifiers, and multiplexer is considered in this analysis. To account for area and energy overhead together with performance gain we use MOPS (million of operation per unit time per unit area per unit energy) as a metric for comparison. Three scenarios have been considered to calculate MOPS:

- Back to Back Read (BBR) operations;
- Back to Back Write (BBW) operations;
- Alternate Read and Write (ARW).

Figure 26-6 shows the pipeline stalls required due to resource conflict for all three scenarios. In the case of conventional unpipelined cache, both read and write have to wait until previous read or write is finished. In a cache with pipelined decoder, DD is hidden into previous read or write access.

In the proposed pipeline design read access, is divided into three stages (Figure 26-6). In the case of BBR, a read comes out of the cache in every clock cycle. A write is a read followed by a write and that requires five stages. Issuing write back-to-back encounters a structural hazard due to multiple access of the multiplexer stage in single write. Also the fourth stage requires the multiplexer and the decoder to be accessed simultaneously causing resource conflict in the pipeline. Issuing another write after two cycles of previous write resolves this hazard. Similar hazard is there in the case of ARW. Again issuing next read or write after two cycles of previous read or write resolves this hazard. Fortunately, similar stalls are required for the case when multiplexer is pipelined and accessed in two clock cycles (0.10 and 0.07  $\mu$  technology). MOPS achieved by the pipelined cache is compared with the conventional and the cache with pipelined decoder.

Table 26-3 shows the percentage improvement in MOPS achieved by pipelining cache. For 0.18  $\mu$  technology pipeline cache achieves 41% and 15.9% improvement in MOPS in the case of BBR and 16.5% and 3.86%



Colour picture

Figure 26-6. Stalls require for BBR, BBW, and ARW.

improvement in MOPS in the case of BBW with respect to conventional cache and the cache with pipelined decoder, respectively. In the case of ARW, MOPS is lower for the proposed cache. For other technologies pipeline cache achieves significant improvement in MOPS ranging from 68.1–85.4% and 47.7–51.8% in the case of BBR, 42.9–49.3% and 32.0–32.4% in the case of BBW, and 13.4–21.0% in the case of ARW with respect to conventional unpipelined cache and the cache with pipelined decoder, respectively. The results show the effectiveness of the proposed methodology in designing a scalable and pipelined cache that can be accessed every clock cycle.

#### 4.2. Increasing the size of the cache

With growing need for higher performance, processors need larger cache to deal with large and independent workloads. Small caches cannot hold sufficient data that is required frequently by independent programs and leads to capacity and conflict misses. Increasing the size of the cache increases the access time. The proposed pipelined technique can be applied to design a large cache whose pipeline stage delay is equal to clock cycle time.

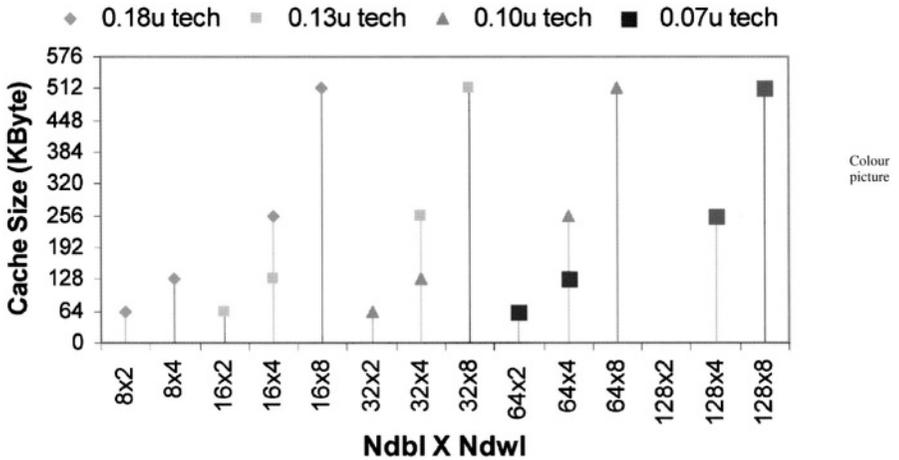


Figure 26-7. Optimal banking required for pipelining the large size cache (64, 128, 256, and 512 KB) for different technology.

Figure 26-7 shows the optimal banking required for pipelining large caches for different technology generations. Increasing the cache size increases either the number of rows or the number of columns. Increase in delay due to extra rows and extra columns can be reduced by increasing Ndbl and Ndwl, respectively. Increasing Ndwl divides the number of columns into multiple sections, creating more but shorter word-lines. It requires a bigger decoder. However, decoder stage delay for 64 K cache is much smaller than clock cycle time (Figure 26-5). Hence, there is sufficient slack available to merge the extra delay due to bigger decoder. Ndbl divides the bit-line capacitance and reduces the bit-line to sense amplifier delay.

### 5. CONCLUSIONS

In this article we explored a design technique to effectively pipeline caches for higher bandwidth. Pipelining is made possible by aggressively banking the cache which makes word-line to sense amplifier delay to fit into a single clock cycle. The technique is capable of splitting the cache into three, four or more pipeline stages as required by clock cycle time requirement. A large cache, having cache access frequency equal to clock frequency, can be designed by carefully banking the cache. The proposed pipeline cache dominates other designs in terms of the MOPS measure. The technique is fully scalable and very effective in pipelining future cache designs.

## ACKNOWLEDGEMENT

The research was funded in part by SRC (98-HJ-638) and by DARPA.

## REFERENCES

1. K. Naogami, T. Sakurai et. al. "A 9-ns Hit-Delay 32-kbyte Cache Macro for High Speed RISC." *IEEE Journal of Solid State Circuits*, Vol. 25, No. 1. February 1990.
2. T. Wada and S. Rajan. "An Analytical Access Time Model for On-Chip cache Memories." *IEEE Journal of Solid State Circuits*, Vol. 27, No. 8, pp. 1147–1156, August 1992.
3. J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann, 2nd Edition.
4. S. J. E. Wilson and N. P. Jouppi. "An Enhanced Access and Cycle Time Model for On-Chip Caches." *Technical Report 93/5*, Digital Equipment Corporation, Western Research Laboratory, July 1994.
5. J. M. Rabaey. *Digital Integrated Circuit*. Prentice Hall, 1996.
6. <http://www-device.eecs.berkeley.edu/~ptm/>.
7. D. M. Tullsen, S. J. Eggers, and H. M. Levy. "Simultaneous Multithreading: Maximizing On-Chip Parallelism." In *Proceedings of the 22th Annual International Symposium on Computer Architecture*, pp. 392–403, June 1995.

# ENHANCING SPEEDUP IN NETWORK PROCESSING APPLICATIONS BY EXPLOITING INSTRUCTION REUSE WITH FLOW AGGREGATION

G. Surendra, Subhasis Banerjee, and S. K. Nandy  
*Indian Institute of Science*

**Abstract.** Instruction Reuse (IR) is a microarchitectural technique that improves the execution time of a program by removing redundant computations at run-time. Although this is the job of an optimizing compiler, they do not succeed many a time due to limited knowledge of run-time data. In this article we concentrate on integer ALU and load instructions in packet processing applications and see how IR can be used to obtain better performance. In addition, we attempt to answer the following questions in the article – (1) Can IR be improved by reducing interference in the *reuse buffer*?, (2) What characteristics of network applications can be exploited to improve IR?, and (3) What is the effect of IR on resource contention and memory accesses? We propose an aggregation scheme that combines the high-level concept of network traffic i.e. “flows” with the low level microarchitectural feature of programs i.e. repetition of instructions and data and propose an architecture that exploits temporal locality in incoming packet data to improve IR by reducing interference in the RB. We find that the benefits that can be achieved by exploiting IR varies widely depending on the nature of the application and input data. For the benchmarks considered, we find that IR varies between 1% and 50% while the speedup achieved varies between 1% and 24%.

**Key words:** network processors, instruction reuse, flows, and multithreaded processors

## 1. INTRODUCTION

Network Processor Units (NPU) are specialized programmable engines that are optimized for performing communication and packet processing functions and are capable of supporting multiple standards and Quality of service (QoS) requirements. Increasing network speeds along with the increasing desire to perform more computation within the network have placed an enormous burden on the processing requirements of NPUs. This necessitates the development of new schemes to speedup packet processing tasks while keeping up with the ever-increasing line rates. The above aim has to be achieved while keeping power requirements within reasonable limits. In this article we investigate dynamic Instruction Reuse (IR) as a means of improving the performance of a NPU. The motivation of this article is to determine if IR is a viable option to be considered during the design of NPUs and to

evaluate the performance improvement that can be achieved by exploiting IR.

Dynamic Instruction Reuse (IR) [1, 2] improves the execution time of an application by reducing the number of instructions that have to be executed dynamically. Research has shown that many instructions are executed repeatedly with the same inputs and hence producing the same output [3]. Dynamic instruction reuse is a scheme in which instructions are buffered in a *Reuse Buffer* (RB) and future dynamic instances of the same instruction use the results from the RB if they have the same input operands. The RB is used to store the operand values and result of instructions that are executed by the processor. This scheme is denoted by  $S_v$  ('v' for value) and was proposed by Sodani and Sohi [1]. The RB consists of *tag*, *input operands*, *result*, *address* and *memvalid* fields [1]. When an instruction is decoded, its operand values are compared with those stored in the RB. The PC of the instruction is used to index into the RB. If a match occurs in the *tag* and *operand* fields, the instruction under consideration is said to be reused and the result from the RB is utilized i.e. the computation is not required. We assume that the reuse test can be performed in parallel with instruction decode [1] and register read stage. The reuse test will usually not lie in the critical path since the accesses to the RB can be pipelined. The *tag* match can be initiated during the instruction fetch stage since the PC value of an instruction is known by then. The accesses into the operand fields in the RB can begin only after the operand registers have been read (register read stage in the pipeline). Execution of load instructions involves a memory address computation followed by an access to the memory location specified by the address. The address computation part of a load instruction can be reused if the instruction operands match an entry in the RB, while the actual memory value (outcome of load) can be reused if the addressed memory location was not written by a store instruction [1]. The *memvalid* field indicates whether the value loaded from memory is valid (i.e. has not been overwritten by store instruction) while the *address* field indicates the memory address. When the processor executes a store instruction, the *address* field of each RB entry is searched for a matching *address*, and the *memvalid* bit is reset for matching entries [1]. In this article we assume that the RB is updated by instructions that have completed execution (non-speculative) and are ready to update the register file. This ensures that precise state is maintained with the RB containing only the results of committed instructions. IR improves performance since reused instructions bypass some stages in the pipeline with the result that it allows the dataflow limit to be exceeded. Performance improvement depends on the number of pipeline stages between the register read and writeback stages and is significant if long latency operations such as divide instructions are reused. Additionally, in case of dynamically scheduled processors, performance is further improved since subsequent instructions that are dependent on the reused instruction are resolved earlier and can be issued earlier (out-of-order issue).

The following are the main contributions of this article – (i) We evaluate some packet processing benchmarks and show that repetition in instruction and data values are common (specially in header processing applications) with the result that significant IR can be exploited, (ii) We propose a flow aggregation scheme that exploits the locality in packet data to improve IR – the idea is to store reuse information of *related* packets (flows) in separate RB's so that interference in the RB's is reduced – and (iii) We analyze the impact of IR on resource contention and memory accesses for the packet processing benchmarks.

The rest of the article is organized as follows. We describe the flow aggregation approach in section 2, present simulation results in section 3 and conclude with section 4.

## 2. IMPROVING IR BY AGGREGATING FLOWS

A flow may be thought of as a sequence of packets sent between a source/destination pair following the same route in the network. A router inspects the Internet Protocol (IP) addresses in the packet header and treats packets with the same source/destination address pairs as belonging to a particular flow. A router may also use application port (layer 4) information in addition to the IP addresses to classify packets into flows. When packets with a particular source/destination IP address pair traverse through an intermediate router in the network, one can expect many more packets belonging to the same flow (*i.e.* having the same address pair) to pass through the same router (usually through the same input and output ports) in the near future. All packets belonging to the same flow will be similar in most of their header (both layer 3 and layer 4) fields and many a time in their payload too. For example, the source/destination IP addresses, ports, version and protocol fields in an IP packet header will be the same for all packets of a particular connection/session. Packet header processing applications such as firewalls, route lookup, network address translators, intrusion detection systems *etc.*, are critically dependent on the above fields and yield higher reuse if flow aggregation is exploited. IR can be improved if applications that process these packets somehow identify the flow to which the packet belongs to, and uses different RB's for different flows. The idea is to have multiple RB's, each catering to a flow or a set of flows so that similarity in data values (at least header data) is preserved ensuring that evictions in the RB is reduced.

A simple example (Figure 27-1) is used to illustrate how flow aggregation reduces the possibility of evictions in a RB and enhances reuse. For simplicity, let us assume that an ALU operation (say addition) is computed by the NPU on incoming packet data  $\mathbf{a}_i$ ,  $\mathbf{b}_i$ . Figure 27-1 (a) shows the reuse scheme without flow aggregation using a single 4 entry RB while Figure 27-1(b) exploits flow aggregation using two RB's. Assume that incoming packets are classified into two flows –  $\mathbf{Pkt}_1$ ,  $\mathbf{Pkt}_2$  and  $\mathbf{Pkt}_6$  belong to  $\mathbf{flow}_A$  while  $\mathbf{Pkt}_3$ ,  $\mathbf{Pkt}_4$ , and  $\mathbf{Pkt}_5$

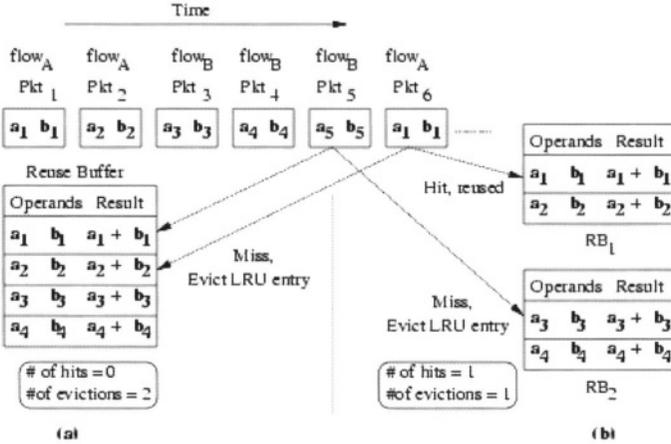


Figure 27-1. An example comparing (a) ordinary instruction reuse with (b) flow based reuse

belong to **flow<sub>B</sub>**. The RB (Figure 27-1(a)) is updated by instructions that operate on the first four packets. When the contents of **Pkt<sub>5</sub>** is processed (there is no hit in the RB), the LRU (Least Recently Used) entry (i.e. **a<sub>1</sub>, b<sub>1</sub>**) is evicted and overwritten with **a<sub>5</sub>, b<sub>5</sub>**. This causes processing of the next packet **Pkt<sub>6</sub>** to also miss (since the contents **a<sub>1</sub>, b<sub>1</sub>** were just evicted) in the RB. We have used an example in which **Pkt<sub>6</sub>** has the same values as that of **Pkt<sub>1</sub>** which is quite possible since they belong to the same flow. Assume that a flow aggregation scheme is used with multiple RB's so that program instructions operating on packets belonging to **flow<sub>A</sub>** query RB<sub>1</sub> and those operating on **flow<sub>B</sub>** query RB<sub>2</sub> for exploiting reuse. Instructions operating on **Pkt<sub>5</sub>** will be mapped to RB<sub>2</sub> (which will be a miss) while instructions operating on **Pkt<sub>6</sub>** mapped to RB<sub>1</sub> will cause a hit and enable the result to be reused leading to an overall improvement in reuse (compared to Figure 27-1 (a)). The amount of IR that can be uncovered depends of the nature of the traffic and data values, and it is quite possible that IR could decrease if smaller RB's are used.

One can relax the previous definition and classify packets related in some other sense as belonging to a flow. For instance, the input port through which packets arrive at a router and the output port through which packets are forwarded could be used as possible alternatives. This is a natural way of flow classification since in most circumstances, packets of the same flow travel along the same path from the source to the destination. Although a very accurate classification scheme is not required, consistency must be maintained for appreciable improvement in results. For every packet that arrives, the NPU must determine the RB to be associated with instructions that operate on that packet. Routers that classify packets based on the IP addresses are required to parse the IP header and maintain state information for every flow. Flow classification based on the output port involves some computation to determine the output port. The output port is determined using the Longest Prefix

Match (LPM) algorithm and is computed for every packet irrespective of whether IR is exploited or not [6]. Most routers employ *route caches* to minimize computing the LPM for every packet thereby ensuring that the output port is known very early [6]. Classification of flows based on the input port involves little or no computation (since the input port through which a packet arrives is known) but uncovers a smaller percentage of reuse for some applications. In case of routers that have a large number of ports, a many-to-one mapping between ports and RB's to be queried by instructions is necessary to obtain good results.

### 2.1. Proposed architecture

For single processor single threaded systems the architecture proposed in [1] with a few minor changes is sufficient to exploit flow-based IR. However, for multiprocessor and multithreaded systems, which are generally used in designing NPUs, extra modifications are required. The essence of the problem at hand is to determine the appropriate RB to be queried by instructions operating on a packet and switch between RB's when necessary. The NPU is essentially a chip multiprocessor consisting of multiple RISC processors (or micro-engines – using the terminology of Intel IXP1200 [7]) with support for hardware multithreading. It is the job of the programmer to partition tasks across threads as well as programs across micro-engines. The inter-thread and inter-processor communication is explicitly managed by the user. Figure 27-2 shows the microarchitecture of a single processing engine. Each processor has a RB array consisting of  $N + 1$  RB's –  $RB_0, \dots, RB_N$ .  $RB_0$  is the default RB that is queried by instructions before the *flow id* of a packet is computed (for output port-based scheme only). The NPU also consists of a Flow identifier table that is updated by the *flow id* for a given packet

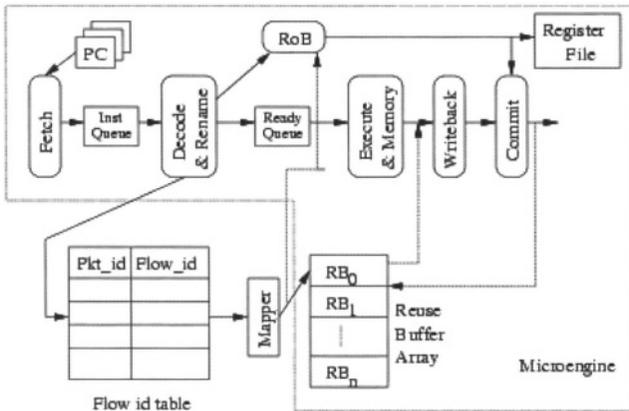


Figure 27-2. A possible microarchitecture to exploit instruction reuse using the flow aggregation scheme.

identifier and a mapper (described in the previous section) that identifies the RB to be used by instructions operating on that packet. Static mapping of ports and RB's is a good (though not optimal at all times) scheme that can be used to reduce hardware complexity. The Flow identifier table and the mapper are accessible to all microengines of the NPU and to the memory controller, which is responsible for filling an entry on the arrival of a new packet.

The *flow id* field is initialized to a default value (say 0) which maps to the default reuse buffer  $RB_0$ . This field is updated once the actual *flow id* is computed based on any of the schemes mentioned previously. One scheme for implementing flow based IR is to tag packets that arrive at the router or NPU. This tag serves as a packet identifier, which after mapping, becomes the RB identifier and is used by instructions operating on a packet to query the appropriate RB. Another scheme would be one in which the *packet id* of the packet currently being processed by a thread is obtained by tracking the memory being accessed by read instructions (we assume that a packet is stored in contiguous memory). Each thread stores the *packet id* of the packet currently being processed, the *thread id* and the *flow id* to which the packet belongs. The selection of the RB based on the *flow id* is made possible by augmenting the Reorder Buffer (RoB) with the *thread id* and the RB id (the mapper gives the *flow id* to RB id mapping). Instructions belonging to different threads (even those in other processors) access the Flow identifier table which indicates the RB to be queried by that instruction. The *flow id* field indicates the default RB ( $RB_0$ ) initially. After a certain amount of processing, the thread that determines the output port (for output-port based scheme) updates the *flow id* entry in the Flow identifier table for the packet being processed. This information is known to all other threads operating on the same packet through the centralized Flow identifier table. Once the *flow id* is known, the mapper gives the exact RB id (RB to be used) which is stored in thread registers as well as in the RoB. When the processing of the packet is complete, it is removed from memory i.e. it is forwarded to the next router or sent to the host processor for local consumption. This action is again initiated by some thread and actually carried out by the memory controller. At this instant the *flow id* field in the Flow identifier table for the packet is reset to the default value. In summary, IR is always exploited with instructions querying either the default RB or a RB specified by the *flow id*. However, the centralized Flow identifier table could be the potential bottleneck in the system. It is also possible to use a hashing function that sends all packets of the same flow to the same processing engine in which case, the hardware complexity in determining the RB to be queried reduces significantly. The drawback of this scheme is that the processing engines could be non-uniformly loaded resulting in performance degradation.

## 2.2. Impact of IR on resources

Since the result value of a reused instruction is available from the RB, the execution phase is avoided reducing the demand for resources. In case of load instructions, reuse (outcome of load being reused) reduces port contention, number of memory accesses [4] and bus transitions. In most current day processors that have hardware support for gating, this could lead to significant savings in energy in certain parts of the processor logic. This could be significant in in-order issue processors, where instructions dependent on reused instructions cannot be issued earlier. In dynamically scheduled processors, IR allows dependent instructions to execute early which changes the schedule of instruction execution resulting in clustering or spreading of request for resources, thereby, increasing or decreasing *resource contention* [2]. Resource contention is defined as the ratio of the number of times resources are not available for executing ready instructions to the total number of requests made for resources.

## 2.3. Operand based indexing

Indexing the RB with the PC enables one to exploit reuse due to dynamic instances of the same static instruction. Indexing the RB with the instruction opcode and operands can capture redundancy across dynamic instances of statically distinct instructions (having the same opcode). One way to implement operand indexing (though not optimal) is to have an opcode field in addition to other fields mentioned previously in the RB and search in parallel the entire RB for matches. In other words, an instruction that matches in the opcode and operand fields can read the result value from the RB. This is in contrast to PC based indexing where the associative search is limited to a portion of the RB. We use a method similar to the one proposed in [5] to evaluate operand based indexing. Operand indexing helps in uncovering slightly more reuse than PC based indexing (for the same RB size). This can be attributed to the fact that in many packet-processing applications, certain tasks are often repeated for every packet. For example, IP header checksum computation is carried out to verify a packet when it first arrives at a router. When packet fragmentation occurs, the IP header is copied to the newly created packets and the checksum is again computed over each new packet. Since there is significant correlation in packet data, the inputs over which processing is done is quite limited (in the above example, the checksum is repeatedly computed over nearly the same IP header) and hence packet (especially header) processing applications tend to reuse results that were obtained while processing a previous packet.

### 3. SIMULATION METHODOLOGY AND RESULTS

The goal of this article is to demonstrate the idea and effectiveness of aggregating flows to improve IR and not the architecture for enabling the same. Hence, we do not simulate the architecture proposed in the previous section (this will be done as future work) but use a single threaded single processor model to evaluate reuse. This gives us an idea of the effectiveness of flow aggregation in improving IR. We modified the SimpleScalar [8] simulator (MIPS ISA) and used the default configuration [8] to evaluate instruction reuse on a subset of programs representative of different classes of applications from two popular NPU benchmarks – CommBench [9] and NetBench [10] (see Table 27-1). It must be noted that we use SimpleScalar since it is representative of an out-of-order issue pipelined processor with dynamic scheduling and support for speculative execution. In other words, we assume that the NPU is based on a superscalar RISC architecture which is representative of many NPUs available in the market. Further, using SimpleScalar makes it easy to compare results with certain other results in [9] and [10] that also use the same simulation environment. When the PC of an instruction matches the PC of the function that reads a new packet, the output port for the packet is read from a precomputed table of output ports. This identifies the RB to be used for the current set of instructions being processed. The RB identifier is stored in the RoB along with the operands for the instruction and the appropriate RB is queried to determine if the instruction can be reused. The results therefore obtained are valid independent of the architecture proposed in the previous section. Since threads and multiple processors are not considered during simulation, the results reported in this article give an upper bound on the speedup that can be achieved by exploiting flow aggregation. However, the bound can be improved further if realistic network traffic traces that are not anonymized in their header and

*Table 27-1.* Reuse and speedup without flow aggregation for different RB configurations. R = % instructions reused, S = % improvement in speedup due to reuse. % Speedup due to operand indexing and % reduction in memory traffic for a (32,8) RB is shown in the last two columns.

Benchmark	32,4 R	32,4 S	128,4 R	128,4 S	1024,4 R	1024,4 S	Op- index	Mem Traffic
FRAG	7.9	3.7	20.4	4.9	24.4	8.3	5.3	42.1
DRR	12.6	0.16	15.5	0.5	18.2	0.86	0.4	11.6
RTR	15.2	3.8	33.2	6.1	47.6	8.1	9.2	71.3
REEDENC	19.8	2	20.3	2.05	25.2	2.95	4.7	8.7
REED DEC	6.6	1.76	11.8	4	16.6	5.6	6	4.9
CRC	19.1	19.6	20.7	19.84	21.8	19.84	20.4	35.1
MD5	1.4	1.3	3.5	2.3	14.2	8.3	1.6	34.3
URL	18.8	9.4	19.9	11.2	22.2	12.7	13.1	42

payload are available. Inputs provided with the benchmarks were used in all the experiments except FRAG for which randomly generated packets with fragment sizes of 64, 128, 256, 512, 1024, 1280 and 1518 bytes were used [11]. We evaluate instruction reuse for ALU and Load instructions for various sizes of the RB. We denote the RB configuration by the tuple  $(x,y)$  where “x” represents the size of the RB (number of entries) and “y” the associativity (number of distinct operand *signatures* per entry), “x” takes on values of 32, 128 and 1024 while “y” takes on values of 1, 4 and 8. Statistics collection begins after 0.1 million instructions and the LRU policy is used for evictions since it was found to be the best scheme.

### 3.1. Reuse and speedup – base case

Table 27-1 shows IR uncovered by different RB configurations and speedup for the benchmark programs considered. We find that packet-processing applications on average have slightly lesser instruction repetition patterns compared to SPEC benchmark results [1]. The locality of data in network traffic varies depending on where packets are collected. For example, we can expect high locality at edge routers while the actual advantage that can be derived from locality depends on the working set sizes which is usually large in most applications [12]. It should also be noted that there is no direct relation between hits in the RB and the speedup achieved. This is because speedup is governed not only by the amount of IR uncovered but also by the availability of free resources and the criticality of the instruction being reused. Though resource demand is reduced due to IR, resource contention becomes a limiting factor in obtaining higher speedup. An appropriate selection of both the number entries in the RB and the associativity of the RB are critical in obtaining good performance improvements. DRR is rather uninteresting yielding a very small speedup even with an (1024,8) RB. A closer examination of the DRR program reveals that though it is loop intensive, it depends on the packet length which varies widely in our simulations. A more significant reason for the small speedup gain is due to the high resource contention (Figure 27-4). On increasing the number of integer ALU units to 6, multiply units to 2 and memory ports to 4 (we shall refer to this configuration as the *extended configuration*), a speedup of around 5.1 % was achieved for a (32,8) RB.

### 3.2. Speedup due to flow aggregation

The IR and speedup that can be achieved due to flow aggregation depends significantly on traffic pattern. We carried out the simulation using 8 network interfaces (ports) with randomly generated addresses and network masks (mask length varies between 16 and 32). We use separate RB’s for ALU and load instructions since only load instructions utilize the *memvalid* and *address* fields in the RB which need not be present for ALU instructions. We find

that the flow aggregation scheme with 2 RB's is sufficient to uncover significant IR (though we use 4 in this article) for most benchmarks considered. We report results only for those cases for which the returns are considerable.

The flow aggregation scheme is based on the output port with a simple mapping scheme (for RTR we use the input port scheme since this is the program that computes the output port). We map instructions operating on packets destined for port 0 and 1 to RB<sub>1</sub>, 2 and 3 to RB<sub>2</sub> and so on. This type of mapping is clearly not optimal and better results can be expected if other characteristics of the network traffic are exploited. Since most traffic traces are anonymized, this kind of analysis is difficult to carry out and we do not explore this design space. Figure 27-3 shows the speedup results due to flow aggregation for FRAG and RTR programs. Flow aggregation is capable of uncovering significant amount of IR even when smaller RB's are used (this is highly dependent on the input data). For example, for the FRAG program, five RB's with (128,8) configuration results in the same speedup as a single RB with a (1024,8) configuration. We carried out experiments with other traces and obtained varying amounts of IR and speedup. While IR invariably increases due to the flow aggregation scheme, speedup, being dependent on other factors (see section 3.1), shows little or no improvement in many cases. The solid lines represent speedup due to ALU instructions in the base case while the dotted lines show results due to the flow-based scheme for ALU instructions only. The dashed lines indicate the additional contribution made by load instructions to the overall speedup. To examine the effect of reducing resource contention on speedup, we tried the *extended configuration* and obtained a 4.8% improvement in speedup for RTR (2.3% for FRAG) over the flow-based scheme (with (32,8) RB). Determining the IR and speedup due to flow aggregation for payload processing applications is rather difficult since

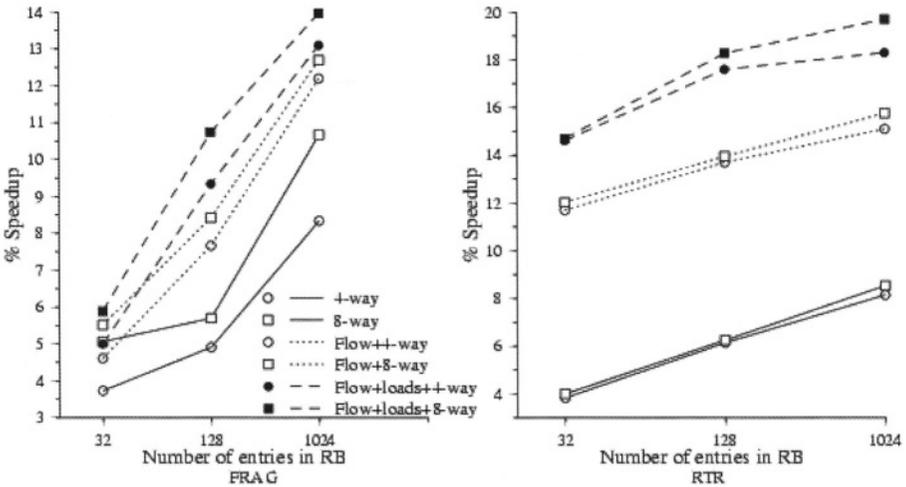


Figure 27-3. Speedup due to flow aggregation for FRAG and RTR.

most network traffic traces are anonymized (with the payload portion being removed). We used the inputs provided with the benchmarks as well as data traffic from our LAN environment to carry out experiments related to payload applications. This obviously is not very precise and does not emulate real world traffic since packets within a LAN environment are usually highly correlated resulting in optimistic results. Speedup is improved by 2% to 8% (best possible results) over the base scheme for other benchmarks.

### 3.3. Impact of reuse on resources

Figure 27-4 shows that resource demand due to flow aggregation is lower than the base scheme. Resource demand is calculated by normalizing the number of resource accesses with IR to that without IR. The figure also shows resource contention normalized to the base case.

As mentioned in section 2.2, contention may increase or decrease due to IR. Resource contention comes down drastically on using the *extended configuration* (see section 3.1). Load instruction reuse also reduces memory traffic [4] significantly (see last column of Table 27-1) which decreases the activity factor over the high capacitance buses making it a possible candidate for low power. Energy savings can also be obtained due to reduced number of executions when an instruction is reused. However, certain amount of energy is spent in accessing the RB. Also, as shown in Table 27-1, operand indexing is capable of uncovering additional IR. An extra speedup of about 2% to 4% is achieved when flow aggregation is used along with operand indexing. Since a reused instruction executes and possibly commits early, it occupies a RoB slot for a smaller amount of time. This reduces the stalls that would occur as a result of the RoB being full. Flow based reuse has the ability to further

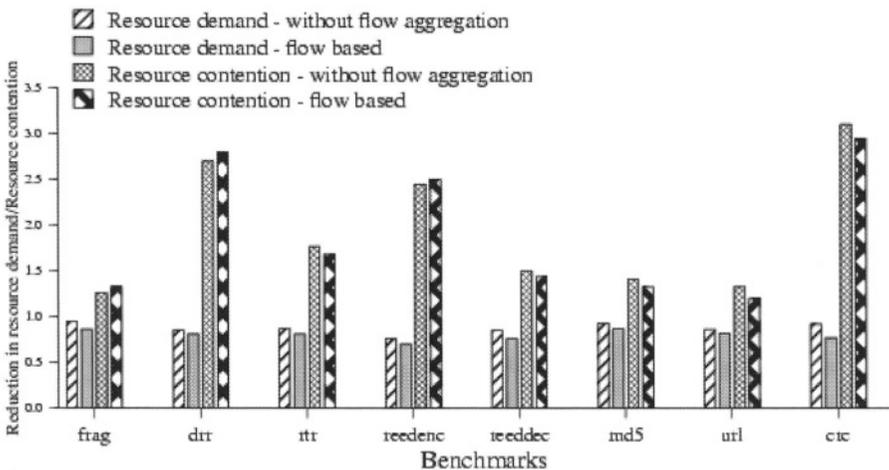


Figure 27-4. Resource demand and contention due to reuse.

improve IR thereby reducing the RoB occupancy even more. Flow based reuse reduces the occupancy of the RoB by 1.5% to 3% over the base reuse scheme.

#### 4. CONCLUSIONS

In this article we examined instruction reuse of integer ALU and load instructions in packet processing applications. To further enhance the utility of reuse by reducing interference, a flow aggregation scheme that exploits packet correlation and uses multiple RB's is proposed. The results indicate that exploiting instruction reuse with flow aggregation does significantly improve the performance of our NPU model (24% improvement in speedup in the best case). IR and speedup that can be achieved depends on the nature of the network traffic and working set sizes. A direct relationship between reuse and speedup does not exist. This means that although instruction reuse improves due to flow aggregation, speedup may not improve proportionally. This is because speedup depends on a lot of other parameters such as the criticality of the instruction being reused. Future work would be to simulate the architecture proposed, evaluate energy issues and determine which instructions really need to be present in the RB. Interference in the RB can be reduced if critical instructions are identified and placed in the RB. Further, a detailed exploration of various mapping schemes is necessary to evenly distribute related data among RB's. Finally, we believe that additional reuse can be recovered from payload processing applications if realistic (non-anonymized) traffic is used and operand indexing is exploited.

#### REFERENCES

1. A. Sodani and G. Sohi. "Dynamic Instruction Reuse." *24th Annual International Symposium on Computer Architecture*, July 1997, pp. 194–205.
2. A. Sodani and G. Sohi. "Understanding the Differences between Value Prediction and Instruction Reuse." *32nd Annual International Symposium on Microarchitecture*, December 1998, pp. 205–215.
3. A. Sodani and G. Sohi. "An Empirical Analysis of Instruction Repetition." In *Proceedings of ASPLOS-8*, 1998.
4. J. Yang and R. Gupta, "Load Redundancy Removal through Instruction Reuse." In *Proceedings of International Conference on Parallel Processing*, August 2000, pp. 61–68.
5. C. Molina, A. Gonzalez and J. Tubella. "Dynamic Removal of Redundant Computations." In *Proceedings of International Conference on Supercomputing*, June 1999.
6. F. Baker. "Requirements for IP Version 4 Routers," RFC – 1812, Network Working Group, June 1995.
7. Intel IXP1200 Network Processor – Hardware Reference Manual, Intel Corporation, December 2001.
8. D. Burger, T. M. Austin, and S. Bennett. "Evaluating Future Microprocessors: The SimpleScalar Tool Set." *Technical Report CS-TR-96-1308*, University of Wisconsin-Madison, July 1996.
9. Tilman Wolf and, Mark Franklin. "CommBench – A Telecommunications Benchmark for

- Network Processors.” *IEEE Symposium on Performance Analysis of Systems and Software*, Apr 2000, pp. 154–162.
10. Gokhan Memik, B. Mangione-Smith, and W. Hu, “NetBench: A Benchmarking Suite for Network Processors.” In *Proceedings of ICCAD*, November 2001.
  11. S. Bradner and J. McQuaid. “A Benchmarking Methodology for Network Interconnect Devices.” RFC – 2544.
  12. Stephen Melvin and Yale Patt. “Handling of Packet Dependencies: A Critical Issue for Highly Parallel Network Processors.” In *Proceedings of CASES*, 2002.

*This page intentionally left blank*

## Chapter 28

# ON-CHIP STOCHASTIC COMMUNICATION

Tudor Dumitraş and Radu Mărculescu

*Carnegie Mellon University, Pittsburgh, PA 15213, USA*

**Abstract.** As CMOS technology scales down into the deep-submicron (DSM) domain, the Systems-On-Chip (SoCs) are getting more and more complex and the costs of design and verification are rapidly increasing due to the inefficiency of traditional CAD tools. Relaxing the requirement of 100% correctness for devices and interconnects drastically reduces the costs of design but, at the same time, requires that SoCs be designed with some degree of system-level fault-tolerance. In this chapter, we introduce a new communication paradigm for SoCs, namely stochastic communication. The newly proposed scheme not only separates communication from computation, but also provides the required built-in fault-tolerance to DSM failures, is scalable and cheap to implement. For a generic tile-based architecture, we show how a ubiquitous multimedia application (an MP3 encoder) can be implemented using stochastic communication in an efficient and robust manner. More precisely, up to 70% data upsets, 80% packet drops because of buffer overflow, and severe levels of synchronization failures can be tolerated while maintaining a much lower latency than a traditional bus-based implementation of the same application. We believe that our results open up a whole new area of research with deep implications for on-chip network design of future generations of SoCs.

**Key words:** system-on-chip, network-on-chip, on-chip communication, high performance, stochastic communication

## 1. INTRODUCTION

As modern systems-on-chip (SoCs) are becoming increasingly complex, the Intellectual Property (IP)-based design is widely accepted as the main reuse paradigm. This methodology makes a clear distinction between *computation* (the tasks performed by the IPs) and *communication* (the interconnecting architecture and the communication protocols used). Currently, the traditional CAD tools and methodologies are very inefficient in dealing with a large number of IPs placed on a single chip.

A recently proposed platform for the on-chip interconnects is the network-on-chip (NoC) approach [1], where the IPs are placed on a rectangular grid of tiles (see Figure 28-1) and the communication between modules is implemented by a stack of networking protocols. However, the problem of defining such *communication protocols* for NoCs does not seem to be an easy matter, as the constraints that need to be satisfied are very tight and the important resources used in traditional data networks are not available at chip level.

Furthermore, as the complexity of designs increases and the technology

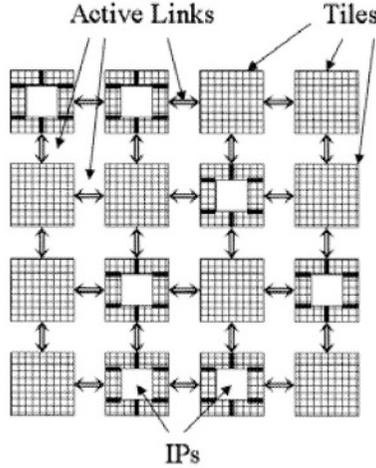


Figure 28-1. Network on chip.

scales down into the deep-submicron (DSM) domain, devices and interconnect are subject to new types of malfunctions and failures that are harder to predict and avoid with the current SoC design methodologies. These new types of failures are impossible to characterize using *deterministic* measurements so, in the near future, *probabilistic* metrics, such as average values and variance, will be needed to quantify the critical design objectives, such as performance and power [10, 11].

Relaxing the requirement of 100% correctness in operation drastically reduces the costs of design but, at the same time, requires SoCs be designed with some degree of system-level *fault-tolerance* [12]. Distributed computing has dealt with fault-tolerance for a long time; unfortunately most of those algorithms are unlikely to be useful, because they need significant resources which are not easily available on-chip. Furthermore, classic networking protocols, as the Internet Protocol or the ATM Layer [13], need *retransmissions* in order to deal with incorrect data, thus significantly increasing the latency. Generally, simple deterministic algorithms do not cope very well with random failures [14]. On the other hand, the cost of implementing full-blown *adaptive routing* [15] for NoCs is prohibitive because of the need of very large buffers, lookup tables and complex shortest-path algorithms.

To address these unsolved problems, the present chapter introduces a new communication paradigm called *on-chip stochastic communication*. This is similar, but *not* identical, to the proliferation of an epidemic in a large population [16]. This analogy explains intuitively the high performance and robustness of this protocol in the presence of failures. More precisely, we place ourselves in a NoC context (see Figure 28-1), where the IPs communicate using a probabilistic broadcast scheme, known as *stochastic communication* [4]. If a tile has a packet to send, it will forward the packet to a randomly

chosen subset of the tiles in its neighborhood. This way, the packets are *diffused* from tile to tile to the entire NoC. Every IP then selects from the set of received messages only those that have its own ID as the destination.

This simple algorithm achieves many of the desired features of the future NoCs. As shown below, the algorithm provides:

- *separation between computation and communication*, as the communication scheme is implemented in the network logic and is transparent to the IPs;
- *fault-tolerance* since a message can still reach its destination despite severe levels of DSM failures on the chip;
- *extremely low latency* since this communication scheme does not require retransmissions in order to deal with incorrect data;
- *low production costs* because the fault-tolerant nature of the algorithm eliminates the need for detailed testing/verification;
- *design flexibility* since it provides a mechanism to tune the tradeoff between performance and energy consumption.

This chapter is organized as follows: at first, we review the previous work relevant to this field. In Section 3, we develop a novel failure model for NoCs and, in Section 4, we describe a communication scheme designed to work in such an environment. In Section 5, we show experimental results that clearly indicate the great potential of this approach. We conclude by summarizing our main contribution.

## 2. PREVIOUS WORK

One of the important mathematical challenges of this century has been developing analytical models for the proliferation of diseases. The earliest research in this direction [16] showed that, under certain conditions, epidemics spread *exponentially fast*. Much later, [3] proposed a probabilistic broadcast algorithm, the *randomized gossip*, for the lazy update of data objects in a database replicated at many sites and proved that this scheme behaves very much like the spreading of an epidemic. Several networking protocols are based on the same principles, as the gossip protocols proved to be very *scalable* and able to maintain *steady throughput* [7, 13]. Recently, these types of algorithms were applied to networks of sensors [8]. Their ability to limit communication to local regions and support lightweight protocols, while still accomplishing their task, is appealing to applications where power, complexity and size constraints are critical.

The problem of designing NoCs for fault-tolerance has been addressed only recently. From a design perspective, in order to deal with node failures, Valtonen et al. [9] proposed an architecture based on autonomous, error-tolerant cells, which can be tested at any time for errors and, if needed, disconnected from the network by the other cells. A more detailed failure

model is described in [2], including data upsets and omission failures. However, a communication protocol which tolerates these types of errors has yet to be specified. Furthermore, the problem of synchronization failures (see Section 3), as well as their impact on NoC communication, have not been addressed so far. Our contribution tries to fill this important gap in the research area of on-chip networks.

### 3. FAILURE MODEL

Several failure models have been identified in the traditional networking literature [19]. *Crash failures* are *permanent* faults which occur when a tile halts prematurely or a link disconnects, after having behaved correctly until the failure. *Transient faults* can be either *omission failures*, when links lose some messages and tiles intermittently omit to send or receive, or *arbitrary failures* (also called Byzantine or malicious), when links and tiles deviate arbitrarily from their specification, corrupting or even generating spurious messages.

However, some of these models are not very relevant to the on-chip networks. In the DSM realm faults that are most likely to occur are fluctuations in the dopant distributions, high fields causing band-to-band tunneling and mobility degradation, or important leakage currents. For an efficient system-level communication synthesis, we need failure models that are easy to manipulate and that reflect the behavior of the circuits.

Because of crosstalk and electromagnetic interference, the most common type of failures in DSM circuits will be data transmission errors (also called upsets) [12]. Simply stated, if the noise in the interconnect causes a message to be scrambled, a *data upset* error will occur; these errors are subsequently characterized by a probability  $p_{upset}$ . Another common situation is when a message is lost because of the buffer overflow; this is modeled by the probability  $p_{overflow}$ . Furthermore, as modern circuits span multiple clock domains (as in GALS architectures [18]) and can function at different voltages and frequencies (as in a “voltage and frequency island”-based architecture advocated by IBM [17]) *synchronization errors* are very hard to avoid. In our experiments, we have adopted a tile-based architecture in which every tile has its own clock domain; synchronization errors are normally distributed with a standard deviation  $\sigma_{synchron}$ .

Summarizing, our fault model depends on the following parameters:

- pupset: probability a packet is scrambled because of a *data upset*;
- poverflow: probability a packet is dropped because of *buffer overflow*;
- $\sigma_{synchron}$ : standard deviation error of the duration of a round ( $T_R$ ) (see Section 4.3), which indicates the magnitude of *synchronization errors*.

We believe that establishing this stochastic failure model is a decisive step towards solving the fault-tolerant communication problem, as it emphasizes

the nondeterministic nature of DSM faults. This suggests that a stochastic approach (described next) is best suited to deal with such realities.

#### 4. STOCHASTIC COMMUNICATION

The technique we call *on-chip stochastic communication* implements the NoC communication using a *probabilistic broadcast* algorithm [4]. The behavior of such an algorithm is similar to the spreading of a rumor within a large group of friends. Assume that, initially, only one person in the group knows the rumor. Upon learning the rumor, this person (the initiator) passes it to someone (confidant) chosen at random. At the next round, both the initiator and the confidant, if there is one, select independently of each other someone else to pass the rumor to. The process continues in the same fashion; namely, everyone informed after  $t$  rounds passes the rumor at the  $(t + 1)$ th round to someone selected at random independently of all other past and present selections. Such a scheme is called *gossip algorithm* and is known to model the spreading of an epidemic in biology [16].

Let  $I(t)$  be the number of people who have become aware of the rumor after  $t$  rounds ( $I(0) = 1$ ) and let  $S_n = \min\{t: I(t) = n\}$  be the number of rounds until  $n$  people are informed. We want to estimate  $S_n$ , in order to evaluate how fast the rumor is spread. A fundamental result states that:

$$S_n = \log_2 n + \ln n + O(1) \quad \text{as } n \rightarrow \infty \tag{1}$$

with probability 1 [6]. Therefore, after  $O(\log_2 n)$  rounds ( $n$  represents the number of nodes) all the nodes have received the message *with high probability (w.h.p.)* [4]. For instance, in Figure 28-2, in less than 20 rounds as many as 1000 nodes can be reached. Conversely, after  $t$  rounds the number of people that have become aware of the rumor is an exponential function of  $t$ , so this algorithm spreads rumors *exponentially fast*.

The analogy we make for the case of NoCs is that tiles are the “gossiping friends” and packets transmitted between them are the “rumors” (see Figure 28-3). Since any friend in the original setup is able to gossip with anyone else in the group, the above analysis can be applied directly only to the case

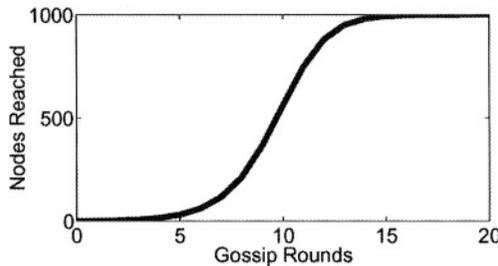


Figure 28-2. Message spreading in a 1000 node fully connected network.

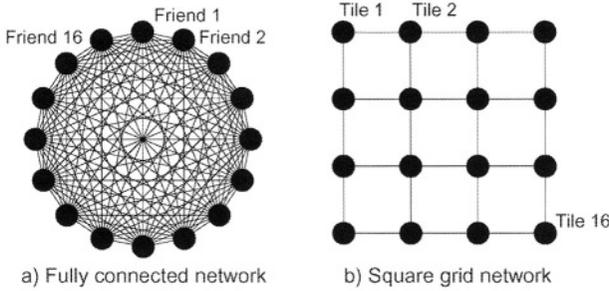


Figure 28-3. Different topologies for a 16 node network.

of a fully connected network, like the one in Figure 28-3a. However, because of its high wiring demands, such an architecture is *not* a reasonable solution for SoCs. Therefore, for NoCs, we consider a grid-based topology (Figure 28-3b), since this is much easier and cheaper to implement on silicon. Although the theoretical analysis in this case is an open research question, our experimental results show that the messages can be spread explosively fast among the tiles of the NoC for this topology as well. To the best of our knowledge, this represents the first evidence that gossip protocols can be applied to SoC communication as well.

In the following paragraphs, we will describe an algorithm for on-chip stochastic communication and show how it can be used for a real-time multimedia application (an MP3 encoder).

**4.1. Example: NoC Producer–Consumer application**

In Figure 28-4 we give the example of a simple Producer–Consumer application. On a NoC with 16 tiles the Producer is placed on tile 6 and the Consumer on tile 12. Suppose the Producer needs to send a message to the Consumer. Initially the Producer sends the message to a randomly chosen subset of its neighbors (ex. Tiles 2 and 7 in Figure 28-4a). At the second gossip round, tiles 6, 2, 7 (the Producer and the tiles that have received the message)

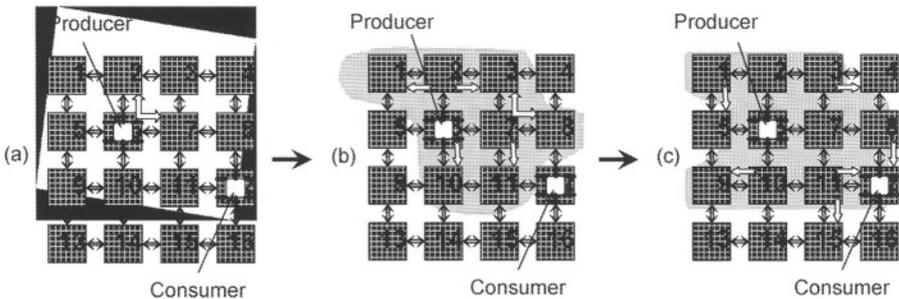


Figure 28-4. Producer–Consumer application for a stochastically communicating NoC.

during the first round) forward it in the same manner. After this round, eight tiles (6, 2, 7, 1, 3, 8, 10, and 11) become aware of the message and are ready to send it to the rest of the network. At the third gossip round, the Consumer finally receives the packet from tiles 8 and 11. Note that:

- the Producer needs *not* know the location of the Consumer, the message will arrive at the destination *w.h.p.*;
- the message reaches the Consumer *before* the full broadcast is completed; that is, by the time when the Consumer gets the message, tiles 14 and 16 have not yet received the message.

The most appealing feature of this algorithm is its excellent performance in the presence of failures. For instance, suppose the packet transmitted by tile 8 is affected by a data upset. The Consumer will discard it, as it receives from tile 11 another copy of the same packet anyway. The presence of such an upset is detected by implementing on each tile an error detection/multiple transmission scheme. The key observation is that, at chip level, bandwidth is *less* expensive than in traditional macro-networks. This is because of existing high-speed buses and interconnection fabrics which can be used for the implementation of a NoC. Therefore, we can afford more packet transmissions than in previous protocols, in order to simplify the communication scheme and guarantee low latencies.

## 4.2. The algorithm

The mechanism presented in the above example assumes that tiles can detect when data transmissions are affected by upsets. This is achieved by protecting packets with a cyclic redundancy code (CRC) [13]. If an error is discovered, then the packet will be simply discarded. Because a packet is retransmitted many times in the network, the receiver does *not* need to ask for retransmission, as it will receive the packet again anyway. This is the reason why stochastic communication can sustain low latencies even under severe levels of failures. Furthermore, CRC encoders and decoders are easy to implement in hardware, as they only require one shift register [13]. We also note that, since a message might reach its destination before the broadcast is completed, the spreading could be terminated even earlier in order to reduce the number of messages transmitted in the network. This is important because this number is directly connected to the bandwidth used and the energy dissipated (see Section 4.4). To do this we assign a *time to live* (TTL) to every message upon creation and decrement it at every hop until it reaches 0; then the message can be garbage-collected. This will lead to important bandwidth and energy savings, as shown in Section 5.

Our algorithm is presented in Figure 28-5 (with standard set theory notations). The tasks executed by all the nodes are concurrent. A tile forwards the packet that is available for sending to all four output ports, and then a random decision is made (with probability  $p$ ) whether or not to transmit the

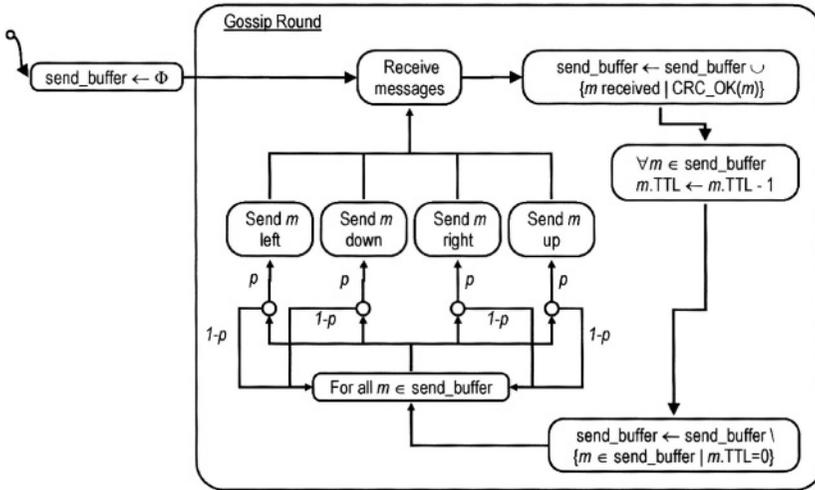


Figure 28-5. Stochastic communication – the algorithm.

message to the next tile. In Section 5 we will show how this probability can be used to tune the tradeoff between performance and energy consumption.

### 4.3. Performance metrics for stochastic communication

A *broadcast round* is the time interval in which a tile has to finish sending all its messages to the next hops; this will usually take several clock cycles. The optimal duration of a round ( $T_R$ ) can be determined using Equation 2, where  $f$  is the maximum frequency of any link,  $N_{packets/round}$  is the average number of packets that a link sends during one round (which is application-dependent), and  $S$  is the average packet size.

$$T_R = \frac{N_{packets/round} S}{f} \tag{2}$$

As we show in Section 5, the fast dissemination of rumors in this algorithm makes it possible to achieve very low latencies. This protocol spreads the traffic onto all the links in the network, reducing the chances that packets are delayed because of congestion. This is especially important in multimedia applications, where a sustainable constant bit rate is highly desirable. Furthermore, the fact that we don't store or compute the shortest paths (as is the case of dynamic routing) makes this algorithm *computationally light-weight*, simpler and easier to customize for every application and interconnection network.

The following parameters are relevant to our analysis:

- the number of broadcast rounds needed is a direct measure of the inter-IP communication latency;

- the total number of packets sent in the network shows the bandwidth required by the algorithm and can be controlled by varying the message TTL;
- the fault-tolerance evaluates the algorithm’s resilience to abnormal functioning conditions in the network;
- the energy consumption, computed with Equation 3 (see next section).

#### 4.4. Energy metrics for stochastic communication

In estimating this algorithm’s energy consumption, we take into consideration the total number of packets sent in the NoC, since these transmissions account for the switching activity at network level. This is expressed by Equation 3, where  $N_{packets}$  is the total number of messages generated in the network,  $S$  is the average size of one packet (in bits) and  $E_{bit}$  is the energy consumed per bit:

$$E_{total} = E_{computation} + E_{communication} = E_{computation} + N_{packets}SE_{bit} \quad (3)$$

$N_{packets}$  can be estimated by simulation,  $S$  is application-dependent, and  $E_{bit}$  is a parameter from the technology library. As shown in Equation 3, the total energy consumed by the chip will be influenced by the activity in the computational cores as well ( $E_{computation}$ ). Since we are trying to analyze here the performance and properties of the *communication scheme*, estimating the energy required by the computation is not relevant to this discussion. This can be added, however, to our estimations from Section 5 to get the combined energy.

## 5. EXPERIMENTAL RESULTS

Stochastic communication can have a wide applicability, ranging from parallel SAT solvers and multimedia applications to periodic data acquisition from non-critical sensors. A thorough set of experimental results concerning the performance of this communication scheme has been presented in [4]. In the present chapter we demonstrate how our technique works with a complex multimedia application (an MP3 encoder). We simulate this application in a stochastically communicating NoC environment and in the following sections we discuss our results.

We have developed a simulator using the Parallel Virtual Machine (PVM)<sup>2</sup> and implemented our algorithm, as described in Section 4. In our simulations, we assume the failure model introduced in Section 3, where data upsets and buffer overflows uniformly distributed in the network with probabilities  $p_{upset}$  and  $p_{overflow}$  respectively. The clocks are *not* synchronized and the duration of each round is normally distributed around the optimal  $T_R$  (see Equation 2), with standard deviation  $\sigma_{synchron}$ . As realistic data about failure patterns in regular SoCs are currently unavailable, we *exhaustively* explore here the parameter

space of our failure model. Another important parameter we vary is  $p$ , the probability that a packet is forwarded over a link (see Section 4). For example, if we set the forwarding probability to 1, then we have a completely deterministic algorithm which floods the network with messages (every tile always sends messages to all its four neighbors). This algorithm is optimal with respect to latency, since the number of intermediate hops between source and destination is always equal to the Manhattan distance, but extremely inefficient with respect to the bandwidth used and the energy consumed. Stochastic communication allows us to tune the tradeoff between energy and performance by varying the probability of transmission  $p$  between 0 and 1.

### 5.1. Target application: An MP3 encoder

MP3 encoders and decoders are ubiquitous in modern embedded systems, from PDAs and cell phones to digital sound stations, because of the excellent compression rates and their streaming capabilities. We have developed a parallel version of the LAME MP3 encoder [5] (shown in Figure 28-6) and introduced the code in our stochastically communicating simulator. The following results are based on this experimental setup.

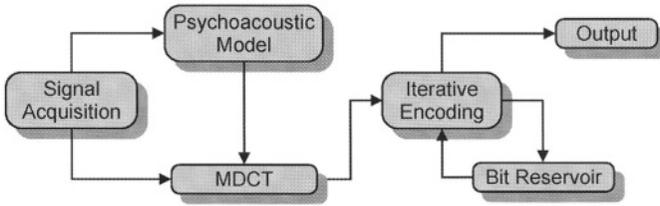
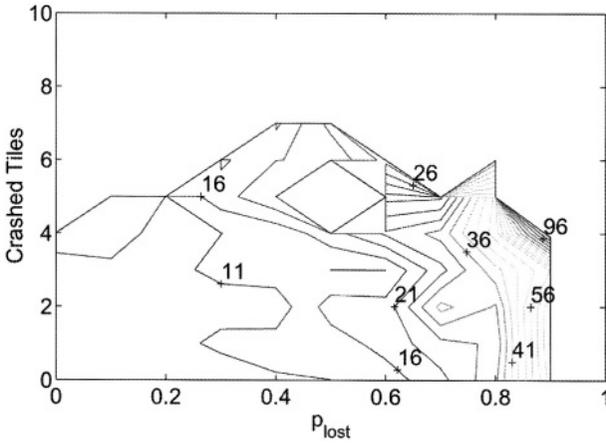


Figure 28-6. The modules of the MP3 encoder.

#### 5.1.1. Latency

The latency is measured as the number of rounds MP3 needs to finish encoding and it is influenced by several parameters: the value of  $p$  (probability of forwarding a message over a link), the levels of data upsets in the network, the buffer overflows and the synchronization errors. Figure 28-7 shows how latency varies with the probability of retransmission  $p$  and with the probability of upsets  $p_{upset}$ . As expected, the lowest latency is when  $p = 1$  (the messages are always forwarded over a link) and  $p_{upset} = 0$  (when there are no upsets in the network), and increases when  $p \rightarrow 0$  and  $p_{upset} \rightarrow 1$  to the point where the encoding of the MP3 cannot finish because the packets fail to reach their destination too often. Note that  $p_{upset}$  is dependent on the technology used, while  $p$  is a parameter that can be used directly by the designer in order to tune the behavior of the algorithm.

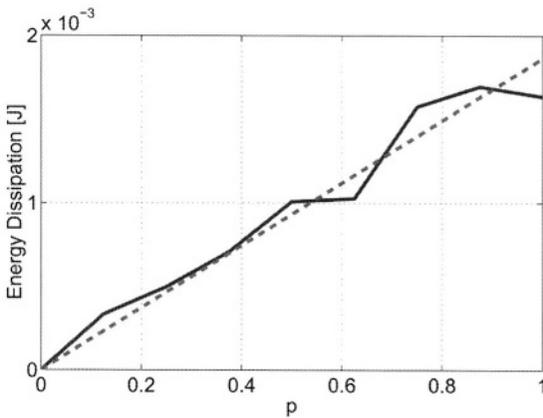


Colour picture

Figure 28-7. Latency.

5.1.2. Energy dissipation

We have estimated the energy dissipation of our algorithms, using Equation 3. We can see from Figure 28-8 that the energy dissipation increases almost linearly with the probability of resending  $p$ . This is because the energy is proportional to the total number of packets transmitted in the network and this number is controlled by  $p$ . As we can see from Figure 28-7, high values of  $p$  mean low latency, but they also mean high energy dissipation. This shows the importance of this parameter, which can be used to tune the tradeoff between performance and energy dissipation, making the stochastic communication flexible enough to fit the needs of a wide range of applications.



Colour picture

Figure 28-8. Energy dissipation.

5.1.3. Fault-tolerance

Different types of failures have different effects on the performance of stochastic communication. The levels of buffer overflow do not seem to have a big impact on latency (see left part of Figure 28-9). However the encoding will not be able to complete if these levels are too high (> 80%, as in point A in Figure 28-9), because one or several packets are lost and none of the tiles has a copies of them. On the other hand, data upsets seem to have little influence on the chances to finish encoding. However, upsets do have an impact on the latency, especially if  $p_{upset} > 0.7$  (see Figure 28-7).

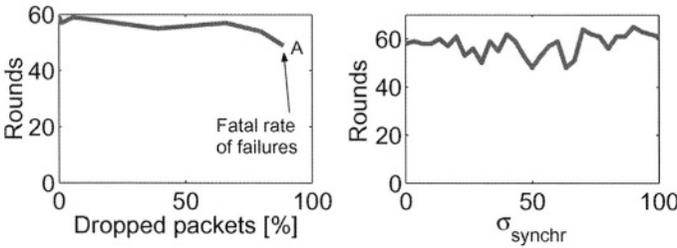


Figure 28-9. Rounds needed to finish encoding.

The influence of synchronization errors on the output bit rate and latency is shown in Figure 28-10 (together with the jitter) and the right part of Figure 28-9. Note that even very important synchronization error levels do not have a big impact on the bit rate, the jitter or the latency. This proves that our method tolerates extremely well very high levels of synchronization errors.

These results show that stochastic communication has a very good behavior in time with respect to latency and energy dissipation. In the worst case (very unlikely for typical levels of failures in NoCs), the protocol will not deliver

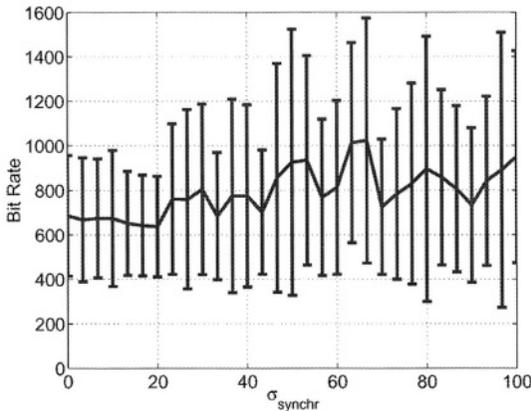


Figure 28-10. Impact of synchronization errors on bit rate.

the packet to the destination; therefore this communication paradigm may be better suited for applications which tolerate small levels of information loss, such as our MP3 encoder. In general, real-time streaming multimedia applications have requirements that match very well the behavior of our new communication paradigm, as they can deal with small information losses as long as the bit rate is steady. The results show that our MP3 encoder tolerates very high levels of failures with a graceful quality degradation. If, however, the application requires strong reliability guarantees, these can be implemented by a higher level protocol built on top of the stochastic communication.

## 6. CONCLUSION

In this chapter, we emphasized the need for on-chip fault-tolerance and proposed a new communication paradigm based on stochastic communication. This approach takes advantage of the large bandwidth that is available on chip to provide the needed system-level tolerance to synchronization errors, data upsets and buffer overflows. At the same time, this method offers a low-latency, low cost and easy to customize solution for on-chip communication. We believe that our results suggest the big potential of this approach and they strongly indicate that further research in this area would lead to vital improvements of the SoC interconnect design.

## ACKNOWLEDGEMENTS

The authors would like to express their appreciation to Sam Kerner, who helped set up some of the simulators used in the present work. We would also like to thank Mr. Jingcao Hu for his insightful remarks on the results of our experiments.

## NOTES

<sup>1</sup> The term *with high probability* means with probability at least  $1 - O(n^{-\alpha})$  for some  $\alpha > 0$ .

<sup>2</sup> PVM is a programming platform which simulates a message passing multi processor architecture [20].

## REFERENCES

1. W. Dally and W. Towles. "Route Packets, Not Wires: On-Chip Interconnection Networks." In *Proceedings of the 38th DAC*, June 2001.
2. T. Dumitraş, S. Kerner, and R. Mărculescu. "Towards On-Chip Fault-Tolerant Communication." In *Proceedings of the ASP-DAC*, January 2003.

3. A. Demers et al. "Epidemic algorithms for replicated database maintenance." In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, August 1987.
4. T. Dumitraş and R. Mărculescu. "On-Chip Stochastic Communication." In *Proceedings of DATE*, March 2003
5. The LAME project. <http://www.mp3dev.org/mp3/>.
6. B. Pittel. "On Spreading a Rumor." *SIAM Journal of Appl. Math.*, 1987.
7. K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. "Bimodal Multicast." *ACM Transactions on Computer Systems*, Vol. 17, No. (2), pp. 41–88, 1999.
8. D. Estrin, R. Govindan, J. Heidemann, and S. Kumar. "Next century challenges: Scalable coordination in sensor Networks." In *Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking*. ACM, August 1999.
9. T. Valtonen et al. "Interconnection of Autonomous Error-Tolerant Cells." In *Proceedings of ISCAS*, 2002.
10. V. Maly. "IC Design in High-Cost Nanometer Technologies era". In *Proceedings of the 38th DAC*, June 2001.
11. L. Benini and G. De Micheli. "Networks on Chips: A new Paradigm for Systems on Chip Design." In *Proceedings of the 39th DAC*, June 2002.
12. Semiconductor Association. "The International Technology Roadmap for Semiconductors (ITRS)", 2001.
13. A. Leon-Garcia and I. Widjaja. *Communication Networks*. McGraw-Hill, 2000.
14. F. T. Leighton et al. "On the Fault Tolerance of Some Popular Bounded-Degree Networks." In *IEEE Symp. on Foundations of Comp. Sci.*, 1992.
15. L. M. Ni and P. K. McKinley. "A Survey of Wormhole Routing Techniques in Direct Networks." *IEEE Computer*, 1993.
16. N. Bailey. *The Mathematical Theory of Infectious Diseases*. Charles Griffin and Company, London, 2nd edition, 1975.
17. D. E. Lackey et al. "Managing Power and Performance for System-on-Chip Designs using Voltage Islands." In *Proceedings of the ICCAD*, November 2002.
18. D. M. Chapiro. *Globally Asynchronous Locally Synchronous Systems*. Ph.D. thesis, Stanford University, 1984.
19. V. Hadzilacos and S. Toueg. "A Modular Approach to Fault-Tolerant Broadcasts and Related Problems." *Technical Report TR941425*, 1994.
20. PVM: Parallel Virtual Machine. [http://www.csm.ornl.gov/pvm/pvm\\_home.html](http://www.csm.ornl.gov/pvm/pvm_home.html).

# HARDWARE/SOFTWARE TECHNIQUES FOR IMPROVING CACHE PERFORMANCE IN EMBEDDED SYSTEMS

Gokhan Memik, Mahmut T. Kandemir, Alok Choudhary and Ismail Kadayif  
<sup>1</sup> *Department of Electrical Engineering, UCLA;* <sup>2</sup> *Department of Computer Science and Engineering, Penn State;* <sup>3</sup> *Department of Electrical and Computer Engineering, Northwestern University;* <sup>4</sup> *Department of Computer Science and Engineering, Penn State*

**Abstract.** The widening gap between processor and memory speeds renders data locality optimization a very important issue in data-intensive embedded applications. Throughout the years hardware designers and compiler writers focused on optimizing data cache locality using intelligent cache management mechanisms and program-level transformations, respectively. Until now, there has not been significant research investigating the interaction between these optimizations. In this work, we investigate this interaction and propose a selective hardware/compiler strategy to optimize cache locality for integer, numerical (array-intensive), and mixed codes. In our framework, the role of the compiler is to identify program regions that can be optimized at compile time using loop and data transformations and to mark (at compile-time) the unoptimizable regions with special instructions that activate/deactivate a hardware optimization mechanism selectively at run-time. Our results show that our technique can improve program performance by as much as 60% with respect to the base configuration and 17% with respect to non-selective hardware/compiler approach.

**Key words:** cache optimizations, cache bypassing, data layout transformations

## 1. INTRODUCTION AND MOTIVATION

To improve performance of data caches, several hardware and software techniques have been proposed. Hardware approaches try to anticipate future accesses by the processor and try to keep the data close to the processor. Software techniques such as compiler optimizations [6] attempt to reorder data access patterns (e.g., using loop transformations such as tiling) so that data reuse is maximized to enhance locality. Each approach has its strengths and works well for the patterns it is designed for. So far, each of these approaches has primarily existed independently of one another. For example, a compiler-based loop restructuring scheme may not really consider the existence of a victim cache or its interaction with the transformations performed. Similarly, a locality-enhancing hardware technique does not normally consider what software optimizations have already been incorporated into the code. Note that the hardware techniques see the addresses generated by the processor,

which already takes the impact of the software restructuring into account. In addition, there is a promising aspect of combining the hardware and software approaches. Usually compilers have a global view of the program, which is hard to obtain at run-time. If the information about this global view can be conveyed to the hardware, the performance of the system can be increased significantly. This is particularly true if hardware can integrate this information with the runtime information it gathers during execution.

But, can we have the best of both worlds? Can we combine hardware and software techniques in a logical and selective manner so that we can obtain even better performance than either applying only one or applying each independently? Are there simple and cost-effective ways to combine them? Can one scheme interfere with another if applied independently; that is, does it result in performance degradation as compared to using either one scheme only or no scheme at all?

To answer these questions, we use hardware and software locality optimization techniques in concert and study the interaction between these optimizations. We propose an integrated scheme, which selectively applies one of the optimizations and turns the other off. Our goal is to combine existing hardware and software schemes intelligently and take full advantage of both the approaches. To achieve this goal, we propose a region detection algorithm, which is based on the compiler analysis and that determines which regions of the programs are suitable for hardware optimization scheme and subsequently, which regions are suitable for software scheme. Based on this analysis, our approach turns the hardware optimizations on and off. In the following, we first describe some of the current hardware and software locality optimization techniques briefly, give an overview of our integrated strategy, and present the organization of this chapter.

### **1.1. Hardware techniques**

Hardware solutions typically involve several levels of memory hierarchy and further enhancements on each level. Research groups have proposed smart cache control mechanisms and novel architectures that can detect program access patterns at run-time and can fine-tune some cache policies so that the overall cache utilization and data locality are maximized. Among the techniques proposed are victim caches [10], column-associative caches [1], hardware prefetching mechanisms, cache bypassing using memory address table (MAT) [8, 9], dual/split caches [7], and multi-port caches.

### **1.2. Software techniques**

In the software area, there is considerable work on compiler-directed data locality optimizations. In particular, loop restructuring techniques are widely used in optimizing compilers [6]. Within this context, transformation techniques such as loop interchange [13], iteration space tiling [13], and loop

unrolling have already found their ways into commercial compilers. More recently, alternative compiler optimization methods, called data transformations, which change the memory layout of data structures, have been introduced [12]. Most of the compiler-directed approaches have a common limitation: they are effective mostly for applications whose data access patterns are analyzable at compile time, for example, array-intensive codes with regular access patterns and regular strides.

### **1.3. Proposed hardware/software approach**

Many large applications, however, in general exhibit a mix of regular and irregular patterns. While software optimizations are generally oriented toward eliminating capacity misses coming from the regular portions of the codes, hardware optimizations can, if successful, reduce the number of conflict misses significantly.

These observations suggest that a combined hardware/compiler approach to optimizing data locality may yield better results than a pure hardware-based or a pure software-based approach, in particular for codes whose access patterns change dynamically during execution. In this study, we investigate this possibility.

We believe that the proposed approach fills an important gap in data locality optimization arena and demonstrates how two inherently different approaches can be reconciled and made to work together. Although not evaluated in this work, selective use of hardware optimization mechanism may also lead to large savings in energy consumption (as compared to the case where the hardware mechanism is on all the time).

### **1.4. Organization**

The rest of this chapter is organized as follows. Section 2 explains our approach for turning the hardware on/off. Section 3 explains in detail the hardware and software optimizations used in this study. In Section 4, we explain the benchmarks used in our simulations. Section 5 reports performance results obtained using a simulator. Finally, in Section 6, we present our conclusions.

## **2. PROGRAM ANALYSIS**

### **2.1. Overview**

Our approach combines both compiler and hardware techniques in a single framework. The compiler-related part of the approach is depicted in Figure 29-1. It starts with a region detection algorithm (Section 2.2) that divides an input program into uniform regions. This algorithm marks each region with

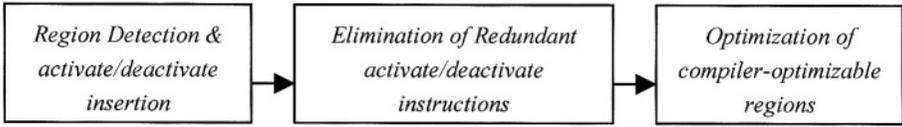


Figure 29-1. Overview of the compiler-related part of our framework.

special *activate/deactivate* (ON/OFF) instructions that activate/deactivate a hardware optimization scheme *selectively* at run-time. Then, we use an algorithm, which detects and eliminates redundant activate/deactivate instructions. Subsequently, the software optimizable regions are handled by the compiler-based locality optimization techniques. The remaining regions, on the other hand, are handled by the hardware optimization scheme at run-time. These steps are detailed in the following sections.

## 2.2. Region detection

In this section, we present a compiler algorithm that divides a program into disjoint regions, preparing them for subsequent analysis. The idea is to detect *uniform regions*, where ‘uniform’ in this context means that the memory accesses in a given region can be classified as either regular (i.e., compile-time analyzable/optimizable), as in array-intensive embedded image/video codes or irregular (i.e., not analyzable/optimizable at compile time), as in non-numerical codes (and also in numerical codes where pattern cannot be statically determined, e.g., subscripted array references). Our algorithm works its way through loops in the nests from the innermost to the outermost, determining whether a given region should be optimized by hardware or compiler. This processing order is important as the innermost loops are in general the dominant factor in deciding the type of the access patterns exhibited by the nests.

The smallest region in our framework is a single loop. The idea behind region detection can be best illustrated using an example. Consider Figure 29-2(a). This figure shows a schematic representation of a nested-loop hierarchy in which the head of each loop is marked with its depth (level) number, where the outermost loop has a depth of 1 and the innermost loop has a depth of 4. It is clear that the outermost loop is imperfectly-nested as it contains three inner nests at level 2. Figure 29-2(b) illustrates how our approach proceeds.

We start with the innermost loops and work our way out to the outermost loops. First, we analyze the loop at level 4 (as it is the innermost), and considering the references it contains, we decide whether a hardware approach or a compiler approach is more suitable (Step 1). Assume for now, without loss of generality, that a hardware approach is more suitable for this loop (how to decide this will be explained later). After placing this information on its

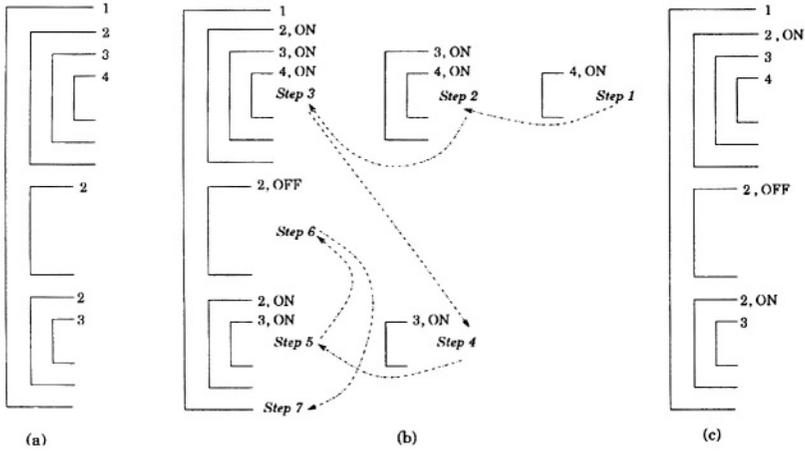


Figure 29-2. An example that illustrates the region detection algorithm. (a) Schematic representation of a nested-loop hierarchy. (b) Steps taken by our approach to insert ON/OFF instructions. (c) The resulting structure after the elimination of redundant ON/OFF instructions.

loop header (in the form of an activate (ON) instruction), we move (Step 2) to the loop at level 3, which encloses the loop at level 4. Since this loop (at level 3) contains only the loop at level 4, we propagate the preferred optimization method of the loop at level 4 to this loop. That is, if there are memory references, inside the loop at level 3 but outside the loop at level 4, they will also be optimized using hardware. In a similar vein, we also decide to optimize the enclosing loop at level 2 using hardware (Step 3). Subsequently, we move to loop at level 1. Since this loop contains the loops other than the last one being analyzed, we do not decide at this point whether a hardware or compiler approach should be preferred for this loop at level 1.

We now proceed with the loop at level 3 in the bottom (Step 4). Suppose that this and its enclosing loop at level 2 (Step 5) are also to be optimized using a hardware approach. We move to the loop at level 2 in the middle (Step 6), and assume that after analyzing its access pattern we decide that it can be optimized by compiler. We mark its loop header with this information (using a deactivate (OFF) instruction). The leftmost part of Figure 29-2(b) shows the situation after all ON/OFF instructions have been placed.

Since we have now processed all the enclosed loops, we can analyze the loop at level 1 (Step 7). Since this loop contains loops with different preferred optimization strategies (hardware and compiler), we cannot select a unique optimization strategy for it. Instead, we need to switch from one technique to another as we process its constituent loops: suppose that initially we start with a compiler approach (i.e., assuming that as if the entire program is to be optimized in software), when we encounter with loop at level 2 at the top position, we activate (using a special instruction) the hardware locality optimization mechanism (explained later on). When we reach the middle loop

at level 2, we deactivate the said mechanism, only to reactivate it just above the loop at level 2 at the bottom of the figure. This step corresponds to the elimination of redundant activate/deactivate instructions in Figure 29-2. We do not present the details and the formal algorithm due to lack of space. In this way, our algorithm partitions the program into regions, each with its own preferred method of locality optimization, and each is delimited by activate/deactivate instructions which will activate/deactivate a hardware data locality optimization mechanism at run-time. The resulting code structure for our example is depicted in Figure 29-2(c).

The regions that are to be optimized by compiler are transformed statically at compile-time using a locality optimization scheme (Section 3.2). The remaining regions are left unmodified as their locality behavior will be improved by the hardware during run-time (Section 3.1). Later in this chapter, we discuss why it is not a good idea to keep the hardware mechanism on for the entire duration of the program (i.e., irrespective of whether the compiler optimization is used or not).

An important question now is what happens to the portions of the code that reside within a large loop but are sandwiched between two nested-loops with different optimization schemes (hardware/compiler)? For example, in Figure 29-2(a), if there are statements between the second and third loops at level 2 then we need to decide how to optimize them. Currently, we assign an optimization method to them considering their references. In a sense they are treated as if they are within an imaginary loop that iterates only once. If they are amenable to compiler-approach, we optimize them statically at compile-time, otherwise we let the hardware deal with them at run-time.

### 2.3. Selecting an optimization method for a loop

We select an optimization method (hardware or compiler) for a given loop by considering the references it contains. We divide the references in the loop nest into two disjoint groups, analyzable (optimizable) references and non-analyzable (not optimizable) references. If the ratio of the number of analyzable references inside the loop and the total number of references inside the loop exceeds a pre-defined threshold value, we optimize the loop in question using the compiler approach; otherwise, we use the hardware approach.

Suppose  $i$ ,  $j$ , and  $k$  are loop indices or induction variables for a given (possibly nested) loop. The analyzable references are the ones that fall into one of the following categories:

- scalar references, e.g.,  $A$
- affine array references, e.g.,  $B[i]$ ,  $C[i+j][k-1]$

Examples of non-analyzable references, on the other hand, are as follows:

- non-affine array references, e.g.,  $D[i^2][j]$ ,  $iE[i/j]$

- indexed (subscripted) array references, e.g.,  $G[IP[j]+2]$
- pointer references, e.g.,  $*H[i]$ ,  $*I$
- struct constructs, e.g.,  $J.field$ ,  $K \rightarrow field$

Our approach checks at compile-time the references in the loop and calculates the ratio mentioned above, and decides whether compiler should attempt to optimize the loop. After an optimization strategy (hardware or compiler) for the innermost loop in a given nested-loop hierarchy is determined, the rest of the approach proceeds as explained in the previous subsection (i.e., it propagates this selection to the outer loops).

### 3. OPTIMIZATION TECHNIQUES

In this section, we explain the hardware and software optimizations used in our simulations.

#### 3.1. Hardware optimizations

The approach to locality optimization by hardware concentrates on reducing conflict misses and their effects. Data accesses together with low set-associativity in caches may exhibit substantial conflict misses and performance degradation. To eliminate the costly conflict misses, we use the strategy proposed by Johnson and Hwu [8, 9]. This is a selective variable size caching strategy based on the characteristics of accesses to memory locations. The principle idea behind the technique presented in [8] is to avoid such misses by not caching the memory regions with low access frequency, thereby keeping the highly accessed regions of the memory in cache. And in the case where spatial locality is expected for the fetched data, fetch larger size blocks.

The scheme has two crucial parts – (1) a mechanism to track the access frequency of different memory locations and detect spatial locality, and (2) a decision logic to assist the cache controller to make caching decisions based on the access frequencies and spatial locality detection. To track the frequency of access to memory locations, the memory is divided into groups of adjacent cache blocks, called macro-blocks [8]. A Memory Access Table (MAT) captures the access frequencies of the macro-blocks. An additional table called Spatial Locality Detection Table (SLDT) is used to detect spatial locality. Each entry in the SLDT tracks spatial hits and misses for the block and stores this information by incrementing or decrementing the Spatial Counter. Exact mechanism of detecting spatial hits can be found in [9].

#### 3.2. Compiler optimization

Compiler techniques for optimizing cache locality use loop and data transformations. In this section, we revise a technique that optimizes regular nested

loops to take advantage of a cache hierarchy. The compiler optimization methodology used in this work is as follows:

- Using affine loop and data transformations, we first optimize temporal and spatial locality aggressively.
- We then optimize register usage through unroll-and-jam and scalar replacement.

For the first step, we use an extended form of the approach presented in [5]. We have chosen this method for two reasons. First, it uses both loop and data transformations, and is more powerful than pure loop ([13]) and pure data ([12]) transformation techniques. Secondly, this transformation framework was readily available to us. It should be noted, however, that other locality optimization approaches such as [12] would result in similar output codes for the regular, array-based programs in our experimental suite. The second step is fairly standard and its details can be found in [4]. A brief summary of this compiler optimization strategy follows. Consider the following loop nest:

```

for (i=1; i<=N; i++)
  for (j=1; j<=N; j++)
    U[j] = V[j][i] + W[i][j];

```

The approach detects that the loop  $j$  (and consequently the loop  $i$ ) can be optimized by the compiler. Informally, the approach optimizes the nest (containing the loops  $i$  and  $j$ ) as follows. It first determines the intrinsic temporal reuse in the nest. In this example, there is temporal reuse for only the reference  $U[j]$ <sup>1</sup> the other references exhibit only spatial reuse [13]. Therefore, in order to exploit the temporal reuse in the innermost position, the loops are interchanged, making the loop  $i$  innermost. Now this loop accesses the array  $v$  along the rows, and the array  $w$  along the columns. These access patterns result in selection of a row-major memory layout for the array  $v$  and a column-major memory layout for the array  $w$ . Once the memory layouts have been determined, implementing them in a compiler that uses a fixed default layout for all arrays (e.g., row-major in C) is quite mechanical [12]. After this step, depending on the architectural model (e.g., number of registers, pipeline structure, etc.), the compiler applies scalar placement and unroll-and-jam.

## 4. METHODOLOGY

### 4.1. Setup

The SimpleScalar [3] processor simulator was modified to carry out the performance evaluations. SimpleScalar is an execution-driven simulator, which can simulate both in-order and out-of-order processors. The baseline processor

Table 29-1. Base processor configuration.

Issue width	4
L1 (data cache) size	32 K, 4-way set-associative cache, 32-byte blocks
L1 (instruction cache) size	32 K, 4-way set-associative cache, 32-byte blocks
L2 size	512 K, 4-way set-associative cache, 128-byte blocks
L1 access time	2 cycle
L2 access time	10 cycles
Memory access time	100 cycles
Memory bus width	8 bytes
Number of memory ports	2
Number of RUU entries	64
Number of LSQ entries	32
Branch prediction	bi-modal with 2048 entries
TLB (data) size	512 K, 4-way associative
TLB (instruction) size	256 K, 4-way associative

configuration for our experiments is described in Table 29-1. The simulator was modified to model a system with the hardware optimization schemes described in Section 3.1. The bypass buffer is a fully-associative cache with 64 double words and uses LRU replacement policy. The MAT has 4,096 entries and the macro-block sizes are set to 1 KB (as in [8]). In addition, a flag indicated whether to apply the hardware optimization or not. The instruction set was extended to include activate/deactivate instructions to turn this optimization flag ON/OFF. When the hardware optimization is turned off, we simply ignore the mechanism.

After extensive experimentation with different threshold values, a threshold value (Section 2.3) of 0.5 was selected to determine whether a hardware or a compiler scheme needs to be used for a given inner loop. In the benchmarks we simulated, however, this threshold was not so critical, because in all the benchmarks, if a code region contains irregular (regular) access, it consists mainly of irregular (regular) accesses (between 90% and 100%). In all the results presented in the next section, the performance overhead of ON/OFF instructions has also been taken into account.

## 4.2. Benchmarks

Our benchmark suite represents programs with a very wide range of characteristics. We have chosen three codes from SpecInt95 benchmark suite (*Perl*, *Compress*, *Li*), three codes from SpecFP95 benchmark suite (*Swim*, *Applu*, *Mgrid*), one code from SpecFP92 (*Vpenta*), and six other codes from several benchmarks: *Adi* from Livermore kernels, *Chaos*, *TPC-C*, and three queries from *TPC-D* (*Q1*, *Q3*, *Q6*) benchmark suite. For the TPC benchmarks, we implemented a code segment performing the necessary operations (to execute query).

An important categorization for our benchmarks can be made according

to their access patterns. By choosing these benchmarks, we had a set that contains applications with regular access patterns (*Swim*, *Mgrid*, *Vpenta*, and *Adi*), a set with irregular access patterns (*Perl*, *Li*, *Compress*, and *Applu*), and a set with mixed (regular + irregular) access patterns (remaining applications). Note that there is a high correlation between the nature of the benchmark (floating-point versus integer) and its access pattern. In most cases, numeric codes have regular accesses and integer benchmarks have irregular accesses. Table 29-2 summarizes the salient characteristics of the benchmarks used in this study, including the inputs used, the total number of instructions executed, and the miss rates (%). The numbers in this table were obtained by simulating the base configuration in Table 29-1. For all the programs, the simulations were run to completion. It should also be mentioned, that in all of these codes, the conflict misses constitute a large percentage of total cache misses (approximately between 53% and 72% even after aggressive array padding).

### 4.3. Simulated versions

For each benchmark, we experimented with four different versions – (1) *Pure Hardware* – the version that uses only hardware to optimize locality (Section 3.1); (2) *Pure Software* – the version that uses only compiler optimizations for locality (Section 3.2); (3) *Combined*– the version that uses both hardware and compiler techniques for the *entire duration* of the program; and (4) *Selective (Hardware/Compiler)* – the version that uses hardware and software approaches *selectively* in an integrated manner as explained in this work (our approach).

Table 29-2. Benchmark characteristics. ‘M’ denotes millions.

Benchmark	Input	Number of Instructions executed	L1 Miss Rate [%]	L2 Miss Rate [%]
Perl	primes.in	11.2 M	2.82	1.6
Compress	training	58.2M	3.64	10.1
Li	train.lsp	186.8M	1.95	3.7
Swim	train	877.5 M	3.91	14.4
Applu	train	526.0 M	5.05	13.2
Mgrid	mgrid.in	78.7M	4.51	3.3
Chaos	mesh.2k	248.4 M	7.33	1.8
Vpenta	Large enough to fill L2	126.7 M	52.17	39.8
Adi	Large enough to fill L2	126.9 M	25.02	53.5
TPC-C	Generated using TPC tools	16.5 M	6.15	12.6
TPC-D, Q1	Generated using TPC tools	38.9M	9.85	4.7
TPC-D, Q2	Generated using TPC tools	67.7M	13.62	5.4
TPC-D, Q3	Generated using TPC tools	32.4M	4.20	11.0

#### 4.4. Software development

For all the simulations performed in this study, we used two versions of the software, namely, the base code and the optimized code. Base code was used in simulating the pure hardware approach. To obtain the base code, the benchmark codes were transformed using an optimizing compiler that uses aggressive data locality optimizations. During this transformation, the highest level of optimization was performed (-O3). The code for the pure hardware approach is generating by turning of the data locality (loop nest optimization) using a compiler flag.

To obtain the optimized code, we first applied the data layout transformation explained in Section 2.1. Then, the resulting code was transformed using the compiler, which performs several locality-oriented optimizations including tiling and loop-level transformations. The output of the compiler (transformed code) is simulated using SimpleScalar. It should be emphasized that the pure software approach, the combined approach, and the selective approach all use the same optimized code. The only addition for the selective approach was the (ON/OFF) instructions to turn on and off the hardware. To add these instructions, we first applied the algorithm explained in Section 2 to mark the locations where (ON/OFF) instructions to be inserted. Then, the data layout algorithm was applied. The resulting code was then transformed using the compiler. After that, the output code of the compiler was fed into SimpleScalar, where the instructions were actually inserted in the assembly code.

### 5. PERFORMANCE RESULTS

All results reported in this section are obtained using the cache locality optimizer in [8, 9] as our hardware optimization mechanism. Figure 29-3 shows the improvement in terms of execution cycles for all the benchmarks

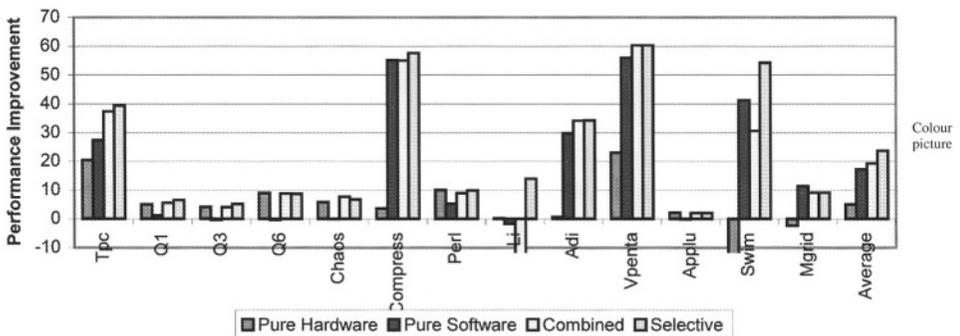


Figure 29-3. Base configuration.

in our base configuration. The improvements reported here are relative to the base architecture, where the base code was simulated using the base processor configuration (Table 29-1). As expected, the pure hardware approach yields its best performance for codes with irregular access. The average improvement of the pure hardware is 5.07%, and the average improvement for codes with regular access is only 2.19%. The average improvement of pure software approach, on the other hand, is 16.12%. The pure software approach does best for codes with regular access (averaging on a 26.63% percent improvement). The improvement due to the pure software approach for the rest of the programs is 9.5%. The combined approach improves the performance by 17.37% on average. The average improvement it brings for codes with irregular access is 13.62%. The codes with regular access have an average improvement of 24.36% when the combined approach is employed.

Although, the naively combined approach performs well for several applications, it does not always result in a better performance. These results can be attributed to the fact that the hardware optimization technique used is particularly suited for irregular access patterns. A hardware mechanism designed for a set of applications with specific characteristics can adversely affect the performance of the codes with dissimilar locality behavior. In our case, the codes that have locality despite short-term irregular access patterns are likely to benefit from the hardware scheme. The core data structures for the integer benchmarks have such access patterns. However, codes with uniform access patterns, such as numerical codes are not likely to gain much out of the scheme. This is because these codes exhibit high spatial reuse, which can be converted into locality by an optimizing compiler, or can be captured by employing a larger cache. The pure software approach has, on the other hand, its own limitations. While it is quite successful in optimizing locality in codes with regular access such as *Adi*, *Vpenta*, and *Swim* (averaging 33.3%), average improvement it brings in codes with high percentage of irregular accesses is only 0.8%. Our selective approach has an average improvement of 24.98%, which is larger than the sum of the improvements by the pure-hardware and pure-software approaches. On the average, the selective strategy brings a 7.61% more improvement than the combined strategy.

*Why does the selective approach increase the performance?* The main reason is that many programs have a *phase-by-phase nature*. History information is useful as long as the program is within the same phase. But, when the program switches to another phase, the information about the previous phase slows down the program until this information is replaced. If this phase is not long enough, the hardware optimization actually increases the execution cycles for the current phase. Intelligently turning off the hardware eliminates this problem and brings significant improvements.

To evaluate the robustness of our selective approach, we also experimented different memory latencies and cache sizes. Figure 29-4 shows the effect of increased memory latency. It gives the percentage improvement in execution cycles where the cost of accessing the main memory is increased to 200 cycles.

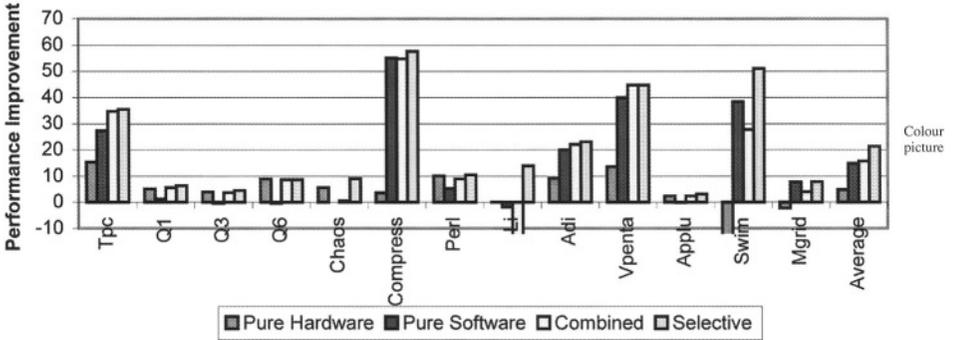


Figure 29-4. Larger memory latency (200 cycles).

As expected, the improvements brought by our scheme have increased. On the average, the selective scheme improved the performance by 28.52%, 22.27%, and 19.94% for integer, numerical and mixed codes, respectively. Figure 29-5 gives the results for larger L2 size. In the experiments, the L2 size is increased to 1 MB. Our selective strategy brings a 22.25% improvement over the base configuration. Although, it may look like the improvement has dropped, this is not the case. When we look at the relative improvement versus the pure hardware, pure software and combined approaches, we see that the relative performance remains the same. Figure 29-6 shows the percentage improvement in execution cycles when the size of the L1 data cache is increased to 64 K. On the average, the selective optimization strategy brings a 24.17% improvement.

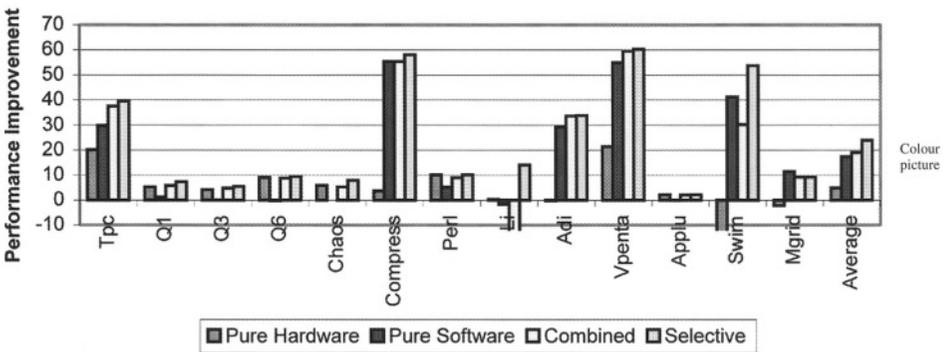


Figure 29-5. Larger L2 size (1 MB).

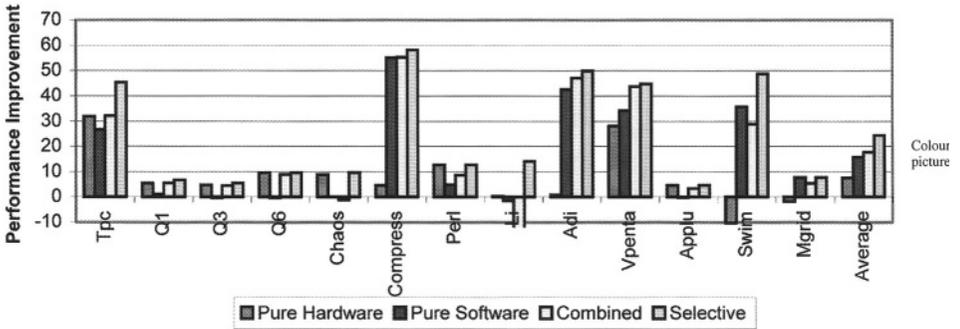


Figure 29-6. Larger L1 size (64 KB).

## 6. CONCLUSION

In this study, we presented a selective cache locality optimization scheme that utilizes both an optimizing compiler technique and a hardware-based locality optimization scheme. This scheme combines the advantages of both the schemes. Our simulation results also confirm this. Also, under different architectural parameters, our scheme consistently gave the best performance.

## NOTES

<sup>1</sup> This reuse is carried by the outer loop *i*. The locality optimizations in general try to put as much of the available reuse as possible into the innermost loop positions.

## REFERENCES

1. Agarwal and S. D. Pudar. "Column-Associative Caches: A Technique for Reducing the Miss Rate of Direct-Mapped Caches." In *Proceedings of 20th Annual International Symposium on Computer Architecture*, San Diego, CA, May 1993.
2. N. An et al. "Analyzing Energy Behavior of Spatial Access Methods for Memory-Resident Data." In *Proceedings of 27th International Conference on Very Large Data Bases*, pp. 411–420, Roma, Italy, September, 2001.
3. D. C. Burger and T. M. Austin. "The SimpleScalar tool set, version 2.0." *Technical Report CS-TR-97-1342*, University of Wisconsin, Madison, June 1998.
4. D. Callahan, S. Carr, and K. Kennedy. "Improving Register Allocation for Subscripted Variables." In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, New York.
5. M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. "Improving Locality Using Loop and Data Transformations in an Integrated Framework." In *Proceedings of 31st International Symposium on Micro-Architecture (MICRO'98)*, Dallas, TX, December 1998.
6. F. Cathoor, S. Wuytack, E. D. Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology – Exploration of Memory Organization for Embedded Multimedia System Design*. Kluwer Academic Publishers, June 1998.

7. A. Gonzalez, C. Aliagas, and M. Valero. "A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality." In *Proceedings of ACM International Conference on Supercomputing*, Barcelona, Spain, pages 338-347, July 1995.
8. T. L. Johnson and W. W. Hwu. "Run-Time Adaptive Cache Hierarchy Management via Reference Analysis." In *Proceedings of the 24th International Symposium on Computer Architecture*, June 2-4, 1997.
9. T. L. Johnson, M. C. Merten, and W. W. Hwu. "Run-Time Spatial Locality Detection and Optimization." In *Proceedings of the 30th International Symposium on Microarchitecture*, December 1-3, 1997.
10. N. P. Jouppi. "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers." In *Proceedings of 17th Annual International Symposium on Computer Architecture*, Seattle, WA, May 1990.
11. W. Li. *Compiling for NUMA Parallel Machines*. Ph.D. Thesis, Department of Computer Science, Cornell University, Ithaca, New York, 1993.
12. M. O'Boyle and P. Knijnenburg. "Non-Singular Data Transformations: Definition, Validity, Applications." In *Proceedings of 6th Workshop on Compilers for Parallel Computers*, pp. 287-297, Aachen, Germany, 1996.
13. M. Wolf and M. Lam. "A Data Locality Optimizing Algorithm." In *Proceedings of ACM Symposium on Programming Language Design and Implementation*, pp. 30-44, June 1991.

*This page intentionally left blank*

## Chapter 30

# RAPID CONFIGURATION & INSTRUCTION SELECTION FOR AN ASIP: A CASE STUDY

Newton Cheung<sup>1</sup>, Jörg Henkel<sup>2</sup> and Sri Parameswaran<sup>1</sup>

<sup>1</sup> *School of Computer Science & Engineering, University of New South Wales, Sydney, NSW 2052, Australia;* <sup>2</sup> *NEC Laboratories America, 4 Independence Way, Princeton, NJ 08540, USA*

**Abstract.** We present a methodology that maximizes the performance of Tensilica based Application Specific Instruction-set Processor (ASIP) through instruction selection when an area constraint is given. Our approach rapidly selects from a set of pre-fabricated coprocessors and a set of pre-designed specific instructions from our library (to evaluate our technology we use the Tensilica platform). As a result, we significantly increase application performance while area constraints are satisfied. Our methodology uses a combination of simulation, estimation and a pre-characterised library of instructions, to select the appropriate coprocessors and instructions. We report that by selecting the appropriate coprocessors and specific instructions, the total execution time of complex applications (we study a voice encoder/decoder), an application's performance can be reduced by up to 85% compared to the base implementation. Our estimator used in the system takes typically less than a second to estimate, with an average error rate of 4% (as compared to full simulation, which takes 45 minutes). The total selection process using our methodology takes 3–4 hours, while a full design space exploration using simulation would take several days.

**Key words:** ASIP, Instruction selection, methodology

## 1. INTRODUCTION

Embedded system designers face design challenges such as reducing chip area, increasing application performance, reducing power consumption and shortening time-to-market. Traditional approaches, such as employing general programmable processors or designing Application Specific Integrated Circuits (ASICs), do not necessarily meet all design challenges. While general programmable processors offer high programmability and lower design time, they may not satisfy area and performance challenges. On the other hand, ASICs are designed for a specific application, where the area and performance can easily be optimised. However, the design process of ASICs is lengthy, and is not an ideal approach when time-to-market is short. In order to overcome the shortcomings of both general programmable processors and ASICs, Application Specific Instruction-set Processors (ASIPs) have become popular in the last few years.

ASIPs are designed specifically for a particular application or a set of

applications. Designers of ASIPs can implement custom-designed specific instructions (custom-designed specific functional units) to improve the performance of an application. In addition, ASIP designers can attach pre-fabricated coprocessors (i.e. Digital Signal Processing Engines and Floating-Point units) and pre-designed functional units (i.e. Multiplier-Accumulate units, shifters, multipliers etc.). They can also modify hardware parameters of the ASIPs (i.e. register file size, memory size, cache size etc.).

As the number of coprocessors/functional units increase and more specific instructions are involved in an application, the design space exploration of ASIPs takes longer. Designers of ASIPs require an efficient methodology to select the correct combination of coprocessors/functional units and specific instructions. Hence, the design cycle and chip area is reduced and an application performance is maximized.

Research into design approaches for ASIPs has been carried out for about ten years. Design approaches for ASIPs can be divided into three main categories: architecture description languages [3, 4, 13, 16, 20]; compiler [5, 8, 18, 21] and methodologies for designing ASIPs [7,9].

The first category of architecture description languages for ASIPs is further classified into three sub-categories based on their primary focus: the structure of the processor such as the MIMOLA system [17]; the instruction set of the processor as given in nML [6] and ISDL [11]; and a combination of both structure and instruction set of the processor as in HMDES [10], EXPRESSION [12], LISA [13], PEAS-III (ASIP-Meister) [16], and FlexWare [19]. This category of approach generates a retargetable environment, including retargetable compilers, instruction set simulators (ISS) of the target architecture, and synthesizable HDL models. The generated tools allow valid assembly code generation and performance estimation for each architecture described (i.e. "retargetable").

In the second category, the compiler is the main focus of the design process using compiling exploration information such as data flow graph, control flow graph etc. It takes an application written in a high-level description language such as ANSI C or C++, and produces application characteristic and architecture parameter for ASIPs. Based on these application characteristics, an ASIP for that particular application can be constructed. In [21], Zhao used static resource models to explore possible functional units that can be added to the data path to enhance performance. Onion in [18] proposed a feedback methodology for an optimising compiler in the design of an ASIP, so more information is provided at the compile stage of the design cycle producing a better hardware ASIP model.

In the third category, estimation and simulation methodologies are used to design ASIPs with specific register file sizes, functional units and coprocessors. Gupta et al. in [9] proposed a processor evaluation methodology to quickly estimate the performance improvement when architectural modifications are made. However, their methodology does not consider an area constraint. Jain [15] proposed a methodology for evaluating register file size

in an ASIP design. By selecting an optimum register file size, they are able to reduce the area and energy consumption significantly.

Our methodology fits between second and third categories of the design approaches given above. In this paper, we propose a methodology for selecting pre-fabricated coprocessors and pre-designed (from our library) specific instructions for ASIP design. Hence, the ASIP achieves maximum performance within the given area constraint. There are four inputs of our methodology: an application written in C/C++, a set of pre-fabricated coprocessors, a set of pre-designed specific instructions and an area constraint. By applying this methodology, it configures an ASIP and produces a performance estimation of the configured ASIP.

Our work is closely related to Gupta et al. in [8] and IMSP-2P-MIFU in [14]. Gupta et al. proposed a methodology, which through profiling an application written in C/C++, using a performance estimator and an architecture exploration engine, to obtain optimal architectural parameters. Then, based on the optimal architectural parameters, they select a combination of four pre-fabricated components, being a MAC unit, a floating-point unit, a multi-ported memory, and a pipelined memory unit for an ASIP when an area constraint is given. Alternatively, the authors in IMSP-2P-MIFU proposed a methodology to select specific instructions using the branch and bound algorithm when an area constraint is given.

There are three major differences between the work at Gupta et al. in [8] & IMSP-2P-MIFU in [14] and our work. Firstly, Gupta et al. only proposed to select pre-fabricated components for ASIP. On the other hand, the IMSP-2P-MIFU system is only able to select specific instructions for ASIP. Our methodology is able to select both pre-fabricated coprocessors and pre-designed specific instructions for ASIP. The second difference is the IMSP-2P-MIFU system uses the branch and bound algorithm to select specific instructions, which is not suitable for large applications due to the complexity of the problem. Our methodology uses performance estimation to determine the best combinations of coprocessors and specific instructions in an ASIP for an application. Although Gupta et al. also use a performance estimator, they require exhaustive simulations between the four pre-fabricated components in order to select the components accurately. For our estimation, the information required is the hardware parameters and the application characteristics from initial simulation of an application. Therefore, our methodology is able to estimate the performance of an application in a short period of time. The final difference is that our methodology includes the latency of additional specific instructions into the performance estimation, where IMSP-2P-MIFU in [14] does not take this factor into account. This is very important since it eventually decides of the usefulness of the instruction when implemented into a real-world hardware.

The rest of this paper is organized as follows: section 2 presents an overview of the Xtensa and its tools; section 3 describes the design methodology on configurable core and functional units; section 4 describes the DSP

application used in this paper; section 5 presents the verification method and results. Finally, section 6 concludes with a summary.

## 2. XTENSA OVERVIEW & TOOLS

Xtensa is a configurable and extendable processor developed by Tensilica Inc. It allows designers to configure their embedded applications by constructing configurable core and designing application specific instructions using Xtensa software development tools. The project described in this paper used the Xtensa environment. Figure 30-1 shows the design flow of the Xtensa processor. In this section, we describe constructing configurable cores, designing specific instructions and the Xtensa tools.

The work carried out and the methodology developed, however, is general and could have been conducted with any other similar reconfigurable processor platform.

Xtensa's configurable core can be constructed from the base instruction set architecture (ISA) by selecting the Xtensa processor configuration options such as the Vectra DSP Engine, floating-point unit, 16-bit Multiplier etc. The quality of the configuration is dependent on the design experience of the designer who analyses an application. Our methodology tries to reduce this dependence based on quick performance estimation of the application.

The second part of designing the Xtensa processor is by using Tensilica Instruction Extensions (TIE). The main idea of TIE language is to design a specific functional unit to handle a specific functionality that is heavily used in the application, and hence this functional unit can lead to higher performance of the application. Figure 30-2 shows an example, which shows the

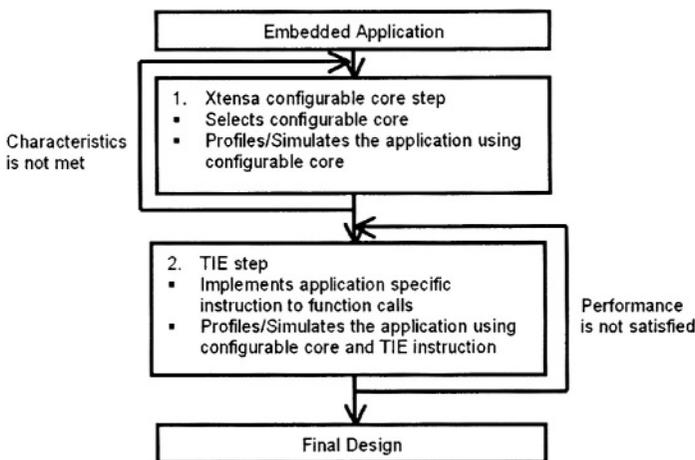


Figure 30-1. Xtensa application's design flow.

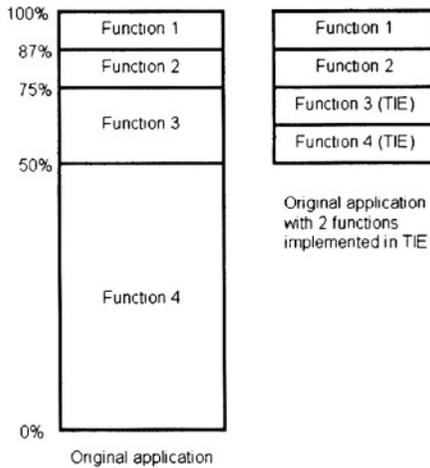


Figure 30-2. An example of application with 2 TIE.

percentage of execution time consumed by four functions and how TIE instructions can lead to higher performance of an application. In this example, function 4 consumes 50% of the execution time of an application and the second one (function 3) consumes 25% of the time. When both TIE instructions for function 3 and function 4 are implemented for this application, the execution time is able to reduce to half of the original execution time.

TIE language is a language that generates Verilog or VHDL code when compiled. The Verilog/VHDL code can then be put through the Synopsys tool Design Compiler to obtain timing and area.

However, adding TIE instructions may incur an increase in the latency of the processor. If this is the case, clock frequency must be slowed in order to compensate for the addition of TIE instructions. Since the simulator only considers the cycle-count, it would mislead the real performance of the application when the latency is increased. The real performance of an application should be the number of cycle-count multiplied by the latency caused by TIE instructions. Therefore, our methodology reinforces this critical point in our selection process of TIE instructions. For more information on TIE language, a detailed description can be found in [1, 2].

During the construction of configurable cores and the design of specific instructions, profiling (to get the application characteristics) and simulation (to find out the performance for each configuration) are required. In order to efficiently obtain the characteristics and performance information, Xtensa provides four software tools: a GNU C/C++ compiler, Xtensa's Instruction Set Simulator, Xtensa's profiler and TIE compiler. A detailed description about embedded applications using Xtensa processor can be found in [7].

### 3. METHODOLOGY

Our methodology consists of minimal simulation by utilising a greedy algorithm to rapidly select both pre-fabricated coprocessors and pre-designed specific TIE instructions for an ASIP. The goal of our methodology is to select coprocessors (these are coprocessors which are supplied by Tensilica) and specific TIE instructions (designed by us as a library), and to maximize performance while trying to satisfy a given area constraint. Our methodology divides into three phases: a) selecting a suitable configurable core; b) selecting specific TIE instructions; and c) estimating the performance after each TIE instruction is implemented, in order to select the instruction. The specific TIE instructions are selected from a library of TIE instructions. This library is pre-created and pre-characterised.

There are two assumptions in our methodology. Firstly, all TIE instructions are mutually exclusive with respect to other TIE instructions (i.e., they are different). This is because each TIE instruction is specifically designed for a software function call with minimum area or maximum performance gain. Secondly, speedup/area ratio of a configurable core is higher than the speedup/area ratio of a specific TIE instruction when the same instruction is executed by both (i.e., designers achieve better performance by selecting suitable configurable cores, than by selecting specific TIE instructions). It is because configurable core options are optimally designed by the manufacturer for those particular instruction sets, whereas the effectiveness of TIE instructions is based on designers' experience.

#### 3.1. Algorithm

There are three sections in our methodology: selecting Xtensa processor with different configurable coprocessor core options, selecting specific TIE instructions, and estimating the performance of an application after each TIE instruction is implemented. Firstly, minimal simulation is used to select an efficient Xtensa processor that is within the area constraint. Then with the remaining area constraint, our methodology selects TIE instructions using a greedy algorithm in an iterative manner until all remaining area is efficiently used. Figure 30-3 shows the algorithm and three sections of the methodology and the notation is shown in Table 30-1.

Since the configurable coprocessor core is pre-fabricated by Tensilica and is associated with an extended instruction set, it is quite difficult to estimate an application performance when a configurable coprocessor core is implemented within an Xtensa processor. Therefore, we simulate the application using an Instruction Set Simulator (ISS) on each Xtensa processor configuration without any TIE instructions. This simulation does not require much time as we have only a few coprocessor options available for use. We simulate a processor with each of the coprocessor options.

Through simulation, we obtain total cycle-count, simulation time, and a

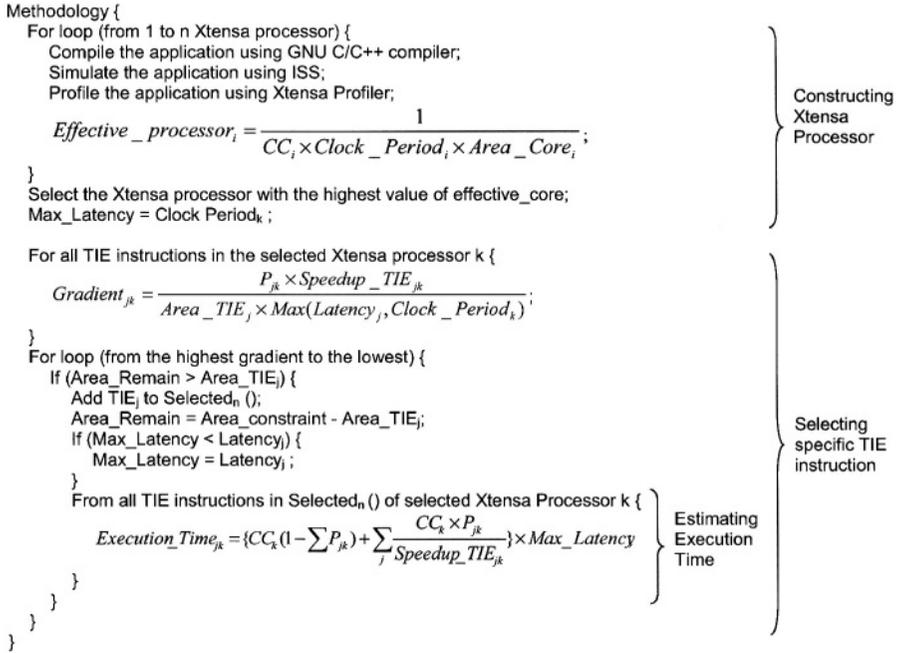


Figure 30-3. The algorithm.

call graph of the application for each Xtensa processor. Then we calculate the effectiveness of the processor for this application by considering total *cycle-count*, *clock period*, and the *area* of each configurable processor. The effectiveness of the Xtensa processor *i*, *Effective\_processor<sub>i</sub>*, is defined as:

$$Effective\_processor_i = \frac{1}{CC_i \times Clock\_Period_i \times Area\_Core_i} \tag{1}$$

This factor indicates a processor is most effective when it has the smallest chip size with the smallest execution time (cycle-count CC multiplied by the

Table 30-1. Notations for algorithm.

Notation	Descriptions
Area_Core <sub>i</sub>	Area in gate for processor <i>i</i>
Speedup_TIE <sub>ij</sub>	Speedup ratio of TIE instruction <i>j</i> in processor <i>I</i>
Area_TIE <sub>j</sub>	Area in gate of TIE instruction <i>j</i>
P <sub>ij</sub>	Percentage of cycle-count for function <i>j</i> in processor <i>i</i>
CC <sub>i</sub>	Total cycle-count spent in processor <i>i</i>
Clock_Period <sub>i</sub>	Clock period of processor <i>i</i>
Latency <sub>j</sub>	Latency of TIE instruction <i>j</i>
Selected <sub>i</sub> ()	Array stores selected TIE instructions in processor <i>i</i>

clock period) of an application. This factor calculates the ratio of performance per area for each processor, and our methodology selects the processor with the highest performance per area ratio that falls within the area constraint. This factor may not select the processor with the best performance. Note, that “Area\_Core” is not only an optimization goal but may also be a (hard) constraints.

Next, the methodology focuses on selecting specific TIE instructions. The selection of a specific TIE instruction is based on the area, speedup ratio, latency (maximum clock period in all TIE instructions and configured processor), and percentage of total cycle-count for the TIE instruction. We define the gradient of TIE instruction  $j$  in Xtensa processor  $k$  as:

$$Gradient_{jk} = \frac{P_{jk} \times Speedup\_TIE_{jk}}{Area\_TIE_j \times Max(Latency_j, Clock\_Period_k)} \quad (2)$$

This factor indicates that the performance gain per area of an application when the TIE instruction is implemented in the Xtensa processor. For example, if an application spent 30% of the time calling function X and 20% of the time calling function Y, two TIE instructions, say  $TIE_X$  and  $TIE_Y$ , are implemented to replace software function call X and software function call Y. The speedup of TIE instructions is 4 times and 6 times for  $TIE_X$  and  $TIE_Y$  respectively, and the area for the implemented TIE instructions are 3000 gates and 4000 gates respectively. The latency value (the clock period) for both instructions is the same. The gradient of functional units is 0.0004 and 0.0003 for  $TIE_X$  and  $TIE_Y$ . This means that  $TIE_X$  will be considered for implementation in the processor before  $TIE_Y$ . This example shows that the overall performance gain of an application will be the same whether  $TIE_X$  or  $TIE_Y$  is implemented. Since  $TIE_X$  has a smaller area,  $TIE_X$  is selected before  $TIE_Y$ . Equation 2 uses the remaining area effectively by selecting the specific TIE instruction with largest performance improvement per area. This is a greedy approach to obtain the largest performance improvement per area. This greedy approach proved to be effective and efficient, though may not prove to be as efficient for another application.

In the last section, after TIE instructions are selected, an estimation of execution time is calculated. The estimation of execution time for an Xtensa processor  $k$  with a set of selected TIE instruction (from 1 to  $j$ ) is defined as:

$$Execution\_Time_{jk} = \left\{ CC_k (1 - \sum_j P_{jk}) + \sum_j \frac{CC_k \times P_{jk}}{Speedup\_TIE_{ij}} \right\} \times Max\_Latency \quad (3)$$

where  $CC_k$  is the original total cycle-count of an application in processor  $k$ . The first part of the equation calculates the cycle-count of an application, then it multiplies with the maximum latency between TIE instructions. Maximum latency is the maximum clock period value of all the TIE instructions and configured processor.

#### **4. SPEECH RECOGNITION PROGRAM & ANALYSIS**

We use a speech recognition application as a case study to demonstrate the effectiveness of our approach. This application provides user voice control over Unix commands in a Linux' shell environment. It employs a template matching based recognition approach, which requires the user to record at least four samples for each Unix command that he/she wants to use. These samples are recorded and are stored in a file. Moreover, since this software involves a user's voice, for the sake of consistency, a test bench is recorded and is stored in another file.

The application consists of three main sections: record, pre-process, and recognition. In the "record" section, it first loads the configuration of the speaker, then it loads the pre-recorded test bench as inputs, and the pre-recorded samples into memory. After that, it puts the test bench into a queue passing through to the next section. In the "pre-process" section, it copies the data from the queue and divides the data into frame size segments. Then, it performs a filtering process using the Hamming window and applies single-precision floating-point 256-point FFT algorithm to the input data to minimize the work done in the following recognition section. Afterwards, it calculates the power spectrum of each frame and puts these frames back into the queue. Finally, in the "recognition" section, it implements the template matching approach utilising Euclid's distance measure in which it compares the input data with the pre-recorded samples that are loaded in memory during the "record" section. In the next step, it stores the compared results in another queue. If the three closest matches of pre-recorded samples are the same command, then the program executes the matched Unix command. However, if the input data does not match the pre-recorded samples, the program goes back to the "record" section to load the input again, and so on. Figure 30-4 shows the block diagram of speech recognition program.

As this application involves a human interface and operates in sequence, it is necessary to set certain real-time constraints. For example, it is assumed that each voice command should be processed within 1 second after the user finished his/her command. So, "record" section should take less than 0.25s to put all the data into a queue. While "pre-process" and "recognition" should consume less than 0.5s and 0.25s respectively.

Through a profiler we analysed the speech recognition software in a first approximation. Table 30-2 shows the percentage of selected software functions that are involved in this application in different Xtensa processors configurations that we denote as P1, P2 and P3. P1 is an Xtensa processor with the minimal configurable core options. P2 is a processor with Vectra DSP Engine and its associated configurable core options. P3 is a processor with a floating-point unit and its associated configurable core options (more configurations are possible, but for the sake of efficacy, only these three configurations are shown here). Moreover, in Table 30-2, the application spent 13.4% of time calling the single precision square root function in Xtensa

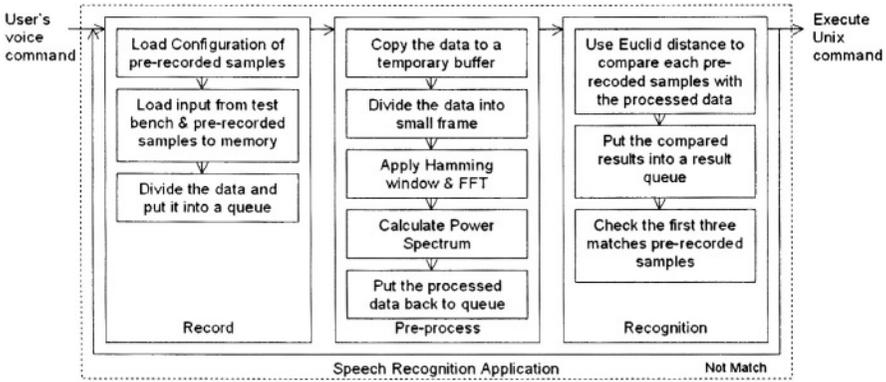


Figure 30-4. Speech recognition application.

processor P3. Moreover, this application simulates with a cycle-count of 422,933,748 and an instruction-count of 338,079,172 in an Xtensa processor with minimal configurable core options and with no additional instructions. The simulation time of this application is about 46 minutes and thus is a quite time consuming. The application consists 20 source files written in ANSI C, which include 50 subroutines and about 3500 lines of code. The compiled source code size is 620 Kbytes.

## 5. VERIFICATION METHODOLOGY & RESULTS

We have pre-configured three Xtensa processors, namely P1, P2 and P3, for this case study as explained above. As mentioned before, although Xtensa can be configured with other hardware parameters such as register file size, cache size etc., in our case, all processors are configured with the same register file size, cache size etc. We considered constructing an Xtensa processor P4 with DSP Engine and FP unit together, but this proved to be inefficient. The hardware cost and initial simulation result for the speech recognition application for each of these processors are shown in Table 30-3. Nine specific functional units (ten TIE instructions) are implemented and the corresponding

Table 30-2. Percentage of time spent for functions.

Function	Description	P1	P2	P3
Sqrtf	Single precision square root	1.30%	2.30%	13.4%
Mod3	Modular 3	0.20%	0.60%	3.00%
Logf	Single precision natural logarithm	2.70%	2.60%	2.40%
Divf	Single precision division	0.80%	1.80%	7.30%
Addf	Single precision addition	13.6%	25.1%	10.0%
Multf	Single precision multiplication	58.3%	31.1%	16.3%

Table 30-3. Hardware cost and initial simulation.

Xtensa Processor	P1	P2	P3
Area (mm <sup>2</sup> )	1.08	4.23	2.28
Area (gates)	35,000	160,000	87,000
Power (mW)	54	164	108
Clock rate (MHz)	188	158	155
Simulation Time (sec)	2,770.93	1,797.34	641.69
Cycle-count	422,933,748	390,019,604	131,798,414

information such as area in gates, and the speedup factor under each Xtensa processor are shown in Table 30-4. The down arrow in Table 30-4 represents a negative speedup, whereas the up arrow with a number, N, represents that the corresponding specific functional unit is N times faster than the software function call. Different sets of specific functional units are combined into a total of 576 different configurations representing the entire design space of the application. Figure 30-5 shows the verification methodology.

In order to verify our methodology, we pre-configured the 3 Xtensa processors and the 9 specific functional units (10 TIE instructions) at the beginning of the verification. Then we simulated the 576 different configurations using the ISS. We applied our methodology (to compare with the exhaustive simulation) to obtain the result under an area constraint. During the first phase of the methodology, just the Xtensa processor is selected without the TIE instructions. Table 30-4 provides the area, cycle-count, and clock rate for an application with each Xtensa processor. This information is used in equation 1 of the methodology. The information from Table 30-2 and Table 30-4 is retrieved from running initial simulation and verifying TIE instructions. The information provided in Table 30-3 is then used in the selection of TIE instructions. For equation 3, all the parameters are obtained from Table 30-2,

Table 30-4. Function unit's information.

Function	TIE instruction/s	Area (gate)	Speedup factor			Latency (ns)
			P1	P2	P3	
FP Addition	FA32	32000	↑ 8.14×	↑ 8.31×	↓	8.50
FP Division (1)	FFDIV	53800	↑ 17.4×	↓	↑ 15.9×	14.6
FP Division (2)	MANT, LP24, COMB	6800	↑ 6.48×	↑ 3.52×	↑ 5.28×	6.80
FP Multiplication	FM32	32000	↑ 11.6×	↑ 8.98×	↓	7.10
Natural Logarithm	FREXPLN	3300	↓	↓	↑ 1.10×	5.90
Modular 3	MOD3	5500	↑ 10.9×	↓	↑ 17.0×	6.40
Square Root (1)	LDEXP, FREXP	3300	↓	↓	↑ 3.30×	7.00
Square Root (2)	LDEXP	1100	↓	↓	↑ 2.50×	6.50
Square Root (3)	FREXP	3200	↓	↓	↑ 1.90×	6.90

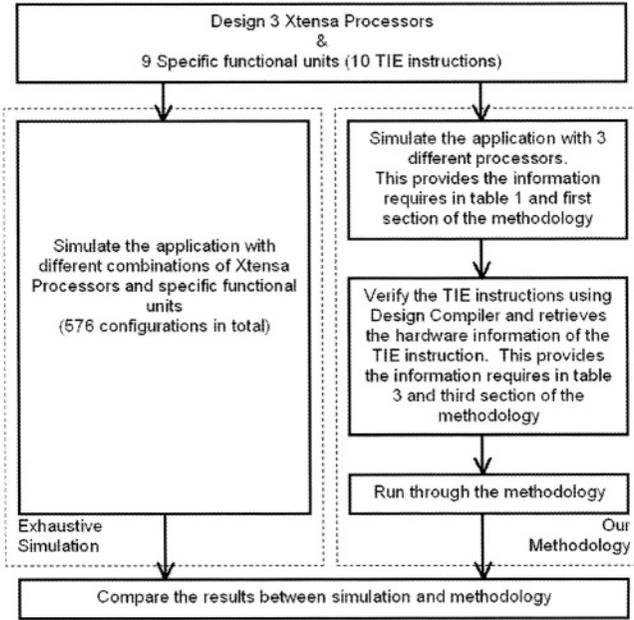


Figure 30-5. Verification methodology.

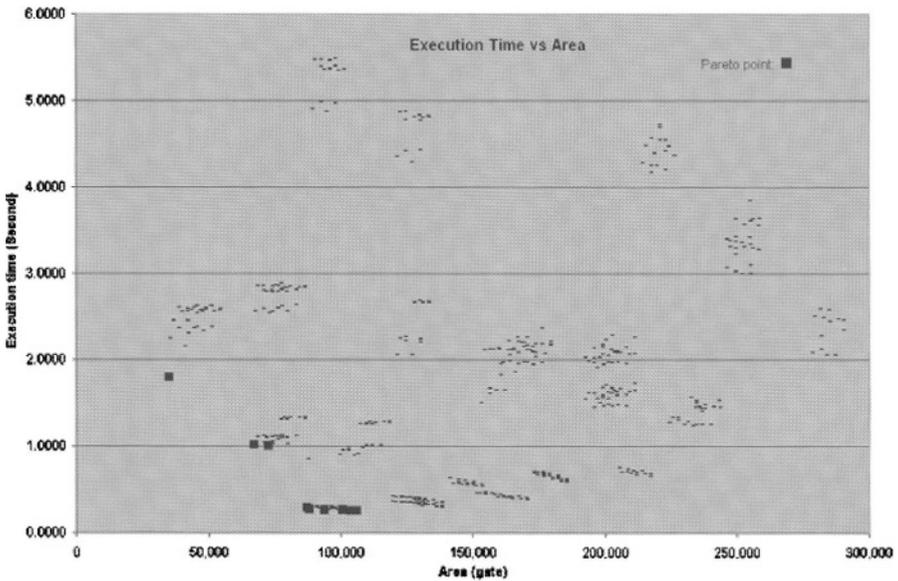


Figure 30-6 The results and the Pareto points.

Table 30-3, and Table 30-4. At the end, we compare results from simulation to our methodology.

The simulation results of 576 configurations are plotted in Figure 30-6 with execution time verse area. The dark squares are the points obtained using our methodology for differing area constraints. Those points selected by our methodology corresponded to the Pareto points of the design space graph. As mentioned above, the 576 configurations represent the entire design space and these Pareto points indicate the fastest execution time under a particular area constraint. Therefore, there are no extra Pareto points in this design space. Moreover, Table 30-5 shows the information of all Pareto points such as area in gates, simulated execution time in seconds, estimated execution time in seconds, latency, selected Xtensa processor and the replaced software function calls. Table 30-5 also indicates the estimation of application performance for each configuration is on average within 4% of the simulation result. For the fastest configuration (configuration 9 in Table 30-5), the application execution time is reduced to 85% of the original execution time (configuration 1 in Table 30-5). The fastest configuration is with Xtensa processor P3 (that is with a floating-point unit and its associated configurable core options). Seven TIE instructions (LDEXP, FREXP, MOD3, FREXPLN, MANT, LP24, COMB) are also implemented in the fastest configuration to replace four software functions. There are square root, modular 3, natural logarithm and floating-point division. The time for selection of core options and instructions are in the order of a few hours (about 3–4 hrs), while the exhaustive simulation method would take several weeks (about 300 hrs) to complete.

Table 30-5. Pareto points.

Config-uration	Area (gate)	Simulated execution time (sec.)	Estimated execution time (sec.)	% diff.	Latency of the proc.	Xtensa Proc. selected	Software function replaced
1	35,000	1.8018	–	–	5.32	P1	–
2	67,000	1.0164	1.1233	10.5	7.1	P1	FP mult
3	72,500	1.0147	1.1188	10.2	7.1	P1	Mod 3, FP mult
4	87,000	0.2975	–	–	6.45	P3	–
5	88,100	0.2738	0.2718	0.7	6.5	P3	Sqrt (2)
6	93,600	0.2670	0.2694	2.1	6.5	P3	Sqrt (1), Log
7	100,400	0.2632	0.2651	1.9	6.8	P3	Sqrt (1), Log, FP div (2)
8	103,700	0.2614	0.2642	1.0	6.9	P3	Sqrt (2), Mod 3, Log, FP div (2)
9	105,900	0.2586	0.2638	2.0	7.0	P3	Sqrt (1), Mod 3, Log, FP div (2)
Avg	–	–	–	4%	–	–	–

## 6. CONCLUSION

This paper describes a methodology to maximize application performance in an ASIP, through the selection of coprocessors and specific TIE instructions when an area constraint is given. The methodology described uses a combination of simulation and estimation to greedily construct an ASIP.

Our methodology has demonstrated how a speech recognition application can be designed within a reconfigurable processor environment (we used Xtensa). The application's execution time is reduced by 85% when a floating-point coprocessor is selected and seven of our proprietary TIE instructions (LDEXP, FREXP, MOD3, FREXPLN, MANT, LP24, COMB) are implemented to replace four software function calls: square root, modular 3, natural logarithm and floating-point division. In addition, our methodology was able to locate all nine Pareto points from the design space of 576 configurations. Finally, the performance estimation for the proposed implementation is on average within 4% of the simulation results.

## REFERENCE

1. Tensilica Instruction Extension (TIE). *Language Ref. Manual* (For Xtensa T1040 Processor Cores): Tensilica, Inc. 2001.
2. Tensilica Instruction Extension (TIE). *Language User's Guide* (For Xtensa T1040 Processor Cores): Tensilica, Inc. 2001.
3. A. Alomary, T. Nakata, Y. Honma, M. Imai, and N. Hikichi. "An ASIP Instruction Set Optimization Algorithm with Functional Module Sharing Constraint." Presented at IEEE/ACM ICCAD, Santa Clara, USA, 1993.
4. N. N. Binh, M. Imai and Y. Takeuchi. "A Performance Maximization Algorithm to Design ASIPs under the Constraint of Chip Area Including RAM and ROM Sizes." Presented at Proceedings of the ASP-DAC, Yokohama, Japan, 1998.
5. H. Choi, H. and I. C. Park. "Covare Pipelining for Exploiting Intellectual Properties and Software Codes in Processor-based Design." Presented at 13th Annual IEEE International ASIC/SOC, Arlington, VA, USA, 2000, pp. 153–157
6. A. Fauth. "Beyond Tool-Specific Machine Description." *Code Generation for Embedded Processors*, 1995, pp. 138–152.
7. R. Gonzalez. "Xtensa: A Configurable and Extensible Processor." *IEEE Micro*, Vol. 20, 2000.
8. T. V. K. Gupta, R. E. Ko, and R. Barua. "Compiler-Directed Customization of ASIP Cores." Presented at 10th International Symposium on Hardware/Software Co-Design, Estes Park, US, 2002
9. T. V. K. Gupta, P. Sharma, M. Balakrishnan, and S. Malik. "Processor Evaluation in an Embedded Systems Design Environment." Presented at Thirteenth International Conference on VLSI Design, Calcutta, India, 2000, pp. 98–103
10. J. C. Gyllenhaal, W. M. W. Hwu, and B. R. Rau. HMDDES Version 2.0 Specification, Uni of Illinois, 1996.
11. G. Hadjiyiannis, P. Russo, and S. Devadas. "A Methodology for Accurate Performance Evaluation in Architecture Exploration." Presented at 36th DAC, 1999, pp. 927–932
12. A. Halambi, P. Grun, V. Ganesh, A. Kahare, N. Dutt, and A. Nicolau. "EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability." Presented at DATE 99, 1999, pp. 485–490

13. A. Hoffmann, T. Kogel et al. "A Novel Methodology for the Design of Application-Specific Instruction-Set Processors (ASIPs) Using a Machine Description Language." *IEEE Transactions on CAD of Integrated Circuits and Systems*, Vol. 20, 2001, pp. 1338–1354.
14. M. Imai, N. N. Binh, and A. Shiomi. "A New HW/SW Partitioning Algorithm for Synthesizing the Highest Performance Pipelined ASIPs with Multiple Identical FUs." Presented at European Design Automation Conference, 1996, pp. 126–131
15. M. K. Jain, L. Wehmeyer et al. "Evaluating Register File Size in ASIP Design." Presented at 9th CODES, Denmark, 2001, pp. 109–115
16. S. Kobayashi, H. Mita, Y. Takeuchi, and M. Imai. "Design Space Exploration for DSP Applications Using the ASIP Development System Peas-III." Presented at IEEE International Conf. on Acoustics, Speech, and Signal Processing, 2002, pp. 3168–3171
17. R. Leupers and P/ Marwedel. "Retargetable Code Generation Based on Structural Processor Descriptions." *Design Automation for Embedded Systems*, Vol. 3, 1998, pp. 75–108.
18. F. Onion, A., Nicolau, and N. Dutt. "Incorporating Compiler Feedback into the Design of ASIPs." Presented at European Design and Test Conference, France, 1995, pp. 508–513
19. P. G. Paulin, C. Liem, T. C. May, and S. Sutawala. "Flexware: A Flexible Firmware Development Environment for Embedded Systems." *Code Generation for Embedded Processors*, 1995, pp. 65–84.
20. J. H. Yang, B. W. Kim et al. "MetaCore: An Application-Specific Programmable DSP Development System." *IEEE Transactions on VLSI Systems*, Vol. 8, 2000, pp. 173–183.
21. Q. Zhao, B., Mesman, and T. Basten. "Practical Instruction Set Design and Compiler Retargetability Using Static Resource Models." Presented at DATE'02, France, pp. 1021–1026

*This page intentionally left blank*

PART IX:

TRANSFORMATIONS FOR REAL-TIME SOFTWARE

*This page intentionally left blank*

# GENERALIZED DATA TRANSFORMATIONS

V. Delaluz<sup>1</sup>, I. Kadayif<sup>1</sup>, M. Kandemir<sup>1</sup> and U. Sezer<sup>2</sup>

<sup>1</sup> CSE Department, The Pennsylvania State University, University Park, PA 16802, USA;

<sup>2</sup> ECE Department, University of Wisconsin, Madison, WI 53706, USA;

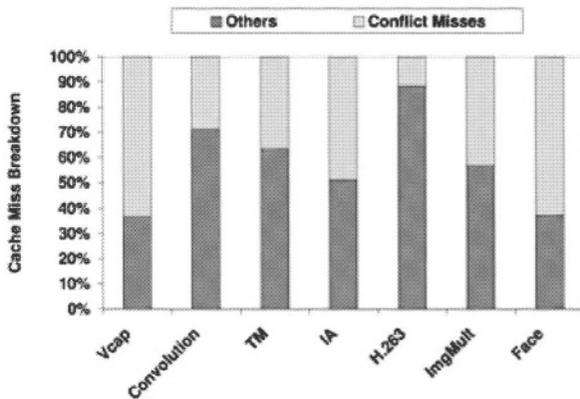
E-mail: {delaluz,kadayif,kandemir}@cse.psu.edu, sezer@ece.wisc.edu

**Abstract.** We present a compiler-based data transformation strategy, called the “generalized data transformations,” for reducing inter-array conflict misses in embedded applications. We present the theory behind the generalized data transformations and discuss how they can be integrated with compiler-based loop transformations. Our experimental results demonstrate that the generalized data transformations are very effective in improving data cache behavior of embedded applications.

**Key words:** embedded applications, data transformations, cache locality

## 1. INTRODUCTION

In many array-intensive embedded applications, conflict misses can constitute a significant portion of total data cache misses. To illustrate this, we give in Figure 31-1, for an 8 KB direct-mapped data cache, the breakdown of cache misses into conflict misses and other (capacity plus cold) misses for seven embedded applications from image and video processing.<sup>1</sup> We see that, on the average, conflict misses consist of 42.2% of total cache misses. In fact, in two applications (Vcap and Face), conflict misses constitute more than 50%

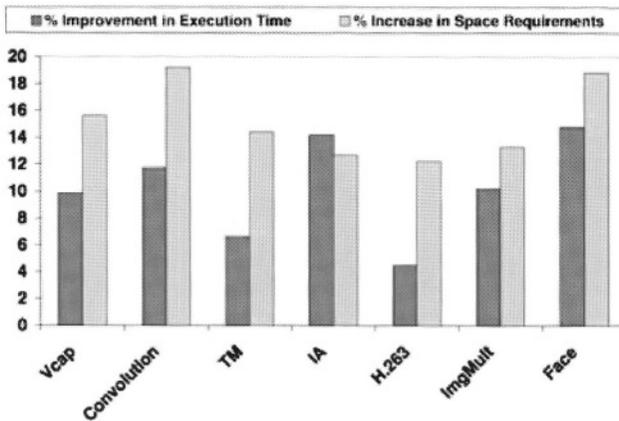


Colour picture

Figure 31-1. Contribution of conflict misses and other (capacity plus cold) misses.

of all misses. The reason for this behavior is the repetitive characteristic of conflict misses; that is, they tend to repeat themselves at regular (typically short) intervals as we execute loop iterations. The small associativities of the data caches employed in embedded systems also contribute to large number of conflict misses.

A well-known technique for reducing conflict misses is array padding [6]. This technique reduces conflict misses by affecting the memory addresses of the arrays declared in the application. It has two major forms: intra-array padding and inter-array padding. In intra-array padding, the array space is augmented with a few extra columns and/or rows to prevent the different columns (or rows) of the array from conflicting with each other in the data cache. For example, an array declaration such as  $A(N, M)$  is modified to  $A(N, M + k)$  where  $k$  is a small constant. This can help to prevent conflicts between two elements on different rows. Inter-array padding, on the other hand, inserts dummy array declarations between two original consecutive array declarations to prevent the potential conflict misses between these two arrays. For instance, a declaration sequence such as  $A(N, M), B(N, M)$  is transformed to  $A(N, M), D(k), B(N, M)$ , where  $D$  is a dummy array with  $k$  elements. This affects the array base addresses and can be used for reducing inter-array conflicts. While array-padding has been shown to be effective in reducing conflict misses in scientific applications [6], there is an important factor that needs to be accounted for when applying it in embedded environments: *increase in data space size*. Since data space demand of applications is the main factor that determines the capacity and cost of data memory configuration in embedded designs, an increase in data space may not be tolerable. To evaluate the impact of array padding quantitatively, we performed a set of experiments with our benchmark codes. Figure 31-2 gives the percentage reduction in (simulated) execution time (for an embedded MIPS processor with an 8 KB data cache) and the percentage increase in data space when array

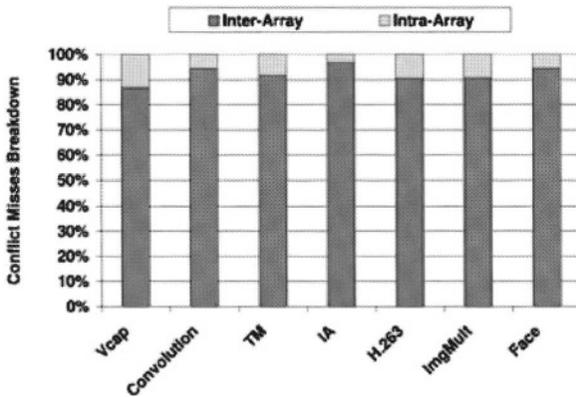


Colour picture

Figure 31-2. Impact of array padding.

padding is applied to our benchmarks. The specific array padding algorithm used to obtain these results is similar to that proposed by Rivera and Tseng [6] and represents the state-of-the-art. The values given in this graph are with respect to the original versions of codes. One can observe from these results that while array padding reduces the execution time by 10.3% on the average, it also increases the data space requirements significantly (15.1% on the average). To see whether most of conflict misses come from intra-array conflicts (i.e., the different portions of the same array are conflicting in the data cache) or from inter-array conflicts (i.e., the portions of different arrays are conflicting), we also measured the contribution of each type of conflict misses. The results presented in Figure 31-3 indicate that the overwhelming majority of conflict misses occur between the different arrays (92.1% on the average). Therefore, a technique that focuses on minimizing inter-array conflict misses without increasing the data space requirements might be very desirable in embedded environments.

In this work, we present such a *compiler-based* data transformation strategy for reducing inter-array conflict misses in embedded applications. This strategy maps the simultaneously used arrays into a common array space. This mapping is done in such a way that the elements (from different arrays) that are accessed one after another in the original execution are stored in consecutive locations in the new array space. This helps to reduce inter-array conflict misses dramatically. We call the data transformations that work on multiple arrays simultaneously the *generalized data transformations*. This is in contrast to many previous compiler works on data transformations (e.g., [2, 5]) that handle each array in isolation (that is, each array is transformed independently and the dimensionality of an array is not changed). Our approach is also different from the previously proposed array interleaving strategies (e.g., [3, 4]) in two aspects. First, our data transformations are more general than the classical transformations used for interleaving. Second, we present a strategy for selecting a dimensionality for the target common array space.



Colour picture

Figure 31-3. Contribution of inter-array and intra-array conflict misses.

We present the theory behind the generalized data transformations in Section 3. Section 4 presents our overall approach. Our experimental results (Section 5) demonstrate that the generalized data transformations are very effective in improving data cache behavior of embedded applications. Finally, in Section 6, we summarize our contributions.

## 2. ASSUMPTIONS AND BACKGROUND

We consider the case of references (accesses) to arrays with affine subscript functions in nested loops, which are very common in array-intensive embedded image/video applications [1]. Let us consider such an access to an  $m$ -dimensional array in an  $n$ -deep loop nest. We use  $\bar{i}$  to denote the iteration vector (consisting of loop indices starting from the outermost loop). Each array reference can be represented as  $X\bar{i} + \bar{x}$ , where the  $m \times n$  matrix  $X$  is called the reference matrix and the  $m$ -element vector  $\bar{x}$  is called the offset vector. In order to illustrate the concept, let us consider an array reference  $A(i + 3, i + j - 4)$  in a nest with two loops:  $i$  (the outer loop) and  $j$  (the inner loop). For this reference, we have

$$\bar{i} = \begin{pmatrix} i \\ j \end{pmatrix}, \quad X = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}, \quad \text{and} \quad \bar{x} = \begin{pmatrix} 3 \\ -4 \end{pmatrix}$$

In execution of a given loop nest, an element is reused if it is accessed (read or written) more than once in a loop. There are two types of reuses: temporal and spatial. Temporal reuse occurs when two references (not necessarily distinct) access the same memory location; spatial reuse arises between two references that access nearby memory locations [7]. The notion of nearby locations is a function of the memory layout of elements and the cache topology. It is important to note that the most important reuses (whether temporal or spatial) are the ones exhibited by the innermost loop. If the innermost loop exhibits temporal reuse for a reference, then the element accessed by that reference can be kept in a register throughout the execution of the innermost loop (assuming that there is no aliasing). Similarly, spatial reuse is most beneficial when it occurs in the innermost loop because, in that case, it may enable unit-stride accesses to the consecutive locations in memory. It is also important to stress that reuse is an intrinsic property of a program. Whenever a data item is present in the cache at the time of reuse, we say that the reference to that item exhibits locality. The key optimization problem is then to convert patterns of reuse into locality, which depends on a number of factors such as cache size, associativity, and block replacement policy. Consider the reference  $A(i + 3, i + j - 4)$  above in a nest with the outer loop  $i$  and the inner loop  $j$ . In this case, each iteration accesses a distinct element (from array  $A$ ); thus, there is no temporal reuse. However, assuming a row-major memory layout, successive iterations of the  $j$  loop (for a fixed value of  $i$ ) access the neighboring elements on the same row. We express this by saying

that the reference exhibits spatial reuse in the  $j$  loop. Since this loop ( $j$  loop) is the innermost loop, we can expect that this reuse will be converted into locality during execution. However, this would not be the case if  $i$  was the inner loop and  $j$  was the outer loop.

Consider the application of a non-singular linear loop transformation to an  $n$ -deep loop nest that accesses an array with the subscript function represented as  $X\bar{I} + \bar{x}$ . This transformation can be represented by an  $n \times n$  non-singular transformation matrix  $T$ , and maps the iteration  $\bar{I}$  of the original loop nest to the iteration  $\bar{I}' = T\bar{I}$  of the transformed loop nest [7]. On applying such a transformation, the new subscript function is obtained as  $XT^{-1}\bar{I} + \bar{x}$ ; that is, the new (transformed) reference matrix is  $XT^{-1}$ . The loop transformation also affects the loop bounds of the iteration space that can be computed using techniques such as Fourier-Motzkin elimination [7]. For convenience, we denote  $T^{-1}$  by  $Q$ . For example, a loop transformation such as:

$$T = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

transforms an original loop iteration  $(i, j)^T$  to  $(j, i)^T$ .

### 3. GENERALIZED DATA TRANSFORMATIONS

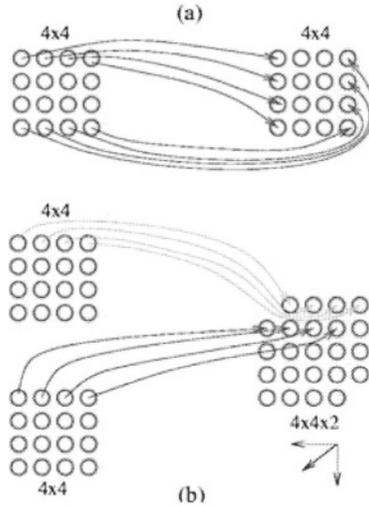
Given an array reference  $A(X\bar{I} + \bar{x})$  to an  $m$ -dimensional array  $A$  in an  $n$ -level nested loop, a data transformation transforms this reference to  $A'(X'\bar{I} + \bar{x}')$ . Here,  $A'$  corresponds to the transformed array, and  $X'$  and  $\bar{x}'$  denote the transformed reference matrix and constant vector, respectively. In general,  $A'$  can have  $m'$  dimensions, where  $m' \neq n$ .

Formally, we can define a data transformation for an array  $A$  using a pair  $(m_A, \bar{m}_A)$ . This pair (referred to as *data transformation* henceforth) performs the following two mappings:

$$\begin{aligned} X &\rightarrow M_A X \\ \bar{x} &\rightarrow M_A \bar{x} + \bar{m}_A \end{aligned}$$

In other words, the original reference matrix  $X$  is transformed to  $M_A X$  and the original constant vector  $\bar{x}$  is transformed to  $M_A \bar{x} + \bar{m}_A$ . We refer to  $M_A$  as the *data transformation matrix* and  $\bar{m}_A$  as the *displacement vector*. If  $A(X\bar{I} + \bar{x})$  is the original reference and  $A'(X'\bar{I} + \bar{x}')$  is the transformed reference, we have  $X' = M_A X$  and  $\bar{x}' = M_A \bar{x} + \bar{m}_A$ . Note that  $M_A$  is  $m'$ -by- $m$  and  $\bar{m}_A$  has  $m'$  entries. An example data transformation is depicted in Figure 31-4(a). In this data transformation, a  $4 \times 4$  array (for illustrative purposes) is transposed; that is, the array element  $(i, j)^T$  in the original array is mapped to array element  $(j, i)^T$  in the transformed space. Note that in this case we have:

$$M_A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad \text{and} \quad \bar{m}_A = \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$



Colour picture

Figure 31-4. (a) 2D to 2D data mapping. (b) 2D to 3D generalized data mapping.

### 3.1. Motivation

In many previous compiler-based studies that employ data transformations for improving cache behavior (e.g., [5]), only the cases where  $m' = m$  are considered. This is because many of these studies target at reducing capacity misses, rather than conflict misses, and, focusing on each array individually is sufficient for reducing capacity misses in most cases. However, such an approach may not be very effective with conflict misses. Consider, for example, the following loop nest:

```

for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    k += A(i, j) + B(i, j)

```

In this nest, assuming row-major memory layouts (as in C), both the references exhibit perfect spatial locality when considered in isolation. However, if the base addresses of arrays  $A$  and  $B$  happen to map on the same cache line, a miss rate of 100% is possible due to the data cache conflicts between these two array references (i.e., inter-array conflicts).

A generalized data transformation, on the other hand, can prevent this by mapping them to the same array index space (data space). Consider, for example, the following data transformations (depicted in Figure 31-4(b) for  $m = 2$  and  $m' = 3$ ):

$$\begin{aligned}
 A(i, j) &\rightarrow A'(i, j, 0) \\
 B(i, j) &\rightarrow A'(i, j, 1)
 \end{aligned}$$

After these transformations are applied, a given loop iteration  $(i, j)^T$  accesses two consecutive elements from  $A'$  (the transformed array). Consequently, the chances for conflict misses are reduced significantly. Note that, in this specific example, we have

$$M_A = M_B = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}; \quad \bar{m}_A = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}; \quad \text{and} \quad \bar{m}_B = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}.$$

In the next subsection, we formulate the problem of determining the data transformation matrices and displacement vectors for each array in a given application.

### 3.2. Problem formulation and solution

In this section, we formulate the problem at several levels of complexity, starting from the simplest case and ending with the most general case. Let us assume first that we have a single nest that accesses two different arrays using a single reference per array. Without loss of generality, we use  $A(X_1\bar{I} + \bar{x}_1)$  and  $B(X_2\bar{I} + \bar{x}_2)$  to refer to these references. Our task is to determine two data transformations  $(M_A, \bar{m}_A)$  (for array  $A$ ) and  $(M_B, \bar{m}_B)$  (for array  $B$ ) such that a given iteration accesses consecutive elements in the loop body.

In mathematical terms, since the transformed references are  $M_A X_1 \bar{I} + M_A \bar{x}_1 + \bar{m}_A$  and  $M_B X_2 \bar{I} + M_B \bar{x}_2 + \bar{m}_B$ , for the best data locality, our data transformations should satisfy the following two constraints:

$$M_A X_1 = M_B X_2 \\ (M_A \bar{x}_1 + \bar{m}_A) - (M_B \bar{x}_2 + \bar{m}_B) = (0, 0, \dots, 0, 0, 1)^T$$

The first constraint demands that the transformed reference matrices should be the same so that the difference between the memory locations accessed by these two references (in a given iteration) does not depend on a specific loop iteration value. This is important as if the difference between the accessed locations depends on loop iterator value, the locality becomes very difficult to exploit. The second constraint, on the other hand, indicates that the difference between the said memory locations should be 1. If we are able to satisfy these two constraints, this means that a given loop iteration accesses successive elements in memory; therefore, the possibility of cache conflict within a loop iteration is eliminated. One way of determining the data transformations  $(M_A, \bar{m}_A)$  and  $(M_B, \bar{m}_B)$  from these constraints is as follows. First, from the first constraint above, we can determine  $M_A$  and  $M_B$ . Then, substituting (the values of the elements of) these matrices in the second constraint, we can solve that constraint for  $\bar{m}_A$  and  $\bar{m}_B$ . For example, in the loop nest above (in Section 3.1), this strategy determines the data transformations given earlier.

It should be stressed that this approach can improve capacity misses as well (in addition to conflict misses). This is because in determining the data trans-

formation matrices (from the first constraint) we can improve self-spatial reuse too. In the following, we generalize our method to handle more realistic cases. We also present a general solution method that can be employed within an optimizing compiler that targets embedded environments where data space optimizations (minimizations) are critical.

Let us now focus on a single nest with multiple arrays. Assume further that each array is accessed through a single reference only. Without loss of generality, let  $A_i(X_i \bar{I} + \bar{x}_i)$  be the reference to array  $A_i$ , where  $1 \leq i \leq s$ . We also assume that these references are touched (during program execution) in increasing values of  $i$  (i.e., from 1 to  $s$ ). We want to determine data transformations ( $M_{A_i}, m_{A_i}$ ). We have two different sets of constraints (one for the data transformation matrices and the other for the displacement vectors):

$$\begin{aligned} M_{A_i} X_i &= M_{A_j} X_j \\ (M_{A_{(k+1)}} \bar{x}_1 + \bar{m}_{A_{(k+1)}}) - (M_{A_k} \bar{x}_k + \bar{m}_{A_k}) &= (0, 0, \dots, 0, 0, 1)^T, \end{aligned}$$

where  $1 \leq i, j \leq s$  and  $1 \leq k \leq (s - 1)$ . Satisfying these constraints for all possible  $i, j$  and  $k$  values guarantees good spatial locality during nest execution.

The discussion above assumes that each array has a single reference in the loop body. If we have multiple references to the same array, then the corresponding constraints should also be added to the constraints given above. Let us assume that there are  $t_i$  references to array  $A_i$  and that these references are accessed (during nest execution) in the order of 1, 2,  $\dots$ ,  $t_i$ . So, for a given array  $A_i$ , we have the following constraints:

$$\begin{aligned} M_{A_i} X_{i,j} &= M_{A_i} X_{i,j'} \\ (M_{A_i} \bar{x}_{i,(k+1)} + \bar{m}_{A_i}) - (M_{A_i} \bar{x}_{i,k} + \bar{m}_{A_i}) &= (0, 0, \dots, 0, 0, 1)^T, \end{aligned}$$

where  $1 \leq j, j' \leq t_i$ ,  $1 \leq k \leq (t_i - 1)$ , and  $X_{i,j}$  and  $\bar{x}_{i,j}$  are the  $j$ th reference matrix and constant vector for array  $A_i$ , respectively.

It should be observed that the system of constraints given above may not always have a valid solution. This is because the second constraint above simplifies to  $M_{A_i}(\bar{x}_{i,(k+1)} - \bar{x}_{i,k}) = (0, 0, \dots, 0, 0, 1)^T$ , and it may not be possible to find an  $M_{A_i}$  to satisfy both this and  $M_{A_i} X_{i,j} = M_{A_i} X_{i,j'}$ . We note, however, that if  $X_{i,j} = X_{i,j'}$ , we can drop  $M_{A_i} X_{i,j} = M_{A_i} X_{i,j'}$  from consideration (as it is always satisfied). This case occurs very frequently in many embedded image processing applications (e.g., in many stencil-type computations). As an example, consider the references  $A(i, j)$  and  $A(i - 1, j + 1)$  in a nest with two loops,  $i$  and  $j$ . Since  $x_{i,j} - x_{i,j'} = (1, -1)^T$ , we need to select a  $M_{A_i}$  which satisfies the following constraint:

$$M_{A_i} = \begin{pmatrix} 1 \\ -1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

It is easy to see that a possible solution is:

$$M_A = \begin{pmatrix} 1 & 1 \\ 0 & 0 \\ 1 & 0 \end{pmatrix}$$

The transformed references are then  $A'(i + j, 0, i)$  and  $A'(i + j, 0, i + 1)$ . We observe that for a given loop iteration  $(i, j)^T$ , these two references access consecutive memory locations.

So far, we have focused only on a single nest. Many large array-intensive embedded applications consist of multiple nests. Our formulation above can easily extend to the cases where we have multiple nests in the application. First, if two nests of an application do not share any array between them, then there is no point in trying to handle these two nests together. In other words, in this case, each nest can be handled independently. On the other hand, if the nests share arrays, then the compiler needs to consider them together. In mathematical terms, the constraints (equations) given above should be written for each nest separately. Note, however, that the equations for nests that access the same array  $A$  should use the same data transformation matrix and same displacement vector (in each nest). This is because in this study we assign a single layout (i.e., a single data transformation) to each array; that is, we do not consider dynamic data transformations during the course of execution.

Given a program with multiple arrays referenced by multiple nest, the system of equations built (in terms of data transformation matrices and displacement vectors) may not have a solution. In this case, to find a solution, we need to drop some equations (constraints) from consideration. To select the equations to drop, we can rank the array references with respect to each other. More specifically, if an array reference is (expected to be) touched (during execution) more frequently than another reference, we can drop the latter from consideration and try to solve the resulting system again. We repeat this process until the resulting system has a solution. To determine the relative ranking of references, our current approach exploits profile information. More specifically, it runs the original program several times with typical input sets and records the number of times each array reference is used (touched). Then, based on these numbers, the references are ordered.

### 3.3. Dimensionality selection

In our discussion so far, we have not explained how we determine the dimensionality of the target (common) array (data space). Consider the following nest which accesses four two-dimensional (2D) arrays.

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    k += A(i,j) + B(i,j) + C(i,j) + D(i,j)
```

These arrays can be mapped to a common array space in multiple ways. Three such ways are given in Table 31-1. The second column shows the case where the target space (i.e., the common array space) is two-dimensional. The next two columns illustrate the cases where the target space is three-dimensional and four-dimensional, respectively. While in these cases we do not have explicit coefficients (as in the 2D case), the address computation (that will be performed by the back-end compiler) is more involved as the number of dimensions is larger. It should be emphasized that all these three mapping strategies have one common characteristic: for a given loop iteration, when we move from one iteration to another, the array elements accessed are consecutively stored in memory (under a row-major memory layout). In selecting the dimensionality of the common address space, our current implementation uses the following strategy. Let  $q$  be the number of references (to distinct arrays) in the loop body and  $m$  the dimensionality of the original arrays. We set the dimensionality of the common (array) space to  $m' = \max\{q, m\}$ . This is reasonable, because as we mentioned above, there is no point in selecting a dimensionality which is larger than the number of references in the loop body. Also, we want the dimensionality of the common to be at least as large as the dimensionality of the original arrays.

#### 4. DISCUSSION AND OVERALL APPROACH

So far, we have discussed the mechanics of the generalized data transformations and tried to make a case for them. As will be shown in the next section, the generalized data transformations can bring about significant performance benefits for array-intensive embedded applications. In this section, we discuss some of the critical implementation choices that we made and summarize our overall strategy for optimizing data locality.

An important decision that we need to make is to select the arrays that will be mapped onto a common address space. More formally, if we have  $s$  arrays in the application, we need to divide them into *groups*. These groups do not have to be disjoint (that is, they can share arrays); however, this can make the optimization process much harder. So, our current implementation uses only disjoint groups. The question is to decide which arrays need to be placed into the same group. A closer look at this problem reveals that there are several factors that need to be considered. For example, the dimensionality of the arrays might be important. In general, it is very difficult (if not impossible) to place arrays with different dimensionalities into the common space. So, our current implementation requires that the arrays to be mapped together should have the same number of dimensions. However, not all the arrays with the same number of dimensions should be mapped together. In particular, if two arrays are not used together (e.g., in the same loop nest), there is no point in trying to map them to a common array space to improve locality. Therefore, our second criterion to select the arrays to be mapped

together is that they should be used together in a given nest. To have a more precise measure, our current approach uses the following strategy. We first create an affinity graph such that the nodes represent arrays and the edges represent the fact that the corresponding arrays are used together. The edge weights give the number of loop iterations (considering all the loop nests in the code) that the corresponding arrays are used together. Then, we cluster the nodes in such a way that the nodes in a given cluster exhibit high affinity (that is, they are generally used together in nests). Each cluster then corresponds to a group provided that the nodes in it also satisfy the other criteria. The last criterion we consider is that the arrays in the same group should be accessed with the same frequency. For example, if one array is accessed by the innermost loop whereas the other array is not, there is not much point in trying to place these arrays into a common address space.

Another important decision is to determine the mapping of the arrays in a given group to the corresponding common address space. To do this, we use the affinity graph discussed in the previous paragraph. If the weight of an edge in this graph is very high, that means it is beneficial to map the corresponding arrays in such a way that the accesses to them occur one after another. As an example, suppose that we have three two-dimensional arrays are to be mapped onto the same address space. Suppose also that the common address space is three-dimensional. If two of these arrays are used together more frequently, then, if possible, their references should be mapped to  $A'(i, j, 0)$  and  $A'(i, j, 1)$ , respectively (or to  $A'(i, j, 1)$  and  $A'(i, j, 2)$ ); that is, they should be consecutive in memory.

Based on this discussion, our overall optimization strategy works as follows. First, we build the affinity graph and determine the groups taking into account the dimensionality of the arrays and access frequency. Then, we use the strategy discussed in the previous two sections in detail to build a system of equations. After that, we try to solve this system. If the system has no solution, we drop some equations from consideration (as explained earlier), and try to solve the resulting system again. This iterative process continues until we find a solution. Then, the output code is generated. The code generation techniques for data transformations are quite well-known and can be found elsewhere [2].

## 5. EXPERIMENTAL SETUP AND RESULTS

All results presented in this section are obtained using an in-house execution-driven simulator that simulates an embedded MIPS processor core (5Kf). 5Kf is a synthesizable 64-bit processor core with an integrated FPU. It has a six-stage, high-performance integer pipeline optimized for SoC design that allows most instructions to execute in 1 cycle and individually configurable instruction and data caches. The default configuration contains separate 8 KB, direct-mapped instruction and data caches. Both the caches have a

(default) line (block) size of 32 bytes. In all experiments, a miss penalty of 70 cycles is assumed. Our simulator takes a C code as input, simulates its execution, and produces statistics including cache hit/miss behavior and execution time.

To measure the effectiveness of our approach, we used seven array-intensive embedded applications from image and video processing domain. *Vcap* is a video capture and processing application. It generates video streams of different picture sizes, color spaces, and frame rates. *Convolution* is an efficient implementation of a convolution filter. One of the abilities of this benchmark is that it can process data elements of variable lengths. *TM* is an image conversion program that converts images from TIFF to MODCA or vice versa. *IA* is an image understanding code that performs target detection and classification. *H.263* is a key routine from a simple H.263 decoder implementation. *ImgMult* is a subprogram that multiplies three images with each other, and adds the resulting images and one of the input images. *Face* is a face recognition algorithm. More details about these applications are beyond the scope of this work.

Figure 31-5 gives the percentage reduction in cache misses (as compared to the original codes) when the global data transformations are used (without the support of loop transformations). Each bar in this graph is broken into two portions to show the reductions in conflict misses and other misses separately. We see from these results that overall (when considered all types of cache misses) we have nearly a 53% reduction in misses. We also observe that more than half of this reduction occurs in conflict misses, clearly demonstrating the effectiveness of our approach in reducing conflict misses. It is also important to compare these results with those of array-padding presented in Figure 31-2. Our generalized data transformations bring much more significant reductions in cache misses as compared to array padding. More importantly, these benefits are obtained without increasing the data space requirements of applications.

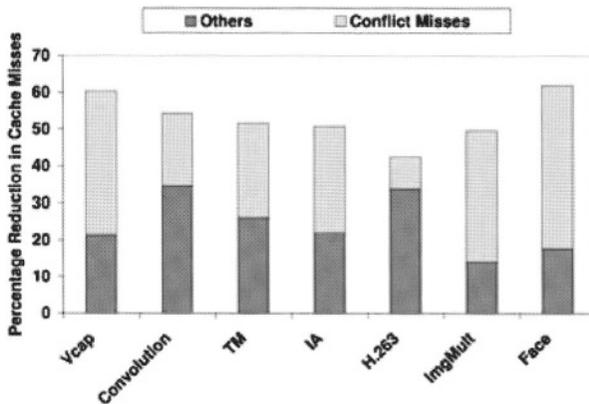


Figure 31-5. Reduction in cache misses.

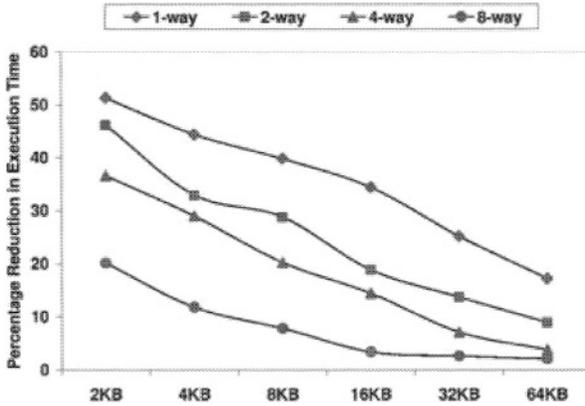


Figure 31-6. Reduction in execution time under different cache configurations.

The graph in Figure 31-6 shows the percentage reduction in execution times (again, with respect to the original codes) with different cache sizes and associativities. Each point in this graph represents an average value across all our benchmarks. One can observe that, using a larger cache or a cache with a higher set-associativity reduces the effectiveness of our approach. The reason that the percentage savings due to our strategy reduce when the cache size or associativity is increased can be explained as follows. When the cache capacity/associativity is increased, the original (i.e., the one without optimization) version starts to perform better as more data are captured in the cache (as compared to the case with the small cache size/associativity). While our approach also performs better with the increased cache size/associativity, the improvement rate of the original codes is much higher than that of the optimized codes. Consequently, we observe a reduction in execution time savings as we increase the cache capacity/associativity. Nevertheless, even with a 32 KB, 2-way data cache, we achieve around 15% reduction in execution time.

### 6. CONCLUDING REMARKS

Most existing work based on data transformations for improving data cache behavior of array dominated codes target a single array at a time. In this work, we develop a new, generalized data space transformation theory and present experimental data to demonstrate its effectiveness. Our results clearly indicate that the proposed approach is very successful in practice.

**NOTE**

<sup>1</sup> The important characteristics of our benchmarks as well as detailed experimental results will be given later.

**REFERENCES**

1. F. Catthoor et al. *Custom Memory Management Methodology – Exploration of Memory Organization for Embedded Multimedia System Design*. Kluwer Academic Publishers, 1998.
2. M. Cierniak and W. Li. “Unifying Data and Control Transformations for Distributed Shared Memory Machines.” In *Proceedings of Programming Language Design and Implementation*, pp. 205–217, 1995.
3. C. Ding. *Improving Effective Bandwidth through Compiler Enhancement of Global and Dynamic Cache Reuse*, Ph.D. Thesis, Rice University, Houston, Texas, January 2000.
4. M. Kandemir. “Array Unification: A Locality Optimization Technique.” In *Proceedings of International Conference on Compiler Construction, ETAPS’2001*, April, 2001.
5. M. Kandemir et al. “Improving Locality Using Loop and Data Transformations in an Integrated Framework.” In *Proceedings of International Symposium on Microarchitecture*, Dallas, TX, December, 1998.
6. G. Rivera and C.-W. Tseng. “Data Transformations for Eliminating Conflict Misses.” In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.
7. M. Wolfe. *High Performance Compilers for Parallel Computing*, Addison-Wesley Publishing Company, 1996.

# SOFTWARE STREAMING VIA BLOCK STREAMING

Pramote Kuacharoen, Vincent J. Mooney III and Vijay K. Madiseti  
*School of Electrical and Computer Engineering, Georgia Institute of Technology, USA*

**Abstract.** Software streaming allows the execution of stream-enabled software on a device even while the transmission/streaming of the software may still be in progress. Thus, the software can be executed while it is being streamed instead of requiring the user to wait for the completion of the software's download. Our streaming method can reduce application load time seen by the user since the application can start running as soon as the first executable unit is loaded into the memory. Furthermore, unneeded parts of the application might not be downloaded to the device. As a result, resource utilization such as memory and bandwidth usage may also be more efficient. Using our streaming method, an embedded device can support a wide range of real-time applications. The applications can be run on demand. In this paper, a streaming method we call *block streaming* is proposed. Block streaming is determined at the assembly code level. We implemented a tool to partition real-time software into parts which can be transmitted (streamed) to an embedded device. Our streaming method was implemented and simulated on a hardware/software co-simulation platform in which we used the PowerPC architecture.

**Key words:** software streaming, embedded software streaming

## 1. INTRODUCTION

Today's embedded devices typically support various applications with different characteristics. With limited storage resources, it may not be possible to keep all features of the applications loaded on an embedded device. In fact, some software features may not be needed at all. As a result, the memory on the embedded device may not be efficiently utilized. Furthermore, the application software will also likely change over time to support new functionality; such change may occur quite rapidly in the case of game software [1]. Today, the user most likely has to download the software and install it (the exception to this is Java, which, since it runs on a virtual machine, does not strictly require "installation" as long as the Java Virtual Machine is already installed). This means that the entire software must be downloaded before it can be run. Downloading the entire program delays its execution. In other words, *application load time* (the amount of time from when the application is selected to download to when the application can be executed) is longer than necessary. To minimize the application load time, the software should

be executable while downloading. Software streaming enables the overlapping of transmission (download) and execution of software.

Software to be streamed must be modified before it can be streamed over a transmission media. The software must be partitioned into parts for streaming. We call the process of modifying code for streaming software *streaming code generation*. We perform this modification after normal compilation. After modification, the application is ready to be executed after the first executable software unit is loaded into the memory of the device. In contrast to downloading the whole program, software streaming can improve application load time. While the application is running, additional parts of the stream-enabled software can be downloaded in the background or on-demand. If the needed part is not in the memory, the needed part must be transmitted in order to continue executing the application. The application load time can be adjusted by varying the size of the first block while considering the time for downloading the next block. This can potentially avoid the suspension of the application due to block misses. The predictability of software execution once it starts can be improved by software profiling which determines the transmission order of the blocks.

In this paper, we present a new method for software streaming especially suitable for embedded applications. Using our method, software transmission can be completely transparent to the user. The software is executed as if it is local to the device. Our streaming method can also significantly reduce the application load time since the CPU can start executing the application prior to completing the download of the entire program.

This paper consists of five sections. The first section introduces the motivation of the paper. The second section describes related work in the area of software streaming. In the third section, our method for streaming real-time embedded software is presented. We provide performance analysis of the streaming environment in the fourth section. Finally, in the fifth section, we present an example application and results.

## 2. RELATED WORK

In a networking environment, a client device can request certain software from the server. A typical process involves downloading, decompressing, installing and configuring the software before the CPU can start executing the program. For a large program, download time can be very long. Typically, application load time refers to the time between when an application is selected to run and when the first instruction of the software is executed. Transmission time of the software predominantly contributes to the application load time. Hence, a long download time is equivalent to a long application load time. A long application load time experienced by the user is an undesirable effect of loading the application from the network.

In Java applet implementation, the typical process of downloading the entire program is eliminated. A Java applet can be run without obtaining all of the

classes used by the applet. Java class files can be downloaded on-demand from the server. If a Java class is not available to the Java Virtual Machine (JVM) when an executing applet attempts to invoke the class functionality, the JVM may dynamically retrieve the class file from the server [2, 3]. In theory, this method may work well for small classes. The application load time should be reduced, and the user should be able to interact with the application rather quickly. In practice, however, Web browsers make quite a few connections to retrieve class files. HTTP/1.0, which is used by most Web servers [4], allows one request (e.g., for a class file) per connection. Therefore, if many class files are needed, many requests must be made, resulting in large communication overhead. The number of requests (thus, connections) made can be reduced by bundling and compressing class files into one file [5], which in turn unfortunately can increase the application load time. While the transition to persistent connections in HTTP/1.1 may improve the performance for the applet having many class files by allowing requests and responses to be pipelined [6], the server does not send the subsequent java class files without a request from the client. The JVM does not request class files not yet referenced by a class. Therefore, when a class is missing, the Java applet must be suspended. For a complex application, the class size may be large, which requires a long download time. As a result, the application load time is also long, a problem avoided by the block streaming method in which the application can start running as soon as the first executable block is loaded into memory (the next section will describe block streaming in detail).

Huneycutt et al. [7] propose a method to solve an extreme case of limited resource issues for networked embedded devices such as distributed sensors and cell phones by implementing software caching. In this system, the memory on the device acts as cache for a more powerful server. The server sends a section of instructions or data to the device. When the device requires code outside the section, a software cache miss occurs. The server is then requested to send the needed code. Dynamic binary rewriting is required to properly supply the code to the device which incurs time overheads of at least 19% compared to no caching [7]. The program will also suffer from suspension delay if the software cache miss constantly occurs. Our method allows software code to be streamed in the background which may reduce software suspension due to code misses.

Raz, Volk and Melamed [8] describe a method to solve long load-time delays by dividing the application software into a set of modules. For example, a Java applet is composed of Java classes. Once the initial module is loaded, the application can be executed while additional modules are streamed in the background. The transmission time is reduced by substituting various code procedures with shortened streaming stub procedures, which will be replaced once the module is downloaded. Since software delivery size varies from one module to another, predicting suspension time may be difficult. However, in block streaming, this issue (unpredictable suspension time) can be avoided by streaming fixed-size blocks.

Eylon et al. [9] describe a virtual file system installed in the client that is configured to appear to the operating system as a local storage device containing all of the application files to be streamed required by the application. The application files are broken up into pieces called streamlets. If the needed streamlet is not available at the client, a streamlet request is sent to the server and the virtual file system maintains a busy status until the necessary streamlets have been provided. In this system, overhead from a virtual file system may be too high for some embedded devices to support. Unlike the method in [9], our block streaming method does not need virtual file system support.

Software streaming can also be done at the source code level. The source code is transmitted to the embedded device and compiled at load time [10]. Although the source code is typically small compared to its compiled binary image and can be transferred faster, the compilation time may be very long and the compiler's memory usage for temporary files may be large. Since the source code is distributed, full load-time compilation also exposes the intellectual property contained in the source code being compiled [11]. Moreover, a compiler must reside in the client device at all times, which occupies a significant amount of storage space. This method may not be appropriate for a small memory footprint and slower or lower-power processor embedded device.

### 3. BLOCK STREAMING

In this section, we present software streaming via block streaming. The presented streaming method is lightweight in that it tries to minimize bandwidth overhead, which is a key issue for streaming embedded software. The major benefit, though, is dramatically reduced application load times. In our approach, the embedded application must be modified before it can be streamed to the embedded device. As shown in Figure 32-1, the block-streaming process on the server involves (i) compiling the application source

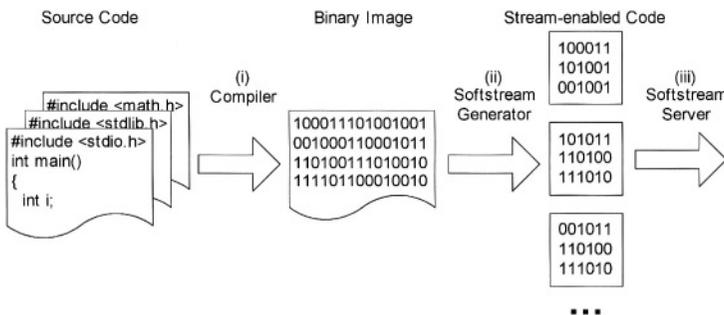


Figure 32-1. Server-side block-streaming process.

code into an executable binary image by a standard compiler such as GCC, (ii) generating a new binary for the stream-enabled application and (iii) transmitting the stream-enable application to the client device.

The process to receive a block of streamed software on the client device is illustrated in Figure 32-2: (i) load the block into memory and (ii) link the block into the existing code. This “linking” process may involve some code modification which will be described later.

We define a code block to be a contiguous address space of data or executable code or both. A block does not necessarily have a well-defined interface; for example, a block may not have a function call associated with the beginning and ending addresses of the block, but instead block boundaries may place assembly code for a particular function into multiple, separate blocks. The compiled application is considered as a binary image occupying memory. This binary image is divided into blocks before the stream-enabled code is generated. See Section 3 of [12] more details.

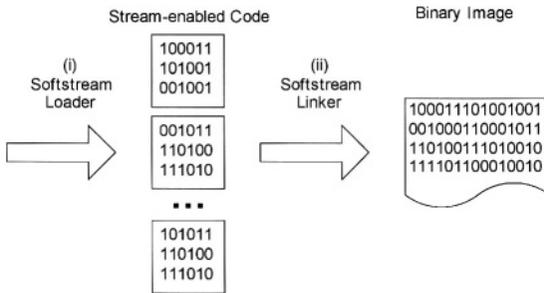


Figure 32-2. Client-side block-streaming process

### 3.1. Embedded software streaming code generation

In our approach, the stream-enabled embedded software is generated from the executable binary image of the software. The executable binary image of the software is created from a standard compiler such as GCC.

#### 3.1.1. Preventing the execution of non-existing code

After the application is divided into blocks, exiting and entering a block can occur in four ways. First, a branch or jump instruction can cause the processor to execute code in another block. Second, when the last instruction of the block is executed, the next instruction can be the first instruction of the following block. Third, when a return instruction is executed, the next instruction would be the instruction after calling the current block subroutine, which could be in another block. Fourth, when an exception instruction is executed, the exception code may be in a different block. We call a branch instruction that may cause the CPU to execute an instruction in a different block an *off-*

*block* branch. All off-block branches to blocks not yet loaded must be modified for streaming to work properly. (Clearly, the reader may infer that our approach involves a form of binary rewriting.)

For the first case, a branch instruction which can potentially cause the processor to execute code in another block is modified to load the block containing the needed code as illustrated in Example 32-1. For the second case, suppose the last instruction in the block is not a goto or a return instruction and that the following block has not yet been loaded into memory. If left this way, the processor might execute the next instruction. To prevent the processor from executing invalid code, an instruction is appended by default to load the next block; an example of this is shown at the end of Example 32-1. For the third case, no return instruction ever needs to be modified if blocks are not allowed to be removed from memory since the instruction after the caller instruction is always valid (even if the caller instruction is the last instruction in the block upon initial block creation, an instruction will be appended according to the second case above). However, if blocks are allowed to be removed, all return instructions returning to another block must be modified to load the other block (i.e., the caller block) if needed. For the last case mentioned in the previous paragraph, all exception code must be streamed before executing the block containing the corresponding exception instruction. Therefore, exception instructions are not modified.

**EXAMPLE 32-1:** Consider the *if-else* C statement in Figure 32-3. The C statement is compiled into corresponding PowerPC assembly instructions. The application can be divided so that the statement is split into different blocks. Figure 32-3 shows a possible split.

In this small example, the first block (Block 1) contains two branch instructions, each of which could potentially jump to the second block. If the second block (Block 2) is not in memory, this application will not work properly. Therefore, these branch instructions must be modified. All off-block branches are modified to invoke the appropriate loader function if the block is not in memory. After the missing block is loaded, the intended location of the original branch is taken. Figure 32-4 shows Block 1 and Block 2 from Figure 32-3

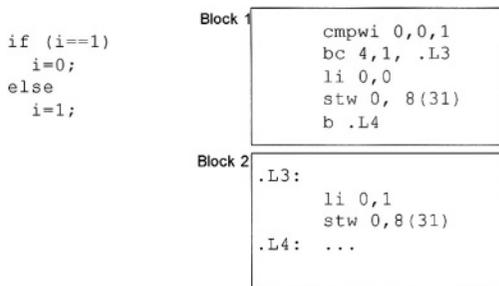


Figure 32-3. C code and corresponding PowerPC assembly.

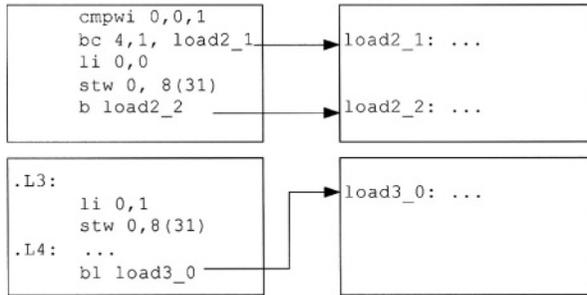


Figure 32-4. Block 1 and Block 2 after the stream-enabled code generation.

after running the software streaming code generation program on the application code. Branch instructions `bc 4,1,L3` and `b .L4`, as seen in Figure 32-3, are modified to `bc 4,1,load2_1` and `b load2_2`, respectively, as seen in Figure 32-4.

Since the last instruction of Block 2 is not a branch or a return instruction, the instruction `bl load3_0` will be appended to Block 2 from Figure 32-3 to load the subsequent block. The instruction `bl load3_0` can be replaced later by the first instruction of the block after Block 2 in order to preserve code continuity. ■

### 3.1.2. Coping with non-interruptible sections

Some real-time applications may contain non-interruptible section(s). This means that while the processor is executing a non-interruptible section, the processor cannot accept interrupts. Usually, a non-interruptible section contains instruction(s) to disable interrupts and one or more instructions to enable interrupts. A real-time code section should not be suspended due to missing software components since downloading software from the network can be very long and unpredictable. Therefore, a non-interruptible real-time section must be put in a single block.

### 3.1.3. Enforcing block boundaries

One drawback for randomly dividing a binary image into blocks is that some of the code in a particular block may not be used. For instance, consider the case where the first instruction of a block is the last instruction of the function in the previous block. For this case, perhaps only one instruction of the entire block (the first instruction) may be needed if the other function(s) in the block are never called. As a result, memory and bandwidth are not efficiently used. However, by obtaining the size of each function, the block boundaries can be enforced to more closely match with function boundaries [12].

### 3.2. Runtime code modification

Recall that, as described in Section 3.1.1, all off-block branch instructions are modified to call appropriate loader functions before proceeding to the target. After a particular loader function finishes making sure that no more block loading is required, the corresponding off-block branch is modified to jump to the target location instead of invoking the loader function the next time the off-block branch is re-executed. Although runtime code modification (runtime binary rewriting) introduces some overhead, the application will run more efficiently if the modified branch instruction is executed frequently. This is because, after modification, there is no check as to whether or not the block is in memory, but instead the modified branch instruction branches to the exact appropriate code location.

*EXAMPLE 32-2:* Suppose that the software from Figure 32-3 is modified using the software streaming code generated as illustrated in Figure 32-4 and is running on an embedded device. When an off-block branch is executed and taken, the missing block must be made available to the application. Figure 32-5 shows the result after the branch to load Block 2 is replaced with a branch to location `.L3` in Block 2 (look at the second instruction in the top left box Figure 32-5).

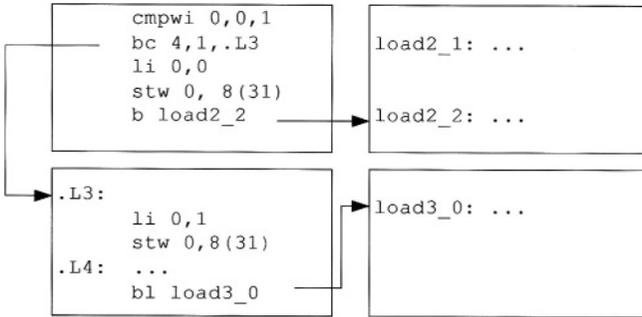


Figure 32-5. Runtime code modification.

The runtime code modification only changes the instruction which issues the request. Other branches (such as `b load2_2` in Figure 32-5) remain untouched even if the corresponding block is already in memory. If such a branch instruction is executed to load a block already in memory, the branch instruction will be modified at that time. ■

## 4. PERFORMANCE ANALYSIS

In order to take advantage of software streaming, the streaming environment must be thoroughly analyzed. The streaming environment analysis will likely

include CPU speed, connection speed, block size and the program execution path. Without knowledge about the environment for software streaming, the performance of the system can be suboptimal. For instance, if the block size is too small and the processor can finish executing the first block faster than the transmission time of the second block, the application must be suspended until the next block is loaded. This would not perform well in an interactive application. Increasing the block size of the first block will delay the initial program execution, but the application may run more smoothly.

#### **4.1. Performance metrics**

##### *4.1.1. Overheads*

The most obvious overhead is the code added during the stream-enabled code generation step for block streaming. For the current implementation, each off-block branch adds 12 bytes of extra code to the original code. The stream-enabling code (added code) for off-block branches consists of (i) the ID of the needed block, (ii) the original instruction of the branch, and (iii) the offset of the instruction. Since each field (i–iii) occupies four bytes, the total is 12 extra bytes added. The original code and the stream-enabling code are put together into a streamed unit. A streamed unit must be loaded before the CPU can safely execute the code. Therefore, the added stream-enabling code incurs both transmission and memory overheads. Increasing the block size may reduce these overheads. However, a larger block size increases the application load time since the larger block takes longer to be transmitted.

Runtime overheads are associated with code checking, code loading and code modification. Code loading occurs when the code is not in the memory. Code checking and code modification occur when an off-block branch is first taken. Therefore, these overheads from the runtime code modifications eventually diminish.

##### *4.1.2. Application load time*

Application load time is the time between when the program is selected to run and when the CPU executes the first instruction of the selected program. The application load time is directly proportional to block size and is inversely proportional to the speed of the transmission media. The program can start running earlier if the application load time is lower. The application load time can be estimated as explained in detail in [12].

##### *4.1.3. Application suspension time*

Application suspension occurs when the next instruction that would be executed in normal program execution is in a block yet to be loaded or only partially loaded into memory. The application must be suspended while the

required block is being streamed. The worst case suspension time occurs when the block is not in memory; in this case the suspension time is the time to load the entire block which includes the transmission time and processing time of the streamed block. The best case occurs when the block is already in memory. Therefore, the suspension time is between zero and the time to load the whole block. Application suspension time is proportional to the streamed block size. The application developer can vary the block size to obtain an acceptable application suspension time if a block miss occurs while avoiding multiple block misses.

For applications involving many interactions, low suspension delay is very crucial. While the application is suspended, it cannot interact with the environment or the user. Response time can be used as a guideline for application suspension time since the application should not be suspended longer than response time. A response time which is less than 0.1 seconds after the action is considered to be almost instantaneous for user interactive applications [13]. Therefore, if the application suspension time is less than 0.1 seconds, the performance of the stream-enabled application should be acceptable for most user interactive applications when block misses occur.

## **4.2. Performance enhancements**

### *4.2.1. Background streaming*

In some scenarios, it may be a better solution if the application can run without interruption due to missing code. By allowing the components or blocks to be transmitted in the background (background streaming) while the application is running, needed code may appear in memory prior to being needed. As a result, execution delay due to code misses is reduced. The background streaming process is only responsible for downloading the blocks and does not modify any code.

### *4.2.2. On-demand streaming*

In some cases where bandwidth and memory are scarce resources, background streaming may not be suitable. Certain code blocks may not be needed by the application in a certain mode. For example, the application might use only one filter in a certain mode of operation. Memory usage is minimized when a code block is downloaded on-demand, i.e., only when the application needs to use the code block. While downloading the requested code, the application will be suspended. In a multitasking environment, the block request system call will put the current task in the I/O wait queue and the RTOS may switch to run other ready tasks. On-demand streaming allows the user to trade off between resources and response times

#### 4.2.3. Software profiling

It is extremely unlikely that the application executes its code in a linear address ordering. Therefore, the blocks should preferably be streamed in the most common order of code execution. This can minimize delays due to missing code. Software profiling can help determine the order of code to be streamed in the background and on-demand. When the software execution path is different from the predicted path, the order of background streaming must be changed to reflect the new path. A software execution path can be viewed as a control/data flow graph. When the program execution flows along a certain path, the streaming will be conducted accordingly. A path prediction algorithm is necessary for background streaming to minimize software misses. The block size for the paths which are unlikely to be taken can be determined according to the appropriate suspension delay. Currently, we manually predict the software execution path.

## 5. SIMULATION RESULTS

We implemented a tool for breaking up executable binary images in PowerPC assembly into blocks and generating corresponding streamed code ready to be streamed to the embedded device. We simulated our streaming methods using hardware/software co-simulation tools which consist of Synopsys<sup>®</sup> VCS<sup>™</sup>, Mentor Graphics<sup>®</sup> Seamless CVE<sup>™</sup>, and Mentor Graphics XRAY<sup>®</sup> Debugger. Among the variety of processor instruction-set simulators (ISSes) available in Seamless CVE<sup>™</sup>, we chose to use an ISS for the PowerPC 750 architecture. As illustrated in Figure 32-6, the simulation environment is a shared-memory system. Each processor runs at 400 MHz while the bus runs at 83 MHz. A stream-enabled application runs on the main processor. The I/O processor is for transferring data. We validated our streaming method in this simulation environment.

In robot exploration, it is impossible to write and load software for all possible environments that the drone will encounter. The programmer may want to be able to load some code for the robot to navigate through one type of terrain and conduct an experiment. As the robot explores, it may roam to another type of terrain. The behavior of the robot could be changed by newly

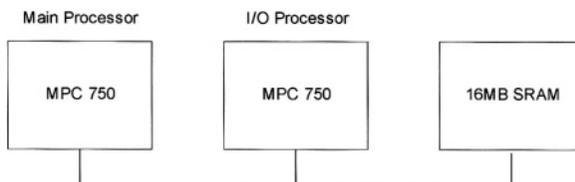


Figure 32-6. Simulation environment.

downloaded code for the new environment. Furthermore, the remote robot monitor may want to conduct a different type of experiment which may not be programmed into the original code. The exploration would be more flexible if the software can be sent from the base station. When the robot encounters a new environment, it can download code to modify the robot's real-time behavior. The new code can be dynamically incorporated without reinitializing all functionality of the robot.

In this application, we used the block streaming method to transmit the software to the robot. A binary application code of size 10 MB was generated. The stream-enabled application was generated using our softstream code generation tool. There were three off-block branch instructions on average in each block. The software was streamed over a 128 Kbps transmission media. Table 32-1 shows average overheads of added code per block and load time for different block sizes. The average added code per block is 36 bytes (due an average in each block of three off-blocks branches each of which adds 12 bytes). This overhead is insignificant (less than 0.5%) for block sizes larger than 1 KB. The load times were calculated using only transmission of the block and the streamed code. We did not include other overhead such as propagation delay from the server to the client and processing.

Without using the block streaming method, the application load time of this application would be over 10 minutes (approximately 655 seconds). If the robot has to adapt to the new environment within 120 seconds, downloading the entire application would miss the deadline by more than eight minutes. However, if the application were broken up into 1 MB or smaller blocks, the deadline could be met. Even the strict deadline is not crucial, the block streaming method reduces the application load time by 10 $\times$  or more for the block sizes of 1 MB or less. The load time for a block size of 1 MB is approximately 65 seconds whereas just streaming the entire block as on 10 MB whole results in an application load time of more than 655 seconds.

If the software profiling predicts the software execution path correctly, the application will run without interruption due to missing blocks; the subsequently-needed blocks can be streaming in the background while the CPU is

Table 32-1. Simulation result for block streaming of robot code.

Block size (bytes)	Number of blocks	Added code/block	Load time (s)
10 M	1	0.0003%	655.36
5 M	2	0.0007%	327.68
2 M	5	0.0017%	131.07
1 M	10	0.0034%	65.54
0.5 M	20	0.0069%	32.77
100 K	103	0.0352%	6.40
10 K	1024	0.3516%	0.64
1 K	10240	3.5156%	0.06
512	20480	7.0313%	0.03

executing the first block. However, if the needed block is available at the client, the application must be suspended until the needed block is downloaded. If the size of the missing block is 1 KB, the suspension time is only 0.06 seconds. As mentioned previously, this suspension delay of less than 0.1 seconds is considered to be almost instantaneous for user interactive applications. While our sample application is not a full industrial-strength example, it does verify the block streaming functionality and provides experimental data.

## 6. CONCLUSION

Embedded software streaming allows an embedded device to start executing an application while the application is being transmitted. We presented a method for transmitting embedded software from a server to be executed on a client device. Our streaming method can lower application load time, bandwidth utilization and memory usages. We verified our streaming method using a hardware/software co-simulation platform for the PowerPC architecture, specifically for MPC 750 processors. Our current approach assumes that no function pointers are used. For our future work, we plan to remove this restriction.

Advantageously, software streaming enables client devices, especially embedded devices, to potentially support a wider range of applications by efficiently utilizing resources. The user will experience a relatively shorter application load time. Additionally, this method facilitates software distribution and software updates since software is directly streamed from the server. In case of a bug fix, the software developer can apply a patch at the server. The up-to-date version of the software is always streamed to the client device. Finally, software streaming has the potential to dramatically alter the way software is executed in the embedded setting where minimal application load time is important to clients.

## ACKNOWLEDGEMENTS

This research is funded by the State of Georgia under the Yamacraw initiative [14]. We acknowledge donations received from Denali, Hewlett-Packard Company, Intel Corporation, LEDA, Mentor Graphics Corp. [15], SUN Microsystems and Synopsys, Inc. [16].

## REFERENCES

1. R. Avner. "Playing GoD: Revolution for the PC with Games-on-Demand." *Extent Technologies*, <http://www.gamesbiz.net/keynotes-details.asp?Article=248>.

2. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*, 2nd ed., Massachusetts: Addison-Wesley Publishing Company, 1999, pp. 158-161.
3. J. Meyer and T. Downing. *Java Virtual Machine*, California: O'Reilly & Associates, Inc., 1997, pp. 44-45.
4. E. Nahum, T. Barzilai and D. D. Kandlur. "Performance Issues in WWW Servers." *IEEE/ACM Transactions on Networking*, Vol. 10, No. 1, pp. 2-11.
5. P. S. Wang. *Java with Object-Oriented Programming and World Wide Web Applications*, California: PWS Publishing, 1999, pp. 193-194.
6. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. "Hypertext Transport Protocol - HTTP/1.1", RFC 2616, The Internet Engineering Task Force, June 1999, <http://www.ietf.org/rfc/rfc2616.txt?number=2616>.
7. M. H. Huneycutt, J. B. Fryman, and K. M. Mackenzie. "Software Caching using Dynamic Binary Rewriting for Embedded Devices." *Proceedings of International Conference on Parallel Processing*, 2002, pp. 621-630.
8. U. Raz, Y. Volk and S. Melamed. *Streaming Modules*, U.S. Patent 6,311,221, October 30, 2001.
9. D. Eylon, A. Ramon, Y. Volk, U. Raz and S. Melamed. *Method and System for Executing Network Streamed Application*, U.S. Patent Application 20010034736, October 25, 2001.
10. G. Eisenhauer, F. Buslament, and K. Schwan. "A Middleware Toolkit for Client-Initiate Service Specialization." *Operating Systems Review*, Vol. 35, No. 2, 2001, pp. 7-20.
11. M. Franx. "Dynamic Linking of Software Components." *Computer*, Vol. 30, pp. 74-81, March 1997.
12. P. Kuacharoen, V. Mooney and V. Madiseti. "Software Streaming via Block Streaming." Georgia Institute of Technology, Atlanta, Georgia, Tech. Rep. GIT-CC-02-63, 2002, [http://www.cc.gatech.edu/tech\\_reports](http://www.cc.gatech.edu/tech_reports).
13. W. Stallings, *Operating Systems*, 2nd ed., New Jersey: Prentice Hall, 1995, pp. 378-379.
14. Yamacraw, <http://www.yamacraw.org>.
15. Synopsys Inc., <http://www.synopsys.com>.
16. Mentor Graphics Corp., <http://www.mentor.com>.

# ADAPTIVE CHECKPOINTING WITH DYNAMIC VOLTAGE SCALING IN EMBEDDED REAL-TIME SYSTEMS

Ying Zhang and Krishnendu Chakrabarty

*Department of Electrical & Computer Engineering, Duke University, Durham, NC 27708, USA*

**Abstract.** We present an integrated approach that provides fault tolerance and dynamic power management for a real-time task executing in an embedded system. Fault tolerance is achieved through an adaptive checkpointing scheme that dynamically adjusts the checkpointing interval during task execution. Adaptive checkpointing is then combined with a dynamic voltage scaling scheme to achieve power reduction. Finally, we develop an adaptive checkpointing scheme for a set of multiple tasks in real-time systems. Simulation results show that compared to previous methods, the proposed approach significantly reduces power consumption and increases the likelihood of timely task completion in the presence of faults.

**Key words:** dynamic voltage scaling, fault tolerance, low power, on-line checkpointing

## 1. INTRODUCTION

Embedded systems often operate in harsh environmental conditions that necessitate the use of fault-tolerant computing techniques to ensure dependability. These systems are also severely energy-constrained since system lifetime is determined to a large extent by the battery lifetime. In addition, many embedded systems execute real-time applications that require strict adherence to task deadlines [1].

Dynamic voltage scaling (DVS) has emerged as a popular solution to the problem of reducing power consumption during system operation [2–4]. Many embedded processors are now equipped with the ability to dynamically scale the operating voltage. Fault tolerance is typically achieved in real-time systems through on-line fault detection [5], checkpointing and rollback recovery [6]; see Figure 33-1. At each checkpoint, the system saves its state in a secure device. When a fault is detected, the system rolls back to the most recent checkpoint and resumes normal execution.

Checkpointing increases task execution time and in the absence of faults, it might cause a missed deadline for a task that completes on time without checkpointing. In the presence of faults however, checkpointing can increase the likelihood of a task completing on time with the correct result. Therefore, the checkpointing interval, i.e., duration between two consecutive checkpoints,

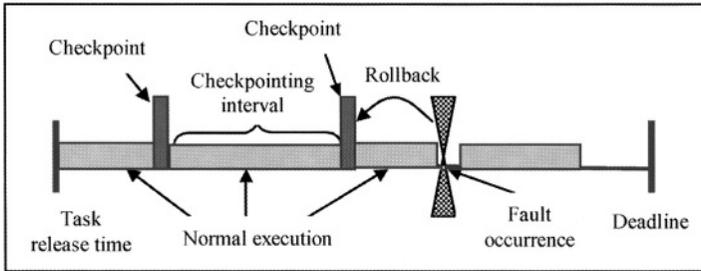


Figure 33-1. Checkpointing and rollback recovery.

must be carefully chosen to balance checkpointing cost (the time needed to perform a single checkpoint) with the rollback time.

Dynamic power management and fault tolerance for embedded real-time systems have been studied as separate problems in the literature. DVS techniques for power management do not consider fault tolerance [2–4], and checkpoint placement strategies for fault tolerance do not address dynamic power management [7–9]. We present an integrated approach that facilitates fault tolerance through checkpointing and power management through DVS. The main contributions of this chapter are as follows.

- We introduce an adaptive checkpointing scheme that dynamically adjusts the checkpointing interval during task execution, based on the frequency of faults and the amount of time remaining before the task deadline.
- The proposed adaptive checkpointing scheme is tailored to handle not only a random fault-arrival process, but it is also designed to tolerate up to  $k$  fault occurrences.
- Adaptive checkpointing is then combined with a DVS scheme to achieve power reduction and fault tolerance simultaneously.
- Adaptive checkpointing is extended to a set of multiple real time tasks.

We assume throughout that faults are intermittent or transient in nature, and that permanent faults are handled through manufacturing testing or field-testing techniques [10]. We also assume a “soft” real-time system in which even though it is important to meet task deadlines, missed deadlines do not lead to catastrophic consequences [11].

## 2. CHECKPOINTING IN REAL-TIME SYSTEMS

In this section, we present a classification of checkpointing schemes for real-time systems that have been presented in the literature.

### 2.1. On-line scheme versus off-line schemes

An off-line checkpointing scheme determines the checkpointing interval for a task *a priori*, i.e., before task execution. Most known checkpointing schemes for real-time systems belong to this category [9, 12, 13]. A drawback here is that the checkpointing interval cannot be adapted to the actual fault occurrence during task execution. An on-line scheme in which the checkpointing interval can be adapted to fault occurrences is therefore more desirable. However, current on-line checkpointing schemes [8] provide only probabilistic guarantees on the timely completion of tasks.

### 2.2. Probabilistic versus deterministic guarantees

Some checkpointing schemes, e.g. [9, 12], assume that faults occur as a Poisson process with arrival rate  $\lambda$ . These schemes use a checkpointing interval that maximizes the probability that a task completes on time for a given fault arrival rate  $\lambda$ . Hence the real-time guarantees in these schemes are probabilistic. Other checkpointing schemes, e.g., [13], offer deterministic real-time guarantees under the condition that at most  $k$  faults occur during task execution. A drawback of these  $k$ -fault-tolerant schemes is that they cannot adapt to actual fault occurrences during task execution.

### 2.3. Equidistant versus variable checkpointing interval

Equidistant checkpointing, as the term implies, relies on the use of a constant checkpointing interval during task execution. It has been shown in the literature that if the checkpointing cost is  $C$  and faults arrive as a Poisson process with rate  $\lambda$ , the mean execution time for the task is minimum if a constant checkpointing interval of  $\sqrt{2C/\lambda}$  is used [12]. We refer to this as the Poisson-arrival approach. However, the minimum execution time does not guarantee timely completion of a task under real-time deadlines. It has also been shown that if the fault-free execution time for a task is  $E$ , the worst-case execution time for up to  $k$  faults is minimum if the constant checkpointing interval is set to  $\sqrt{EC/k}$  [13]. We refer to this as the  $k$ -fault-tolerant approach. A drawback with these equidistant schemes is that they cannot adapt to actual fault arrivals. For example, due to the random nature of fault occurrences, the checkpointing interval can conceivably be increased halfway through task execution if only a few faults occur during the first half of task execution. Another drawback of equidistant checkpointing schemes is that they do not exploit the advantages offered by DVS for dynamic power management. For these reasons, we consider on-line checkpointing with variable checkpointing intervals.

## 2.4. Constant versus variable checkpointing cost

Most prior work has been based on the assumption that all checkpoints take the same amount of time, i.e., the checkpointing cost is constant. An alternative approach, taken in [8], but less well understood, is to assume that the checkpointing cost depends on the time at which it is taken. We use the constant checkpointing cost model in our work because of its inherent simplicity.

Our goal in this chapter is to develop a two-priority energy-aware checkpointing scheme for real-time systems. The first priority is to meet the real-time deadline under faulty conditions. The second priority is to save energy consumption for real-time execution in faulty conditions.

## 3. ADAPTIVE CHECKPOINTING

We are given the following parameters for a real-time task: deadline  $D$ ; execution time  $E$  when there are no fault in the system ( $E < D$ ); an upper limit  $k$  on the number of fault occurrences that must be tolerated; checkpointing cost  $C$ . We make the following assumptions related to task execution and fault arrivals:

- Faults arrive as a Poisson process with rate  $\lambda$ .
- The task starts execution at time  $t = 0$ .
- The times for rollback and state restoration are zero.
- Faults are detected as soon as they occur, and no faults occur during checkpointing and rollback recovery.

We next determine the maximum value of  $E$  for the Poisson-arrival and the  $k$ -fault-tolerant schemes beyond which these schemes will always miss the task deadline. Our proposed adaptive checkpointing scheme is more likely to meet the task deadline even when  $E$  exceeds these threshold values. If the Poisson-arrival scheme is used, the effective task execution time in the absence of faults must be less than the deadline  $D$  if the probability of timely completion of the task in the presence of faults is to be nonzero. This implies that  $E + (E/(\sqrt{2C/\lambda}) - 1)C \leq D$ , from which we get the threshold:

$$E_{\lambda th} = \frac{D + C}{1 + \sqrt{\lambda C/2}} \quad (1)$$

Here  $(E/(\sqrt{2C/\lambda}) - 1)$  refers to the number of checkpoints. The re-execution time due to rollback is not included in the formula for  $E_{\lambda th}$ . If  $E$  exceeds  $E_{\lambda th}$  for the Poisson-arrival approach, the probability of timely completion of the task is simply zero. Therefore, beyond this threshold, the checkpointing interval must be set by exploiting the slack time instead of utilizing the optimum checkpointing interval for the Poisson-arrival approach. The checkpointing interval  $I_m$  that barely allows timely completion in the fault-free case

is given by  $E + (E/I_m - 1)C = D$ , from which it follows that  $I_m = EC/(D + C - E)$ . To decrease the checkpointing cost, we set the checkpointing interval to  $2I_m$  in our adaptive scheme.

A similar threshold on the execution time can easily be calculated for the  $k$ -fault-tolerant scheme. In order to satisfy the  $k$ -fault-tolerant requirement, the worst-case re-execution time is incorporated. The following inequality must hold:  $E + (E/(\sqrt{EC/k}) - 1)C + k\sqrt{EC/k} \leq D$ . This implies the following threshold on  $E$ :

$$E_{kth} = [(D + C) + 2kC] - 2\sqrt{kC(D + C) + (kC)^2} \quad (2)$$

If the execution time  $E$  exceeds  $E_{kth}$ , the  $k$ -fault-tolerant checkpointing scheme cannot provide a deterministic guarantee to tolerate  $k$  faults.

### 3.1. Checkpointing algorithm

The adaptive checkpointing algorithm attempts to maximize the probability that the task completes before its deadline despite the arrival of faults as a Poisson process with rate  $\lambda$ . A secondary goal is to tolerate, as far as possible, up to  $k$  faults. In this way, the algorithm accommodates a pre-defined fault-tolerance requirement (handle up to  $k$  faults) as well as dynamic fault arrivals modeled by the Poisson process. We list below some notation that we use in our description of the algorithm:

1.  $I_1(C, \lambda) = \sqrt{2C/\lambda}$  denotes the checkpointing interval for the Poisson-arrival approach.
2.  $I_2(E, k, C) = \sqrt{EC/k}$  denotes the checkpointing interval for the  $k$ -fault-tolerant approach.
3.  $I_3(E, D, C) = 2I_m = 2EC/(D + C - E)$  denotes the checkpointing interval if the Poisson-arrival approach is not feasible for timely task completion.
4.  $R_t$  denotes the remaining execution time. It is obtained by subtracting from  $E$  the amount of time the task has executed (not including checkpointing and recovery).
5.  $R_d$  denotes the time left before the deadline. It is obtained by subtracting the current time from  $D$ .
6.  $R_f$  denotes an upper bound on the remaining number of faults that must be tolerated.
7. The threshold  $T_{rk}(R_d, \lambda, C)$  is obtained by replacing  $D$  with  $R_d$  in (1).
8. The threshold  $T_{rk}(R_d, R_f, C)$  is obtained by replacing  $D$  with  $R_d$  and  $k$  with  $R_f$  in (2).

The procedure  $interval(R_d, R_t, C, R_f, \lambda)$  for calculating the checkpointing interval is described in Figure 33-2(a), and the adaptive checkpointing scheme  $adaptchp(D, E, C, k, \lambda)$  is described in Figure 33-2(b). The adaptive checkpointing procedure is event-driven and the checkpointing interval is adjusted when a fault occurs and rollback recovery is performed.

<p><b>Procedure</b> <i>interval</i>(<math>R_d, R_b, C, R_f, \lambda</math>)</p> <ol style="list-style-type: none"> <li>1. <math>Exp\_fault = \lambda R_b</math>;</li> <li>2. <b>if</b> (<math>Exp\_fault \leq R_f</math>) {</li> <li>3.     <b>if</b> (<math>R_t &gt; Th_\lambda(R_d, \lambda, C)</math>) <b>then</b></li> <li>        <math>chk\_interval = I_3(R_b, R_d, C)</math>;</li> <li>4.     <b>else if</b> (<math>R_t &gt; Th_\lambda(R_d, R_f, C)</math>) <b>then</b></li> <li>        <math>chk\_interval = I_2(R_b, Exp\_fault, C)</math>;</li> <li>5.     <b>else</b> <math>chk\_interval = I_2(R_b, R_f, C)</math>;</li> <li>6. <b>else</b> {<b>if</b> (<math>R_t &gt; Th_\lambda(R_d, \lambda, C)</math>) <b>then</b></li> <li>        <math>chk\_interval = I_3(R_b, R_d, C)</math>;</li> <li>7.     <b>else</b> <math>chk\_interval = I_1(C, \lambda)</math>;</li> <li>8. <b>return</b> <math>chk\_interval</math>;</li> </ol> <p style="text-align: center;">(a)</p>	<p><b>Procedure</b> <i>adaptchk</i>(<math>D, E, C, k, \lambda</math>)</p> <ol style="list-style-type: none"> <li>1. <math>R_t = E</math>; <math>R_d = D</math>; <math>R_f = k</math>; <math>Itv = interval(R_d, R_b, C, R_f, \lambda)</math>;</li> <li>2. <b>while</b> (<math>R_t &gt; 0</math>) <b>do</b> {</li> <li>3.     <b>if</b> (<math>R_t &gt; R_d</math>) <b>break</b>;</li> <li>4.     Case 1: During normal execution, <b>do</b> {</li> <li>        4.1 Insert checkpoints with interval length <math>Itv</math>;</li> <li>        4.2 Update <math>R_b, R_f</math>;</li> <li>5.     Case 2: Upon fault occurrence, <b>do</b> {</li> <li>        5.1 Roll back and restore status;</li> <li>        5.2 <math>R_f = R_f - 1</math>;</li> <li>        5.3 <math>Itv = interval(R_d, R_b, C, R_f, \lambda)</math>;</li> <li>        5.4 Resume execution;}</li> </ol> <p style="text-align: center;">(b)</p>
---	---

Figure 33-2. (a) Procedure for calculating the checkpointing interval. (b) Adaptive checkpointing procedure.

### 3.2. Simulation results on adaptive checkpointing

We carried out a set of simulation experiments to evaluate the adaptive checkpointing scheme (referred to as ADT) and to compare it with the Poisson-arrival and the  $k$ -fault-tolerant checkpointing schemes. Faults are injected into the system using a Poisson process with various values for the arrival rate  $\lambda$ . Due to the stochastic nature of the fault arrival process, the experiment is repeated 10,000 times for the same task and the results are averaged over these runs. We are interested here in the probability  $P$  that the task completes on time, i.e., either on or before the stipulated deadline. As in [11], we use the term task utilization  $U$  to refer to the ratio  $E/D$ .

For  $\lambda < 0.002$  and  $U < 0.7$  (low fault arrival rate and low task utilization), the performances of the three schemes, measured by the probability of timely completion of the task, are comparable. For  $\lambda > 0.002$  and  $U > 0.7$ , the adaptive checkpointing scheme clearly outperforms the other two schemes; the results are shown in Table 33-1. The value of  $P$  is as much as 30% higher for the ADT scheme. Note that even though the results are reported only for  $D = 10000$ ,  $C = 10$ , and  $k = 10$ , similar trends were observed for other values of  $D$ ,  $C$ , and  $k$ . For  $\lambda < 0.002$  and  $U \geq 0.9$ , the ADT scheme outperforms the other two schemes; see Table 33-2.

To further illustrate the advantage of the ADT scheme, we note that if we set  $U = 0.99$  and  $k = 1$  (using the values of  $D$  and  $C$  as before), the value of  $P$  drops to zero for both the Poisson-arrival and the  $k$ -fault-tolerant schemes if  $\lambda > 3 \times 10^{-5}$ . In contrast, the proposed ADT scheme continues to provide significant higher value of  $P$  as  $\lambda$  increases (Table 33-3).

For  $\lambda > 0.002$  and  $U \geq 0.9$ , the ADT scheme again outperforms the other two schemes. In conclusion, we note that the ADT scheme is more likely to

Table 33-1. (a) Variation of  $P$  with  $\lambda$  for  $D = 10000$ ,  $C = 10$ , and  $k = 10$ .

$U = E/D$	Fault arrival rate $\lambda (\times 10^{-2})$	Probability of timely completion of tasks, $P$		
		Poisson-arrival	$k$ -fault-tolerant	ADT
0.80	0.24	0.505	0.476	0.532
	0.28	0.229	0.243	0.273
0.82	0.24	0.204	0.168	0.235
	0.28	0.052	0.042	0.092

Table 33-1. (b) Variation of  $P$  with  $U$  for  $D = 10000$ ,  $C = 10$ , and  $k = 10$ .

$\lambda (\times 10^{-2})$	$U = E/D$	Probability of timely completion of tasks, $P$		
		Poisson-arrival	$k$ -fault-tolerant	ADT
0.26	0.76	0.887	0.888	0.909
	0.78	0.655	0.666	0.715
0.30	0.76	0.864	0.823	0.872
	0.78	0.589	0.597	0.626

meet task deadlines when the task utilization is high and the fault arrival rate is not very low.

Table 33-2. (a) Variation of  $P$  with  $\lambda$  for  $D = 10000$ ,  $C = 10$ , and  $k = 1$ .

$U = E/D$	Fault arrival rate $\lambda (\times 10^{-4})$	Probability of timely completion of tasks, $P$		
		Poisson-arrival	$k$ -fault-tolerant	ADT
0.92	1.0	0.902	0.945	0.947
	2.0	0.770	0.786	0.831
0.95	1.0	0.659	0.649	0.774
	2.0	0.372	0.387	0.513

Table 33-2. (b) Variation of  $P$  with  $U$  for  $D = 10000$ ,  $C = 10$ , and  $k = 1$ .

$\lambda (\times 10^{-4})$	$U = E/D$	Probability of timely completion of tasks, $P$		
		Poisson-arrival	$k$ -fault-tolerant	ADT
1.0	0.92	0.902	0.945	0.947
	0.94	0.747	0.818	0.852
2.0	0.92	0.770	0.786	0.831
	0.94	0.573	0.558	0.643

Table 33-3. Variation of  $P$  with  $\lambda$  for  $D = 10000$ ,  $C = 10$ , and  $k = 1$ .

$U = E/D$	Fault arrival rate $\lambda (\times 10^{-5})$	Probability of timely completion of tasks, $P$		
		Poisson-arrival	$k$ -fault-tolerant	ADT
0.99	1.0	0.893	0.000	0.907
	3.0	0.000	0.000	0.732
	5.0	0.000	0.000	0.515

#### 4. ADT-DVS: ADAPTIVE CHECKPOINTING WITH DVS

We next show how adaptive checkpointing scheme can be combined with DVS to obtain fault tolerance and power savings in real-time systems. We consider adaptive intra-task voltage scaling, wherein the processor speed is scaled up in response to a need for increased slack for checkpointing, and scaled down to save power if the slack for a lower speed is adequate. We consider a two-speed processor here, and for the sake of simplicity, we use the terms processor speed and processor frequency interchangeably.

We use the same notation as described in Section 3.1. In addition, we are given the following: (1) A single processor with two speeds  $f_1$  and  $f_2$ . Without loss of generality, we assume that  $f_2 = 2f_1$ ; (2) The processor can switch its speed in a negligible amount of time (relative to the task execution time); (3) The number of computation cycles  $N$  for the task in the fault-free condition.

We note that if supply voltage  $V_{dd}$  is used for a task with  $N$  single-cycle instructions, the energy consumption is proportional to  $NV_{dd}^2$ . We also note that the clock period is proportional to  $V_{dd}/(V_{dd} - V_t)^2$ , where  $V_t$  is the transistor threshold voltage. We assume here without loss of generality that  $V_t = 0.8$  V, and the supply voltage  $V_{dd1}$  corresponding to speed  $f_1$  is 2.0 V. Using the formula for the clock period, we find that the supply voltage  $V_{dd2}$  corresponding to speed  $f_2$  is 2.8 V.

Let  $R_c$  be the number of instructions of the task that remain to be executed at the time of the voltage scaling decision. Let  $c$  be the number of clock cycles that a single checkpoint takes. We first determine if processor frequency  $f$  can be used to complete the task before the deadline. As before, let  $R_d$  be the amount of time left before the task deadline. The checkpointing cost  $C$  at frequency  $f$  is given by:  $C = c/f$ . Let  $t_{est}$  be an estimate of the time that the task has to execute in the presence of faults and with checkpointing. The expected number of faults for the duration  $t_{est}$  is  $\lambda t_{est}$ . To ensure  $\lambda t_{est}$ -fault-tolerance during task execution, the checkpointing interval must be set to  $\sqrt{t_{est}C/(\lambda t_{est})} = \sqrt{C/\lambda} = \sqrt{c/(\lambda f)}$ . Now, the parameter  $t_{est}$  can be expressed as follows:

$$t_{est} = \frac{R_c}{f} + t_{est} \sqrt{\frac{c}{\lambda f}} + \frac{c}{f} \frac{R_c/f}{\sqrt{c/(\lambda f)}} \quad (3)$$

The first term on the right-hand side of (3) denotes the time for forward execution, the second term denotes the recovery cost for  $\lambda t_{est}$  faults, and the third term denotes the checkpointing cost. From (3), we get

$$t_{est} = \frac{R_c(1 + \sqrt{\lambda c/f})}{f(1 - \sqrt{\lambda c/f})}$$

We consider the voltage scaling (to frequency  $f$ ) to be feasible if  $t_{est} \leq R_d$ . This forms the basis of the energy-aware adaptive checkpointing procedure *adap\_dvs* described in Figure 33-3. At every DVS decision point, an attempt is made to run the task at the lowest-possible speed.

```

Procedure adap_dvs( $D, N, c, k, \lambda$ )
1.  $R_c = N$ ;  $R_d = D$ ;  $R_f = k$ ;
2. if ( $t_{est}(R_c, f_1) \leq R_d$ )  $f = f_1$ ; else  $f = f_2$ ;
3.  $Itv = interval(R_d, R_c/f, c/f, R_f, \lambda)$ ;
4. while ( $R_t > 0$ ) do {
5. if ( $R_t > R_d/f$ ) break;
6. Case 1: During normal execution, do {
6.1 Insert checkpoints with interval length  $Itv$ ;
6.2 Update  $R_c, R_d$  according to speed  $f$ ; }
7. Case 2: Upon fault occurrence, do {
7.1 Roll back and restore status;
7.2  $R_f = R_f - 1$ ;
7.3 if ( $t_{est}(R_c, f_1) \leq R_d$ )  $f = f_1$ ; else  $f = f_2$ ;
7.4  $Itv = interval(R_d, R_c/f, c/f, R_f, \lambda)$ ;
7.5 Resume execution; } }

```

Figure 33-3. Energy-aware adaptive checkpointing procedure.

#### 4.1. Simulation results on ADT\_DVS

We compare the adaptive DVS scheme, denoted by ADT\_DVS, with the Poisson-arrival and  $k$ -fault-tolerant schemes in terms of the probability of timely completion and energy consumption. We use the same experimental set-up as in Section 3.2. In addition, we consider the normalized frequency values  $f_1 = 1$  and  $f_2 = 2$ . First we assume that both the Poisson-arrival and the  $k$ -fault-tolerant schemes use the lower speed  $f_1$ . The task execution time at speed  $f_1$  is chosen to be less than  $D$ , i.e.,  $N/f_1 < D$ . The task utilization  $U$  in this case is simply  $N/(f_1 D)$ . Our experimental results are shown in Table 33-4. The ADT\_DVS scheme always leads to timely completion of the task by appropriately choosing segments of time when the higher frequency  $f_2$  is used. The other two schemes provide a rather low value for  $P$ , and for larger values of  $\lambda$  and  $U$ ,  $P$  drops to zero. The energy consumption for the ADT\_DVS scheme is slightly higher than that for the other two schemes; however, on average, the task runs at the lower speed  $f_1$  for as much as 90% of the time. The combination of adaptive checkpointing and DVS utilizes the slack

Table 33-4. (a) Variation of  $P$  with  $\lambda$  for  $D = 10000$ ,  $c = 10$ , and  $k = 2$ .

$U$	Fault arrival rate $\lambda (\times 10^{-4})$	Probability of timely completion of tasks, $P$		
		Poisson-arrival	$k$ -fault-tolerant	ADT_DVS
0.95	0.5	0.790	0.704	1.000
	1.0	0.648	0.508	1.000
	1.5	0.501	0.367	1.000
	2.0	0.385	0.244	1.000

Table 33-4. (b) Variation of  $P$  with  $U$  for  $D = 10000$ ,  $c = 10$ , and  $k = 2$ .

$\lambda (\times 10^{-4})$	$U$	Probability of timely completion of tasks, $P$		
		Poisson-arrival	$k$ -fault-tolerant	ADT_DVS
1.0	0.92	0.924	0.960	1.000
	0.96	0.549	0.000	1.000
	1.00	0.000	0.000	1.000
2.0	0.92	0.799	0.849	1.000
	0.96	0.229	0.000	1.000
	1.00	0.000	0.000	1.000

effectively and stretches the task completion time to as close to the deadline as possible.

Next we assume that both the Poisson-arrival and the  $k$ -fault-tolerant schemes use the higher speed  $f_2$ . The task execution time at speed  $f_2$  is chosen to be less than  $D$ , i.e.,  $N/f_2 < D$ , and the task utilization here is  $N/(f_2 D)$ . Table 33-5 shows that since even though ADT\_DVS uses both  $f_1$  and  $f_2$ , adaptive checkpointing allows it to provide a higher value for  $P$  than the other two methods that use only the higher speed  $f_2$ . The energy consumption for ADT\_DVS is up to 50% less than for the other two methods for low to moderate values of  $\lambda$  and  $U$ ; see Table 33-6. When either  $\lambda$  or  $U$  is high, the energy consumption of ADT\_DVS is comparable to that of the other two schemes. (Energy is measured by summing the product of the square of the

Table 33-5. Variation of  $P$  with  $\lambda$  for  $D = 10000$ ,  $c = 10$ , and  $k = 1$ .

$U$	Fault arrival rate $\lambda (\times 10^{-4})$	Probability of timely completion of tasks, $P$		
		Poisson-arrival	$k$ -fault-tolerant	ADT_DVS
0.95	0.8	0.898	0.939	0.965
	1.2	0.841	0.868	0.912
	1.6	0.754	0.785	0.871
	2.0	0.706	0.695	0.791

Table 33-6. (a) Variation of energy consumption with  $\lambda$  for  $D = 10000$ ,  $c = 10$ , and  $k = 10$ .

$U$	Fault arrival rate $\lambda (\times 10^{-4})$	Energy consumption		
		Poisson-arrival	$k$ -fault-tolerant	ADT_DVS
0.60	2.0	25067	26327	21568
	4.0	25574	26477	21642
	6.0	25915	26635	21714
	8.0	26277	26806	22611

Table 33-6. (b) Variation of energy consumption with  $U$  for  $D = 10000$ ,  $c = 10$ , and  $k = 10$ .

$\lambda (\times 10^{-4})$	$U$	Energy consumption		
		Poisson-arrival	$k$ -fault-tolerant	ADT_DVS
5.0	0.10	4295	4909	2508
	0.20	8567	9335	4791
	0.30	12862	13862	7026
	0.40	17138	17990	9223
	0.50	21474	22300	15333

voltage and the number of computation cycles over all the segments of the task.) This is expected, since ADT\_DVS attempts to meet the task deadline as the first priority and if either  $\lambda$  or  $U$  is high, ADT\_DVS seldom scales down the processor speed.

## 5. ADAPTIVE CHECKPOINTING FOR MULTIPLE TASKS

We are given a set  $\Gamma = \{\tau_1, \tau_2, \dots, \tau_n\}$  of  $n$  periodic tasks, where task  $\tau_i$  is modeled by a tuple  $\tau_i = \langle T_i, D_i, E_i \rangle$ ,  $T_i$  is the period of  $\tau_i$ ,  $D_i$  is the relative deadline ( $D_i \leq T_i$ ), and  $E_i$  is the computation time under fault-free conditions. We are also aiming to tolerate up to  $k$  faults for each task instance.

The task set is first scheduled off-line with a general scheduling algorithm scheme under fault-free conditions. Here we employ the earliest-deadline-first (EDF) algorithm [11]. A sequence of  $m$  jobs  $\Phi = \{\theta_1, \theta_2, \dots, \theta_m\}$  is obtained for each hyperperiod. We further denote each job  $\theta_i$ ,  $1 \leq i \leq m$ , as a tuple  $\theta_i = \langle a_i, b_i, c_i \rangle$ , where  $a_i$  is the starting time,  $b_i$  is the execution time and  $c_i$  is the deadline for  $\theta_i$ . All these parameters are known *a priori*. Note that  $a_i$  is the absolute time when  $\theta_i$  starts execution instead of the relative release time, execution time is equal to that for the corresponding task, and job deadline is equal to the task deadline plus the product of task period and corresponding number of periods. In addition, since we are employing EDF, we have  $c_1 \leq c_2 \leq \dots \leq c_m$ .

Based on the job set  $\Phi$ , we develop a checkpointing scheme that inserts

checkpoints to each job by exploiting the slacks in a hyperperiod. The key issue here is to determine an appropriate value for the deadline that can be provided as an input parameter to the *adapchp*( $D, E, C, k, \lambda$ ) procedure. This deadline must be no greater than the exact job deadline such that the timing constraint can be met, and it should also be no less than the job execution time such that time redundancy can be exploited for fault-tolerance. To obtain this parameter, we incorporate a pre-processing step immediately after the off-line EDF scheduling is carried out. The resulting values are then provided for subsequent on-line adaptive checkpointing procedure.

We denote the slack time for job  $\theta_i$ ,  $1 \leq i \leq m$ , as  $h_i$ . We also introduce the concept of checkpointing deadline  $v_i$ , which is the deadline parameter provided to the *adapchp*( $D, E, C, k, \lambda$ ) procedure. It is defined as the sum of job execution time  $b_i$  and slack time  $h_i$ . Furthermore, for the sake of convenience of problem formulation, we add a pseudo-job  $\theta_0$ , which has parameters  $a_0 = b_0 = c_0 = h_0 = 0$ .

### 5.1. Linear-programming model

Now we illustrate the pre-processing step needed to obtain the slack time  $h_i$  for each job  $\theta_i$ ,  $1 \leq i \leq m$ . According to the deadline constraints, we have:

$$\max \{a_i + b_i + h_i, a_{i+1}\} + b_{i+1} + h_{i+1} \leq c_{i+1}, \text{ where } 0 \leq i \leq m - 1.$$

On the other hand, to ensure each job has the minimum time-redundancy for fault-tolerance, we require the slack of each job to be greater than a constant threshold value  $Q$ , which is defined as a given number of checkpoints. Then we have:  $h_i \geq Q$ , where  $1 \leq i \leq m$ .

The problem now is that of allocating the slacks appropriately to the jobs subject to the above constraints. If we choose the total slack as the optimization function, then the problem is how to maximize the sum of all slacks. This is a linear-programming problem and  $h_i$  can be obtained by employing linear programming solver tools such as BPMPD [14]. Since this processing step is done off-line prior to the actual execution of the job set, the additional computation is acceptable.

### 5.2. Modification to the adapchp procedure

After obtaining the slacks for all jobs off-line through the pre-processing step, we next incorporate them into the on-line adaptive checkpointing scheme. The *adapchp* procedure of Figure 33-2 needs to be modified due to the following reasons.

(1) In the *adapchp* procedure, a job is declared to be unschedulable if the deadline is missed. When this happens, the execution of the entire set of jobs is terminated. Here we are using checkpointing deadline as an input parameter for the adaptive checkpointing procedure. Sometimes however, a job deadline is not really missed even if its checkpointing deadline is missed. Therefore it

is not correct to always declare the job as unschedulable if its checkpointing deadline is missed. We overcome this problem by adopting the following solution: if the checkpointing deadline is missed but the actual job deadline is not missed, the job continues execution without inserting any more checkpoints.

(2) Since the actual execution time of a job is affected by fault arrival pattern, it is necessary to adjust the starting time and slack of the next job during execution. In our proposed solution, during actual execution, once the current job finishes its execution, the *adapchp* procedure returns its completion time. The next job starts its execution based on the previous job's completion time and its own pre-computed starting time, which is obtained off-line. Meanwhile, the pre-computed slack of the next job is adjusted accordingly. We explain this formally below.

Let the actual starting time of  $\theta_i$  be  $a'_i$ , and the actual execution time be  $b'_i$ . Then the actual starting time  $a'_{i+1}$  of the next job  $\theta_{i+1}$  can be calculated as:

$$a'_{i+1} = \max \{a_{i+1}, a'_i + b'_i\}.$$

The actual slack time  $h'_{i+1}$  of  $\theta_{i+1}$  is adjusted as:

$$h'_{i+1} = h_{i+1} - (a'_{i+1} - a_{i+1}),$$

and the actual checkpointing deadline  $v'_{i+1}$  is adjusted as:

$$v'_{i+1} = \begin{cases} b_{i+1} & \text{if } h'_{i+1} < 0 \\ b_{i+1} + h'_{i+1} & \text{if } h'_{i+1} \geq 0 \end{cases}$$

Then we can apply *adapchp*( $v'_{i+1}$ ,  $b_{i+1}$ ,  $C$ ,  $k$ ,  $\lambda$ ) to job  $\theta_{i+1}$ ; the procedure returns  $b'_{i+1}$ , the value which is the actual execution time of  $\theta_{i+1}$  including checkpointing and fault recovery cost.

### 5.3. Experimental results

We consider two tasks  $\tau_1 = (5000, 7000, 12000)$  and  $\tau_2 = (4000, 11000, 18000)$ . Here we assume that the single checkpointing cost is 10, and we require that at least 20 checkpoints are inserted for each slack.

We compare the multi-task adaptive scheme, denoted by ADT\_MUL, with the Poisson-arrival and  $k$ -fault-tolerant schemes in terms of the probability of timely completion. The experimental results are shown in Table 33-7. These results show that the adaptive scheme provides a higher probability of timely completion for multi-task systems than the other two schemes.

## 6. CONCLUSIONS

We have presented a unified approach for adaptive checkpointing and dynamic voltage scaling for a real-time task executing in an embedded system. This

Table 33-7. Results on adaptive checkpointing for five jobs: variation of  $P$  with  $\lambda$  for (a)  $C = 10$ , and  $k = 5$ . (b)  $C = 20$ , and  $k = 5$ .

	Fault arrival rate $\lambda$ ( $\times 10^{-4}$ )	Probability of timely completion of tasks, $P$		
		Poisson-arrival	$k$ -fault-tolerant	ADT_MUL
(a)	6	0.996	1.000	1.000
	10	0.967	0.983	0.987
	14	0.870	0.899	0.914
	18	0.647	0.610	0.696
(b)	6	0.884	0.815	0.930
	10	0.487	0.370	0.542
	14	0.149	0.086	0.153
	18	0.018	0.013	0.028

approach provides fault tolerance and facilitates dynamic power management. The proposed energy-aware adaptive checkpointing scheme uses a dynamic voltage scaling criterion that is based not only on the slack in task execution but also on the occurrences of faults during task execution. We have presented simulation results to show that the proposed approach significantly reduces power consumption and increases the probability of tasks completing correctly on time despite the occurrences of faults. We have also extended the adaptive checkpointing scheme to a set of multiple tasks. A linear-programming model is employed in an off-line manner to obtain the relevant parameters that are used by the adaptive checkpointing procedure. Simulation results show that the adaptive scheme is also capable of providing a high probability of timely completion in the presence of faults for a set of multiple tasks.

We are currently investigating checkpointing for distributed systems with multiple processing elements, where data dependencies and inter-node communication have a significant impact on the checkpointing strategy. We are examining ways to relax the restrictions of zero rollback and state restoration costs, as well as the assumption of no fault occurrence during checkpointing and rollback recovery. It is quite straightforward to model nonzero rollback and state restoration cost, but it appears that the assumption of no faults during checkpointing is difficult to relax.

## REFERENCES

1. P. Pop, P. Eles and Z. Peng. "Schedulability Analysis for Systems with Data and Control Dependencies." *Proceedings of Euromicro RTS*, pp. 201–208, June 2000.
2. T. Ishihara and H. Yasuura. "Voltage Scheduling Problem for Dynamically Variable Voltage Processors." *Proceedings of International Symposium on Low Power Electronics and Design*, August 1998.
3. Y. Shin, K. Choi, and T. Sakurai. "Power Optimization of Real-Time Embedded Systems

- on Variable Speed Processors.” *Proceedings of International Conference on Computer-Aided Design*, pp. 365–368, June 2000.
4. G. Quan and X. Hu. “Energy Efficient Fixed-priority Scheduling for Real-Time Systems on Variable Voltage Processors.” *Proceedings of Design Automation Conference*, pp. 828–833, June 2001.
  5. K. G. Shin and Y.-H. Lee. “Error Detection Process – Model, Design and Its Impact on Computer Performance.” *IEEE Trans. on Computers*, Vol. C-33, pp. 529–540, June 1984.
  6. K. M. Chandy, J. C. Browne, C. W. Dissly, and W. R. Uhrig. “Analytic Models for Rollback and Recovery Strategies in Data Base Systems.” *IEEE Trans. Software Eng.*, Vol. 1, pp. 100–110, March 1975.
  7. K. Shin, T. Lin, and Y. Lee. “Optimal Checkpointing of Real-Time Tasks.” *IEEE Trans. Computers*, Vol. 36, No. 11, pp. 1328–1341, November 1987.
  8. A. Ziv and J. Bruck. “An On-Line Algorithm for Checkpoint Placement.” *IEEE Trans. Computers*, Vol. 46, No. 9, pp. 976–985, September 1997.
  9. S. W. Kwak, B. J. Choi, and B. K. Kim. “An Optimal Checkpointing-Strategy for Real-Time Control Systems under Transient Faults.” *IEEE Trans. Reliability*, Vol. 50, No. 3, pp. 293–301, September 2001.
  10. M. L. Bushnell and V. D. Agrawal. *Essentials of Electronic Testing*, Kluwer Academic Publishers, Norwell, MA, 2000.
  11. J. W. Liu. *Real-Time Systems*, Prentice Hall, Upper Saddle River, NJ, 2000.
  12. A. Duda. “The Effects of Checkpointing on Program Execution Time.” *Information Processing Letters*, Vol. 16, pp. 221–229, June 1983.
  13. H. Lee, H. Shin, and S. Min, “Worst Case Timing Requirement of Real-Time Tasks with Time Redundancy.” *Proceedings of Real-Time Computing Systems and Applications*, pp. 410–414, 1999.
  14. NEOS Solvers: <http://www-neos.mcs.anl.gov/neos/server-solver-types.html>.

*This page intentionally left blank*

PART X:

LO POWER SOFTWARE

*This page intentionally left blank*

## Chapter 34

# SOFTWARE ARCHITECTURAL TRANSFORMATIONS

## *A New Approach to Low Energy Embedded Software*

Tat K. Tan<sup>1</sup>, Anand Raghunathan<sup>2</sup> and Niraj K. Jha<sup>1</sup>

<sup>1</sup> *Department of Electrical Engineering, Princeton University, Princeton, NJ 08544, USA;*

<sup>2</sup> *NEC Laboratory, Princeton, NJ 08540, USA; E-mail: {tktan,jha}@ee.princeton.edu, anand@nec-labs.com*

**Abstract.** We consider software architectural transformations in the context of the multi-process software style driven by an operating system (OS), which is very commonly employed in energy-sensitive embedded systems. We propose a systematic methodology for applying these transformations to derive an energy-optimized architecture for any given embedded software. It consists of: (i) constructing a software architecture graph representation, (ii) deriving initial energy and performance statistics using a detailed energy simulation framework, (iii) constructing sequences of atomic software architectural transformations, guided by energy change estimates derived from high-level energy macro-models, that result in maximal energy reduction, and (iv) generation of program source code to reflect the optimized software architecture. We employ a wide suite of software architectural transformations whose effects span the application-OS boundary, including how the program functionality is structured into architectural components (*e.g.*, application processes, signal handlers, and device drivers) and connectors between them (inter-component synchronization and communication mechanisms).

The effects of the software transformations we consider cross the application-OS boundary (by changing the way in which the application interacts with the OS). Hence, we use an OS-aware energy simulation framework to perform an initial energy profiling, and OS energy macro-models to provide energy change estimates that guide the application of our transformations.

We present experimental results on several multi-process embedded software programs, in the context of an embedded system that features the Intel StrongARM processor and the embedded Linux OS. The presented results clearly underscore the potential of the proposed methodology (up to 66.1% reduction in energy is obtained). In a broader sense, our work demonstrates the impact of considering energy during the earlier stages of the software design process.

**Key words:** embedded software, software architectural transformations, OS-driven software

### 1. INTRODUCTION

Low power design techniques have been investigated in the hardware design domain at various levels of the design hierarchy. Figure 34-1 presents the different levels of hardware design abstraction and illustrates that the efficiency of analysis, and the amount of power savings obtainable, are much

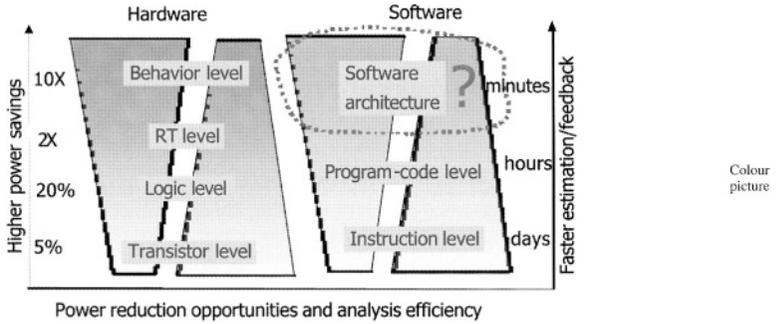


Figure 34-1. Analysis and optimization efficiency at different levels of design abstraction.

larger at higher levels [1–3]. It is natural to hypothesize whether such an observation can be extended to the software design domain. In the software design domain, low power techniques have been extensively investigated at the instruction level, and through enhancements to various stages of the high-level programming language compilation process. However, if we consider the assembly instruction level for software to be analogous to the logic level for hardware, and software compilation to be analogous to logic synthesis, it is clear that there exists a “gap” at the software architecture level, where very little, if any, research has been conducted on energy reduction.

In this chapter, we focus on the impact of software architecture design on energy consumption. We consider various *software architectural transformations* that affect how the program functionality is structured into architectural components, as well as the connectors among them [4]. The specific software architectural style that we use for our illustrations consists of multi-process applications being executed in the run-time environment provided by an embedded OS. We call this an OS-driven multi-process software architectural style. The architectural components correspond to software entities such as application processes, signal handlers, device drivers, etc., while connectors include inter-process communication (IPC) and synchronization mechanisms.

Within the framework of the OS-driven multi-process style, we consider two broad classes of software architectural transformations. The first class is component structuring, which refers to the task of converting a behavioral description (model) of the application into a software architectural model that consists of software architectural components. Systematic criteria for process structuring (without regard to energy consumption) have been discussed in previous work [5]. The other class, called connector replacement, is the replacement of one communication or synchronization method with another, within the scope of alternatives provided by the chosen OS. These two broad classes of software architectural transformations can be thought of as *moves* in the general area of design synthesis.

By defining an appropriate software architectural representation, it is also possible to automatically perform software architectural transformations to arrive at an optimized software architecture. Naturally, reasonably efficient and accurate feedback (estimation) mechanisms need to be provided in order to drive software architectural exploration. In the case of energy optimization, the energy impact of each architectural transformation, or “move”, needs to be evaluated. Our work advocates the use of high-level energy macro-models for the system functions to derive the feedback mechanism that is used to guide the architectural transformations.

In the next section, we highlight our contributions and discuss some related work. In Section 3, we describe our software architectural transformation methodology in detail. In Section 4, we describe the experimental method used to evaluate our techniques, and present the experimental results. We conclude in Section 5.

## **2. CONTRIBUTIONS AND RELATED WORK**

In this section, we highlight our contributions and discuss some related work.

### **2.1. Contributions**

The major contributions of this work are as follows:

- We propose a systematic methodology for applying software architectural transformations, which consists of: (i) constructing a software architecture graph to represent a software program, (ii) deriving an initial profile of the energy consumption and execution statistics using a detailed energy simulation framework, (iii) evaluating the energy impact of atomic software architecture transformations through the use of energy macro-models, (iv) constructing sequences of atomic transformations that result in maximal energy reduction, and (v) generation of program source code to reflect the optimized software architecture.
- We show how to use OS energy macro-models to provide energy change estimates that predict the effect of various software architectural transformations.
- We experimentally demonstrate the utility of our techniques in the context of several typical programs running on an embedded system that features the Intel StrongARM processor and embedded Linux as the OS. Significant energy savings (up to 66.1%) were achieved for the energy-efficient software architectures generated by the proposed techniques.

### **2.2. Related work**

Low energy software design at the instruction level has been extensively investigated. Research work in this area usually involves the idea of adapting

one or more steps in the compilation process, including transformations, instruction selection, register assignment, instruction scheduling, etc. Representative work in this area includes, but is not limited to, [6–10]. At one level above the instruction level, the performance and energy impact of source code transformations has been investigated in [11].

System-level software energy reduction strategies usually involve some kind of resource scheduling policy, whether it is scheduling of the processor or the peripherals. In complex embedded systems, this often centers around the adaptation of the OS. Some general ideas on adapting software to manage energy consumption were presented in [12–14]. Vahdat et al. [15] proposed revisiting several aspects of the OS for potential improvement in energy efficiency. Lu et al. [16] proposed and implemented OS-directed power management in Linux and showed that it can save more than 50% power compared to traditional hardware-centric shut-down techniques. Bellosa [17] presented thread-specific online analysis of energy-usage patterns that are fed back to the scheduler to control the CPU clock speed.

Software architecture has been an active research area for the past few years in the real-time software and software engineering communities, with emphasis on software understanding and reverse engineering. Many common software architectural styles are surveyed in [4]. Wang et al. [18] evaluated the performance and availability of two different software architectural styles. Carrière et al. [19] studied the effect of connector transformation at the software architecture level in a client-server application. None of these studies considered the effect of the software architecture on energy consumption. Our work, on the other hand, provides a systematic framework for harnessing software architectural transformations to minimize energy consumption.

IMEC’s work on embedded software synthesis [20] considers synthesis of real-time multi-threaded software based on a multi-thread graph (MTG) model. They focus on static scheduling of the threads without the use of an OS. They do not target energy consumption. Our work also considers multi-process embedded software, but emphasizes energy-efficient usage of the OS through appropriate software transformations.

### **3. ENERGY MINIMIZATION THROUGH SOFTWARE ARCHITECTURAL TRANSFORMATIONS**

In this section, we describe a methodology for minimizing the energy consumption of embedded software through software architectural transformations. Section 3.1 provides an overview of the entire flow, while Sections 3.2 through 3.5 detail the important aspects.

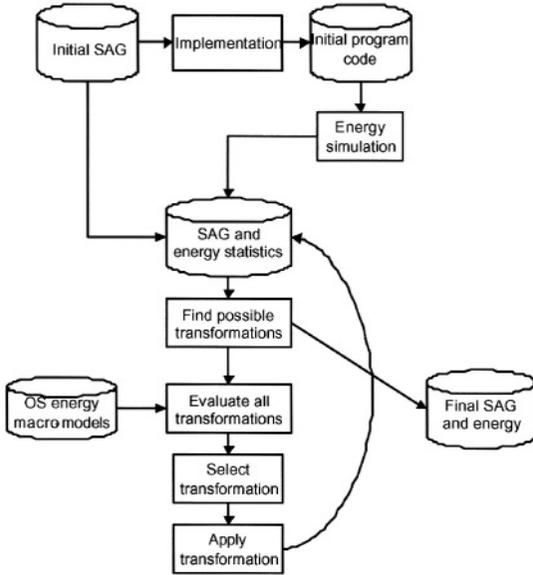


Figure 34-2. The software architecture level energy minimization methodology.

### 3.1. Overview of the software architectural energy minimization methodology

The methodology to minimize the energy consumption of embedded software through software architectural transformations is illustrated in Figure 34-2.

In the figure, the cylinders represent the entities to be operated upon, and the rectangles represent the steps in the algorithm. The methodology starts with an initial software architecture, represented as a *software architecture graph* or SAG (as described in Section 3.2). The energy consumption  $E_0$  of the initial software architecture is obtained by compiling the corresponding program source code into an optimized binary for the target embedded system, and by profiling this implementation using a detailed energy simulation framework.<sup>1</sup> The execution and energy statistics collected from this step are subsequently used to guide the application of software architectural transformations. Our methodology optimizes the software architecture by applying a sequence of atomic software architectural transformations, or moves, that lead to maximum energy savings. These transformations are formulated as transformations of the SAG, and are described in detail in Section 3.5. We use an iterative improvement strategy to explore sequences of transformations. Selection of a transformation at each iteration is done in a greedy fashion. That is, at each iteration, we choose from all the possible transformations the one that yields maximum energy reduction. The iterative process ends when no more transformation is possible.

During the iterative process, for each architectural transformation considered, we need to estimate the resulting change in energy consumption. Since this estimation is iterated a large number of times, it needs to be much more efficient than hardware or instruction-level simulation. We utilize high-level energy macro-models to provide energy change estimates, as described in Section 3.4.

After an energy-efficient software architecture is obtained (as an SAG), the program source code is generated to reflect the optimized software architecture. The optimized program code can be executed in the low-level energy simulation framework to obtain accurate energy estimates for the optimized software architecture.

The remainder of this section details the important steps in the proposed methodology.

### 3.2. Software architecture graph

We represent the architecture of an embedded software program as an SAG. An example of an SAG, for a program employed in a situational awareness system, is depicted in Figure 34-3.

In the SAG, vertices represent hardware or software entities. Vertices can be of several types:

- Hardware devices are represented by an empty box, with an optional crossbar for active devices (e.g., the UART peripheral in Figure 34-3).

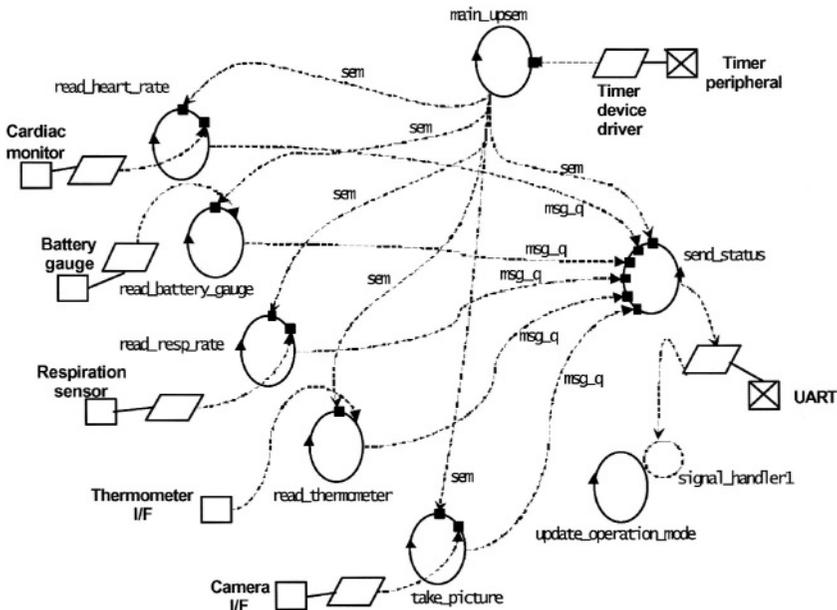


Figure 34-3. Software architecture graph for a situational awareness sub-system.

- Device drivers are represented by a parallelogram (e.g., the timer device driver in Figure 34-3).
- Application processes are represented by an ellipse with an arrow on the perimeter (e.g., process `read_heart_rate` in Figure 34-3). Signal handlers are represented by a dotted circle attached to the corresponding application process (e.g., `signal_handler1`, which is attached to process `update_operation_mode` in Figure 34-3).

The association between hardware devices and device drivers is depicted by a solid line connecting them. A box with a crossbar represents an active hardware device, whereas a box without a crossbar represents a passive hardware device. Explanation of the difference between an active device and a passive device is given later. Arrowed edges between any two vertices represent the communication of data or control messages, and are annotated with the selected IPC mechanisms. Since the system is OS-driven, they are to be implemented as calls to the system functions. For example, the edge from process `main_upsem` to process `read_heart_rate` in Figure 34-3 is a control message that is implemented using the semaphore service provided by the OS. Naturally, an edge from or to a hardware device represents the transfer of data using OS system functions. A small solid square at the termination of an edge indicates a blocking communication. Otherwise, the communication is assumed to be non-blocking.

Behind the software architectural representation is a model of computation featuring the following details:

1. A hardware device can be either active or passive. An active device initiates data transfer spontaneously. A timer is a special active device that spontaneously initiates data (signal) transfer at regular intervals. A passive device, on the other hand, waits for the processor to initiate data transfer. There are two types of passive devices. Fast devices respond to the request from the processor immediately, slow devices require some time (typically more than 1 ms) to respond to the processor. To efficiently use the processor, data transfer with a slow passive device is always blocking, so that the process requesting the data transfer can be suspended and the OS can bring in other processes for execution. On the other hand, data transfer with a fast passive device is naturally non-blocking since the device will respond to the processor request immediately.
2. Instead of responding to the processor's request, an active device can initiate a data transfer. Therefore, data transfer with an active device has to be blocking, waiting for the active device to take the initiative. Moreover, to avoid data loss, no process in the SAG should be blocked by more than one active device, either directly, or indirectly through other processes. On the other hand, to efficiently use the processor (no busy waiting), every process in the SAG should be blocked (directly or indirectly) by at least one active device. Therefore, the natural consequence of these requirements restricts the number of "active" blocking points for

each process to be exactly one. In other words, two processes driven by two different active devices can only communicate through a non-blocking IPC.

3. An SAG, as given, assumes implicitly a schedule of the processes. The validity of this schedule has to be checked separately. A valid schedule has to meet several requirements. A necessary requirement is to have no deadlock. A stronger requirement is to have all the processes meet the deadline. That is, all the processes interfacing with the active devices should not miss any request initiated by these devices. An even stronger, but optional, requirement, is to allow efficient utilization of the CPU. That is, no busy waiting should be employed by any process.

### 3.3. Energy modeling at the software architecture level

We denote a specific software architecture configuration for a given embedded software program as  $C$ . The energy consumption of this software architecture for a fixed amount of computation (application functionality) is denoted by  $E(C)$ . The energy consumption of the initial architecture  $C_0$  is denoted by  $E_0 = E(C_0)$ . In an absolute sense, for a given embedded system,  $E(C)$  depends on various parameters related to how the software architecture is translated into the final executable implementation (e.g., compiler optimizations used). The energy consumption of each software architecture could be accurately evaluated by specifying it as program source code, feeding it through the software compilation flow, and using a detailed system simulation framework. However, this would be too time-consuming to use in the context of an automatic software architectural transformation framework, since each candidate architecture could require hours to evaluate. In our context, the following observations are worth noting:

- We are only interested in comparing the inherent energy efficiency of different software architectures without regard to the subsequent software implementation process.
- We perform a (one-time) detailed energy profiling of the initial software architecture. The initial architecture is modified by applying a sequence of atomic transformations. Hence, we only require *energy change estimates*, i.e., estimates of the difference in energy consumption before and after the application of each atomic software architectural transformation.
- The transformations utilized in our methodology do not affect the “core functionality” of an application. Rather, they affect energy consumption by altering the manner in which OS services are employed. As shown later, this implies that we can use high-level OS energy macro-models to evaluate the energy impact of software architectural transformations.

The initial configuration  $C_0$  can be easily obtained by doing a one-to-one mapping from the behavioral model to the software architectural model, resulting in a so-called fully parallel software architecture. Such a mapping

preserves the required functionality. Given the initial configuration  $C_0$ , we can find an energy-optimized configuration  $C_E$  by performing a series of *functionality-invariant* transformations on  $C_0$ . The effects of these transformations can be analyzed in the following manner.

Given a software architecture configuration  $C_1$ , and a transformation  $T_{C_1 \rightarrow C_2}$ , a new configuration  $C_2$  is created. Transformation  $T_{C_1 \rightarrow C_2}$  has to be underscored by  $C_1$  and  $C_2$  because its specific moves are defined based on the knowledge of  $C_1$  and  $C_2$ . The equation relating the energy consumption of these two configurations is:

$$E(C_2) - E(C_1) = \Delta E(T_{C_1 \rightarrow C_2}) \quad (1)$$

where  $\Delta E(T_{C_1 \rightarrow C_2})$  denotes the energy change incurred by performing transformation  $T_{C_1 \rightarrow C_2}$ .

In the next sub-section, we will show how  $\Delta E(T_{C_1 \rightarrow C_2})$  can be estimated for the selected set of transformations we are interested in.

### 3.4. Energy change estimation for the transformations

In theory, a software architecture transformation  $T_{C_1 \rightarrow C_2}$  can relate any two arbitrary software architecture configurations  $C_1$  and  $C_2$ . However, the actual change we are going to introduce at every transformation step can be incremental, by choice. Such a restriction to the use of atomic transformations at every step can make the estimation of  $\Delta E(T_{C_1 \rightarrow C_2})$  easier if the change is localized. That is, at every transformation step, only a small subset of all the components (processes) in configuration  $C_1$  is involved. By this restriction, we can make the following approximation:

$$\Delta E(T_{C_1 \rightarrow C_2}) = \Delta E(M) \quad (2)$$

where  $M$  is the incremental move that carries universal definition, instead of being tied to  $C_1$  or  $C_2$ .

By decoupling move  $M$  from the actual configuration, we are able to estimate  $\Delta E$  using high-level energy macro-models specific to properties of the architectural style we have chosen. In our case, since the style of choice is OS-driven, the high-level energy macro-models are actually characteristics of the OS. An energy macro-model is a function (e.g., an equation) expressing the relationship between the energy consumption of a software function and some predefined parameters. In the above-mentioned context, these energy macro-models are called the OS energy characteristic data [21]. The OS energy macro-models proposed in [21] show on average 5.9% error with respect to the energy data used to obtain the macro-models. This level of accuracy is sufficient for relative comparisons of different transformations.

Basically, the OS energy characteristic data consist of the following:

- *The explicit set:* These include the energy macro-models for those system functions that are explicitly invoked by application software, e.g., IPC

mechanisms. Given these energy macro-models and execution statistics collected during the initial system simulation, the energy change due to an atomic software architectural transformation can be computed. For example, suppose the energy macro-models for two IPC's,  $IPC_1$  and  $IPC_2$ , are given by:

$$E_{ipc1}(x) = c_{11} + c_{12} x \quad (3)$$

$$E_{ipc2}(x) = c_{21} + c_{22} x \quad (4)$$

where  $x$  is the number of bytes being transferred in each call, and  $c$ 's are the coefficients of the macro-models. The amount of energy change incurred by replacing  $IPC_1$  with  $IPC_2$  is given by

$$\Delta E = [E_{ipc2}(x) - E_{ipc1}(x)] N_{ipc} \quad (5)$$

$$= [(c_{21} - c_{11}) + (c_{22} - c_{12})x] N_{ipc} \quad (6)$$

where  $N_{ipc}$  is the number of times this IPC is invoked in the specific application process under consideration. The values of parameters such as  $N_{ipc}$  and  $x$  are collected during the detailed profiling of the initial software architecture. The energy changes due to other system function replacements can be calculated similarly.

- *The implicit set:* These include the macro-models for the context-switch energy, timer-interrupt energy and re-scheduling energy. In particular, the context-switch energy,  $E_{ctx}$ , is required to calculate the energy change due to process merging.  $E_{ctx}$  is defined to be the amount of round-trip energy overhead incurred every time a process is switched out (and switched in again) [21].<sup>2</sup> This definition is convenient for calculating the energy change due to process merging.

In the next section, we introduce a set of atomic software architectural transformations that we have selected. Computation of the energy changes incurred by these transformations is facilitated by the availability of the OS energy macro-models.

### 3.5. Proposed software architectural transformations

Since we have adopted the OS-driven multi-process software architectural style, the moves that we consider are mainly manipulations of the components (application processes, signal handlers, and device drivers), and the connectors (inter-process synchronization and communication mechanisms). Some of the specific transformations presented here, including manipulation of application processes or process structuring, have been investigated in other related areas such as software engineering, mostly in a qualitative manner. For example, a good introduction to process structuring can be found in [5]. In this sub-section, we formulate a wide range of atomic software architectural transformations and show how they can be systematically used for transformations of the SAG. For each atomic transformation, we also include a brief analysis of the energy change incurred. Note that some of these atomic

transformations may not directly result in a large energy reduction, but may enable other moves that reduce more energy eventually. Also, note that the inverse of any move discussed here is also a valid move.

### 3.5.1. Temporal cohesion driven process merging

This transformation involves merging of two software processes that are driven by events whose instances have a one-to-one correspondence. A special case is a periodic event that occurs at the same rate. Application of this transformation decreases the number of processes by one. This transformation is illustrated in Figure 34-4.

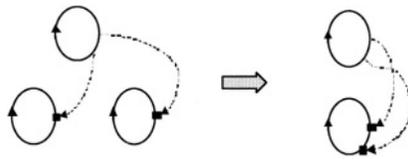


Figure 34-4. Temporal cohesion driven process merging.

The energy change due to this transformation is estimated as:

$$\Delta E = -E_{ctx}N_{ctx} \quad (7)$$

where  $E_{ctx}$  is the context switch energy explained before, and  $N_{ctx}$  is the number of times each of the processes is activated. Note that this transformation can be collated with other transformations such as code computation migration and IPC merging to further reduce energy, as illustrated later.

### 3.5.2. Sequential cohesion driven process merging

This transformation involves merging of two software processes that are executed sequentially because one process passes data or control to the other. It decreases the number of processes by one and also removes the IPC between them. This transformation is illustrated in Figure 34-5.

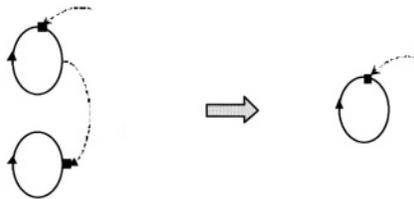


Figure 34-5. Sequential cohesion driven process merging.

The energy change due to this transformation is estimated as:

$$\Delta E = -E_{ipc}(x)N_{ipc} - E_{ctx}N_{ctx} \quad (8)$$

where  $E_{ipc}(x)$  is the total IPC (read and write) energy for communicating  $x$  amount of data,  $N_{ipc}$  is the number of times the IPC between the two processes is invoked, and  $N_{ctx}$  is the number of times each of the processes is activated.

### 3.5.3. Intra-process computation migration

This transformation involves moving some of the code in a process so that two IPC writes (or sends) can be replaced by a single write (or send). It exploits the capability of IPC system calls that can accept multiple messages (with potentially different destinations), e.g., the `msg_snd()` function in Linux. It is useful in reducing the constant overhead involved in invoking the IPC function. It is illustrated in Figure 34-6.



Figure 34-6. Intra-process computation migration.

The energy change due to this transformation is estimated as:

$$\Delta E = -[E_{ipc\_wr}(x) + E_{ipc\_wr}(y) - E_{ipc\_wr}(x + y)] N_{ipc\_wr} \quad (9)$$

where  $E_{ipc\_wr}$  refers to IPC write energy,  $N_{ipc\_wr}$  is the number of times one of the IPC writes is invoked, and  $x$  and  $y$  represent the average amount of data transferred per call through the first and second IPC writes, respectively. Note that this transformation enables further IPC merging in some cases.

### 3.5.4. Temporal cohesion driven IPC merging

This transformation involves merging of two IPC's that are driven by the same event. This move is illustrated in Figure 34-7.

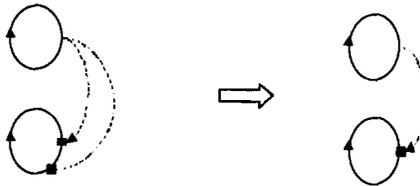


Figure 34-7. Temporal cohesion driven IPC merging.

The energy change due to this transformation is estimated as:

$$\Delta E = -E_{ipc\_read}(x) N_{ipc\_read} \quad (10)$$

where  $E_{ipc\_read}$  is the IPC read energy for  $x$  amount of data and  $N_{ipc\_read}$  is the number of times one of the IPC reads is invoked.

3.5.5. IPC replacement

This transformation involves replacing the IPC mechanism associated with an edge in the SAG by another functionally equivalent mechanism. For example, message passing can be replaced by shared memory with semaphore protection. This transformation is illustrated in Figure 34-8.



Figure 34-8. IPC replacement.

The energy change due to this transformation is estimated as:

$$\Delta E = [E_{ipc2}(x) - E_{ipc1}(x)] N_{ipc} \quad (11)$$

where  $E_{ipc1}$  and  $E_{ipc2}$  refer to the energy of the first and second IPC mechanisms, respectively, and  $N_{ipc}$  is the number of times this IPC is invoked.

3.5.6. System function replacement

System functions initially used in the application program may not be the most energy-efficient choices under the specific context in which they are used. In this case, replacement of these system functions by lower energy alternatives leads to energy reduction. This transformation is illustrated in Figure 34-9.

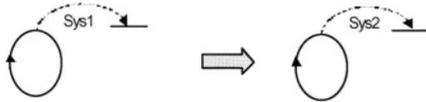


Figure 34-9. System function replacement.

The energy change due to this transformation is estimated as:

$$\Delta E = [E_{sys2}(x) - E_{sys1}(x)] N_{sys} \quad (12)$$

where  $E_{sys1}$  ( $E_{sys2}$ ) is the energy of system function *sys1* (*sys2*) and  $N_{sys}$  is the number of times the system function is invoked.

3.5.7. Process embedding as signal handlers

This advanced transformation embeds a process as a signal handler into another process. By doing so, the number of processes is reduced by one, reducing context switch related energy. However, there is an overhead due to signal handling. This transformation is illustrated in Figure 34-10.

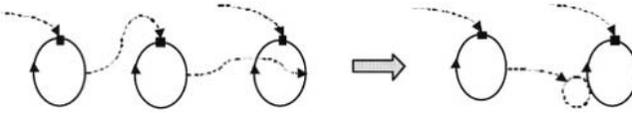


Figure 34-10. Process embedding.

The dashed circle attached to the second process of the transformed SAG represents the signal handler of the second process. The energy change due to this transformation is estimated as:

$$\Delta E = -E_{ctx}N_{ctx} + E_{sig}N_{sig} \quad (13)$$

where  $N_{sig}$  is the number of times the signal handler, being the replacement of the embedded process, is activated.  $E_{sig}$  is the signal handling energy.

#### 4. EXPERIMENTAL RESULTS

As a proof of concept, we applied our software architecture energy minimization methodology to various multi-process example programs designed to run under the embedded Linux OS, specifically `arm-linux v2.2.2` [22]. The benchmarks include: *Aware* – the situational awareness system shown in Figure 24-3, *Headphone* – a program used in an audio headset application, *Vcam* – embedded software from a video camera recorder, *Climate* – a telemetric application for collecting and processing climate information, *Navigator* – software from a global navigation system, and *ATR* – a part of an automatic target recognition program. The hardware platform for our experiments was based on the Intel StrongARM embedded processor [23]. We used EMSIM [24] as the detailed energy simulation framework in our experiments. With detailed system models, EMSIM allows execution of the Linux OS within the simulated environment. The high-level energy macro-models of the Linux OS, which were used to guide the application of software architectural transformations, were obtained from [21].

Table 34-1 shows the experimental results. Major columns 2 and 3 show the details of the original and optimized software programs, respectively. # proc denotes the number of processes. # ipc denotes the number of IPC's involved. Major column 4 shows the energy reduction as a percentage of the original energy consumption. Significant energy reductions (up to 66.1%, average of 25.1%) can be observed for the examples. Note that the energy savings obtained through software architectural transformations are largely independent of, and complementary to, energy savings obtained through lower-level optimizations, including compiler techniques.

We next illustrate the application of architectural transformations to a simple software program.

We consider the climate system from Table 34-1. An SAG that repre-

Table 34-1. Experimental results showing the energy impact of software architectural transformations.

Example	Original			Optimized			% Energy reduction
	Energy (mJ)	# proc	# ipc	Energy (mJ)	# proc	# ipc	
Aware	12.956	8	11	8.204	7	9	36.7%
Headphone	1.668	6	8	1.461	3	2	12.4%
Vcam	1.572	4	5	1.375	2	1	12.5%
Climate	0.239	4	5	0.081	2	1	66.1%
Navigator	1.659	5	7	1.456	3	3	12.2%
ATR	6.940	4	7	6.199	3	4	10.7%

sents the initial software architecture for this program is shown in Figure 34-11(a). Explanation of the annotations in the figure were provided in Section 3.2. For the purpose of illustration, we consider this program to execute under the embedded Linux OS. Edges coming from the hardware devices (indicated as empty square boxes) indicate the data flow from the devices to the application processes, and are implemented using `read()` system calls in Linux. Some of these reads are blocking, indicated by a small square box at the tip of the arrow.

The initial software architecture program code is written in C. The energy consumption statistics of this program are obtained by running it in the energy simulation framework EMSIM [24]. Under the initial software architecture,

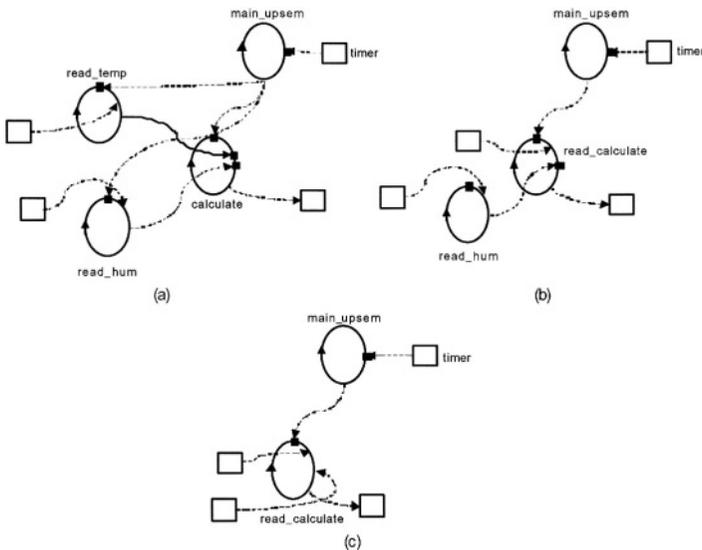


Figure 34-11. Sequence of software architectural transformations for an example system.

the energy consumption of the entire system for three iterations of program execution is 0.239 mJ. That is,  $E_0 = 0.239$  mJ. The first two transformations merge the software processes `read_temp` and `calculate` as well as the IPC edges that involve them, resulting in the `read_calculate` process. The new SAG is shown in Figure 34-11(b). Note that the evaluation of the transformations is performed with the help of OS energy macro-models, as described in Section 3.4. The next two transformations merge `read_hum` and `read_calculate` as well as the IPC edges that involve them. The final SAG is shown in Figure 34-11(c).

The transformed program is re-constructed from the final software architecture, and simulated again in the EMSIM energy simulation framework. The energy consumption estimate is obtained to be  $E = 0.081$  mJ, which corresponds to an energy savings of 66.1%.

## 5. CONCLUSIONS

Energy minimization of embedded software at the software architecture level is a new field that awaits exploration. As a first step in this direction, we presented a systematic methodology to optimize the energy consumption of embedded software by performing a series of selected software architectural transformations. As a proof of concept, we applied the methodology to a few multi-process example programs. Experiments with the proposed methodology have demonstrated promising results, with energy reductions up to 66.1 %. We believe that software architecture level techniques for energy reduction can significantly extend and complement existing low power software design techniques.

## ACKNOWLEDGEMENTS

This work was supported by DARPA under contract no. DAAB07-02-C-P302.

## NOTES

<sup>1</sup> Note that the energy simulation framework needs to simulate the OS together with the application, since the effects of the software architectural transformations span the application-OS boundary.

<sup>2</sup> For repetitive processes, a process once switched out has to be switched in later.

## REFERENCES

1. J. Rabaey and M. Pedram (eds.). *Low Power Design Methodologies*, Kluwer Academic Publishers, Norwell, MA, 1996.

2. L. Benini and G. De Micheli, *Dynamic Power Management: Design Techniques and CAD Tools*, Kluwer Academic Publishers, Norwell, MA, 1997.
3. A. Raghunathan, N. K. Jha, and S. Dey. *High-level Power Analysis and Optimization*, Kluwer Academic Publishers, Norwell, MA, 1998.
4. D. Garlan and M. Shaw. "An Introduction to Software Architecture." *Technical Report CMU-CS-94-166*, School of Computer Science, Carnegie-Mellon University, Jan. 1994.
5. H. Gomaa. *Software Design Methods for Real-Time Systems*, Addison-Wesley, Boston, MA, 1993.
6. V. Tiwari, S. Malik, A. Wolfe, and T. C. Lee. "Instruction Level Power Analysis and Optimization of Software." *VLSI Signal Processing Systems*, Vol. 13, pp. 223–238, 1996.
7. H. Mehta, R. M. Owens, M. J. Irwin, R. Chen, and D. Ghosh. "Techniques for Low Energy Software." In *Proceedings of International Symposium on Low Power Electronics & Design*, August 1997, pp. 72–74.
8. M. Lee, V. Tiwari, S. Malik, and M. Fujita. "Power Analysis and Minimization Techniques for Embedded DSP Software." *IEEE Trans. VLSI Systems*, Vol. 2, No. 4, pp. 437–445, December 1996.
9. M. Lorenz, R. Leupers, P. Marwedel, T. Dräger and G. Fettweis, "Low-Energy DSP Code Generation Using a Genetic Algorithm." In *Proceedings of International Conference Computer Design*, September 2001, pp. 431–437.
10. N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim, and W. Ye. "Energy-Driven Integrated Hardware-Software Optimizations Using SimplePower." In *Proceedings of International Symposium on Computer Architecture*, June 2000, pp. 95–106.
11. E. Chung, L. Benini, and G. De Micheli. "Source Code Transformation Based on Software Cost Analysis." in *Proceedings of International Symposium on System Synthesis*, October 2001, pp. 153–158.
12. J. R. Lorch and A. J. Smith. "Software Strategies for Portable Computer Energy Management." *IEEE Personal Communications*, Vol. 5, No. 3, pp. 60–73, June 1998.
13. J. Flinn and M. Satyanarayanan. "Energy-Aware Adaptation for Mobile Applications." In *Proceedings of ACM Symposium on Operating System Principles*, December 1999, pp. 48–63.
14. K. I. Farkas, J. Flinn, G. Back, D. Grunwald, and J. M. Anderson. "Quantifying the Energy Consumption of a Pocket Computer and a Java Virtual Machine." In *Proceedings of SIGMETRICS*, June 2000, pp. 252–263.
15. A. Vahdat, A. Lebeck, and C. S. Ellis, "Every Joule is precious: The case for revisiting operating system design for energy efficiency," in Proc. 9th ACM SIGOPS European Workshop, Sept. 2000.
16. Y. H. Lu, L. Benini, and G. De Micheli. "Operating-System Directed Power Reduction." In *Proceedings of International Symposium on Low Power Electronics & Design*, July 2000, pp. 37–42.
17. F. Belloso. "The Benefits of Event-Driven Energy Accounting in Power-Sensitive Systems." In *Proceedings of ACM SIGOPS European Workshop*, September 2000.
18. W. L. Wang, M. H. Tang, and M. H. Chen. "Software Architecture Analysis – A Case Study." In *Proceedings Annual International Computer Software & Applications Conference*, August 1999, pp. 265–270.
19. S. J. Carrière, S. Woods, and R. Kazman. "Software Architectural Transformation." In *Proceedings of 6th Working Conf. Reverse Engineering*, October 1999.
20. M. Cornero, F. Thoen, G. Goossens, and F. Curatelli. "Software Synthesis for Real-Time Information Processing Systems." In P. Marwedel and G. Goossens (eds.), *Code Generation for Embedded Processors*, Chapter 15, pp. 260–296. Kluwer Academic Publishers, Boston, MA, 1994.
21. T. K. Tan, A. Raghunathan, and N. K. Jha. "Embedded Operating System Energy Analysis and Macro-Modeling." in *Proceedings of International Conference Computer Design*, September 2002, pp. 515–522.
22. ARM Linux, <http://www.arm.linux.org.uk/>.

23. Intel Corporation, Intel StrongARM SA-1100 Microprocessor Developer's Manual, Aug. 1999.
24. T. K. Tan, A. Raghunathan, and N. K. Jha. "EMSIM: An Energy Simulation Framework for an Embedded Operating System." in *Proceedings of International Symposium on Circuit & Systems*, May 2002, pp. 464–467.

# DYNAMIC FUNCTIONAL UNIT ASSIGNMENT FOR LOW POWER

Steve Haga, Natsha Reeves, Rajeev Barua, and Diana Marculescu  
*University of Maryland and Carnegie Mellon University, USA*

**Abstract.** A hardware method for functional unit assignment is presented, based on the principle that a functional unit's power consumption is approximated by the switching activity of its inputs. Since computing the Hamming distance of the inputs in hardware is expensive, only a portion of the inputs are examined. Integers often have many identical top bits, due to sign extension, and floating points often have many zeros in the least significant digits, due to the casting of integer values into floating point. The accuracy of these approximations is studied and the results are used to develop a simple, but effective, hardware scheme.

**Key words:** low power, functional unit assignment, bit patterns in data

## 1. INTRODUCTION

Power consumption has become a critical issue in microprocessor design, due to increasing computer complexity and clock speeds. Many techniques have been explored to reduce the power consumption of processors. Low power techniques allow increased clock speeds and battery life.

We present a simple hardware scheme to reduce the power in functional units, by examining a few bits of the operands and assigning functional units accordingly. With this technique, we succeeded in reducing the power of integer ALU operations by 17% and the power of floating point operations by 18%. In [4] it was found that around 22% of the processor's power is consumed in the execution units. Thus, the decrease in total chip power is roughly 4%. While this overall gain is modest, two points are to be noted. First, one way to reduce overall power is to combine various techniques such as ours, each targeting a different critical area of the chip. Second, reducing the execution units' power consumption by 17% to 18% is desirable, independent of the overall effect, because the execution core is one of the hot-spots of power density within the processor.

We also present an independent compiler optimization called swapping that further improves the gain for integer ALU operations to 26%.

## 2. ENERGY MODELING IN FUNCTIONAL UNITS

A series of approximations are used to develop a simple power consumption model for many computational modules. To begin, it is known [15] that the most important source of power dissipation in a module is the dynamic charging and discharging of its gates, called the *switched capacitance*. This switched capacitance is dependent upon the module's input values [14]. It has been further shown [6, 13] that the *Hamming distance* of consecutive input patterns, defined as the number of bit positions that differ between them, provides a suitable measure of power consumption. In [6, 13], power is modeled as:  $Power \approx \frac{1}{2} V_{dd}^2 f \sum_k C_k sw_k \approx \frac{1}{2} V_{dd}^2 f C_{module} h_{input}$ , where  $V_{dd}$  = voltage,  $f$  = clock frequency,  $C_k$  = capacitance of output gate  $k$ ,  $sw_k$  = average # transitions for output gate  $k$  (called *switching activity*),  $C_{module}$  = the total capacitance of the module, and  $h_{input}$  = the Hamming distance of the current inputs to the previous ones.

Since power consumption is approximately linearly proportional to  $h_{input}$ , it is desirable to minimize  $h_{input}$ . Such a minimization is possible because modern processors contain multiple integer arithmetic logic units, or IALUs, and multiple floating point arithmetic units, or FPAUs. IALU and FPAU are types of functional units, or FUs. Current superscalars assign operations to the FUs of a given type without considering power; a better assignment can reduce power, however, as illustrated by the example in Figure 35-1, which shows the input operands to three identical FUs, in two successive cycles. The alternate routing consumes less power because it reduces the Hamming distance between cycles 1 and 2. Figure 35-1 assumes that the router has the ability to not only assign an operation to any FU, but also to swap the operands, when beneficial and legal. For instance, it is legal to swap the operands of an add operation, but not those of a subtract. (A subtract's

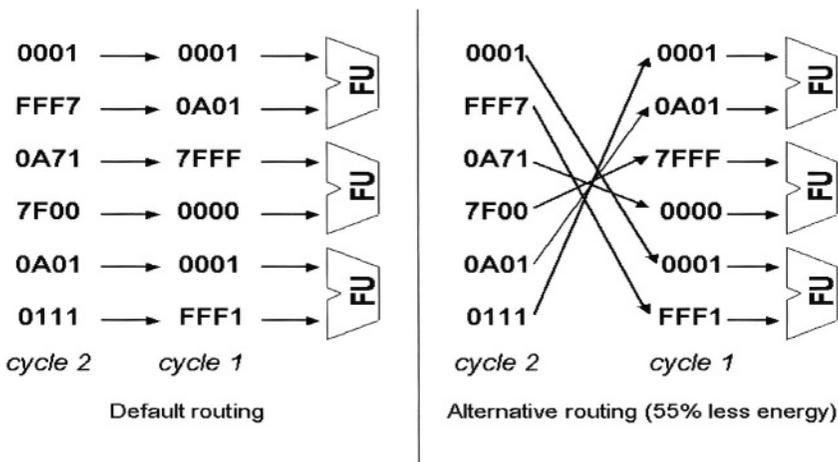


Figure 35-1. Alternative data routes for a 3-way processor.

operands can be swapped if they are first inverted, but the cost outweighs the benefit.)

Because superscalars allow out-of-order execution, a good assignment strategy should be dynamic. The case is less clear for VLIW processors, yet some of our proposed techniques are also applicable to VLIWs.

Our approach has also been extended to multiplier FUs; these units present a different set of problems. Even modern superscalars tend to only have one integer and one floating point multiplier. Some power savings are still possible, however, by switching operands. In the case of the Booth multiplier, the power is known to depend not only on the switching activity of the operands, but also on the number of 1's in the second operand [12]. Power aware routing is not beneficial for FUs that are not duplicated and not commutative, such as the divider FU.

### **3. RELATED WORK**

A variety of hardware techniques have been proposed for power reduction, however these methods have not considered functional unit assignment. Existing hardware techniques are generally independent of our method. For example, implementing a clock-slowing mechanism along with our method will not impact the additional power reduction that our approach achieves.

Work has also been done to reduce the power of an application through compiler techniques [12], by such means as improved instruction scheduling, memory bank assignment, instruction packing and operand swapping. We in fact also consider the effect of compile-time operand swapping upon our results. But the difficulty of compile-time methods is that the power consumed by an operation is highly data dependent [10], and the data behavior dynamically changes as the program executes.

The problem of statically assigning a given set of operations to specific functional units has also been addressed in the context of high-level synthesis for low power [6]. In [6], the authors present an optimal algorithm for assigning operations to existing resources in the case of data flow graphs (DFGs) without loops. More recently, [11] shows that the case of DFGs with loops is NP-complete, and proposes an approximate algorithm. The case of a modern superscalar processor employing out-of-order execution and speculation is much more complicated, however.

These difficulties point to the need for dynamic methods in hardware to reduce the power consumption.

### **4. FUNCTIONAL UNIT ASSIGNMENT FOR LOW POWER**

In this section we present an approach for assigning operations to specific FUs that reduces the total switched capacitance of the execution stage. As

shown in Figure 35-1, we can choose a power-optimal assignment by minimizing the switching activity on the inputs.

Our main assumption is that whenever an FU is not used, it has little or no dynamic power consumption. This is usually achievable via power management techniques [1, 2, 16] which use transparent latches to keep the primary input values unchanged whenever the unit is not used. We will also use this hardware feature for our purposes, and thus, we assume that it is already in place for each of the FUs. To reduce the effect of leakage current when an FU is idle, techniques such as those in [9] may also be used, if needed.

Some nomenclature is helpful in the following discussion:

**$M_j$** : the  $j^{\text{th}}$  module of the given FU type.

**$I_j$** : the  $j^{\text{th}}$  instruction to execute this cycle on this FU type.

**Num( $M$ ), Num( $I$ )**: # of modules of type  $M$ , or # of instructions of the given type. The maximum value for Num( $I$ ) is Num( $M$ ), indicating full usage of that FU type.

**OP1( $M_j$ ), OP2( $M_j$ )**: the first and second operands of the previous operation performed on this module.

**OP1( $I_j$ ), OP2( $I_j$ )**: the first and second operands of the given instruction.

**Commutative( $I_j$ )**: indicates if  $I_j$  is commutative.

**Ham( $X$ ,  $Y$ )**: the Hamming distance between the numbers  $X$  and  $Y$ . For floating point values, only the mantissa portions of the numbers are considered.

#### 4.1. The optimal assignment

Although its implementation cost is prohibitive, we first consider the optimal solution, so as to provide intuition. On a given cycle, the best assignment is the one that minimizes the total Hamming distance for all inputs. To find this, we compute the Hamming distance of each input operand to all previous input operands. The algorithm to compute these costs is shown in Figure 35-2. Once these individual costs have been computed, we examine all possible assignments and choose the one that has the smallest total cost (defined as the sum of its individual costs).

This algorithm is not practical. Since the routing logic lies on the critical path of the execution stage, such lengthy computations are sure to increase the cycle time of the machine. Moreover, the power consumed in computing so many different Hamming distances is likely to exceed the power savings from inside the modules.

```

for i = 1 to #(modules of desired type, eg. Int ALU)
  for j = 1 to #(Instructions of this type, on this cycle)
    Costji = Ham(OP1(Ij), OP1(Mi)) +
              Ham(OP2(Ij), OP2(Mi))
    if (Commutative(Ij))
      Costji = Min(Costji, (Ham(OP1(Ij), OP2(Mi))
                    + Ham(OP2(Ij), OP1(Mi)))
  
```

Figure 35-2. Finds the cost of every possible assignment.

### 4.2. Approximating the hamming distance computation

To reduce the overhead of the above strategy, full Hamming distance computations must be avoided. We achieve this by exploiting certain properties of numerical data to represent an operand by a single *information bit*.

*Integer bit patterns.* It is known [3], that most integer operands are small. As a consequence, most of the bits in a 2's complement representation are constant and represent the sign. It is likely that the Hamming distance between any two consecutive operand values will be dominated by the difference in their sign bits. Thus we can replace the operands by just their top bits, for the purposes of Figure 35-2.

Table 35-1 validates this technique. This table records the operand bit

Table 35-1. Bit patterns in data. The values in the first three columns are used to separate the results into 8 rows. Columns 1 and 2 show the information bits of both operands; (for integers, the top bit; for floating points, the OR-ing of the bottom four bits of the mantissa). Columns 4 and 7 are the occurrence frequencies for the given operand bits and commutativity pattern, as a percentage of all executions of the FU type. Columns 5, 6, 8, and 9 display, for the specified operand, the probability of any single bit being high.

OP1	OP2	Commu- tative (?)	IALU			FPAU		
			Frequency (%)	Operand 1 bit prob.	Operand 2 bit prob.	Frequency (%)	Operand 1 bit prob.	Operand 2 bit prob.
0	0	Yes	40.11	0.123	0.068	16.79	0.099	0.094
0	0	No	29.38	0.078	0.040	10.28	0.107	0.158
0	1	Yes	9.56	0.175	0.594	15.64	0.188	0.522
0	1	No	0.58	0.109	0.820	4.90	0.132	0.514
1	0	Yes	17.07	0.608	0.089	5.92	0.513	0.190
1	0	No	1.51	0.643	0.048	4.22	0.500	0.188
1	1	Yes	1.52	0.703	0.822	31.00	0.508	0.502
1	0	No	0.27	0.663	0.719	11.25	0.507	0.506

patterns of our benchmarks. (Section 6 describes the benchmarks and how they are run.) By combining the data from the table using probability methods, we find that on average, when the top bit is 0, so are 91.2% of the bits, and when this bit is 1, so are 63.7% of the bits.

*Floating point bit patterns.* Although not as common, a reasonable number of floating point operands also contain only a few bits of precision, for three main reasons: (1) the casting of integer values into floating point representation, such as happens when incrementing a floating point variable, (2) the casting of single precision numbers into double precision numbers by the hardware because there are no separate, single-precision registers and FUs, and (3) the common use of round numbers in many programs.

Hence, the Hamming distance of floating point numbers may also be approximated. Floating points that do not use all of their precision will have many trailing zeroes. On the other hand, numbers with full-precision have a 50% probability of any given bit being zero. The bits of a full-precision number can be considered random, so the Hamming distance between a full-precision number and anything else is about 50% of the bits. The Hamming distance is only reduced when two consecutive inputs have trailing zeros.

OR-ing of the bottom four bits of the operand is an effective means of determining which values do not use the full precision. Simply examining the least significant bit of the mantissa is not sufficient, because this will capture half of the full-precision numbers, in addition to the intended numbers that have trailing zeroes. But using four bits only misidentifies 1/16 of the full-precision numbers. We do not wish to use more than four bits, so as to maintain a fast circuit. Although we refer to an OR gate because it is more intuitive, a NOR gate is equally effective and slightly faster.

Table 35-1 describes the effectiveness of our approach. From Table 35-1 we can derive that 42.4% of floating point operands have zeroes in their bottom 4 bits, implying that 3.8% of these are full precision numbers that happen to have four zeroes ( $(100 - 42.4) / 15$ ) and that the remaining 38.6% do indeed have trailing zeroes ( $42.4 - 3.8$ ). This table also shows that, on average, when the bottom four bits are zero, 86.5% of the bits are zero.

### 4.3. A lightweight approach for operand steering

In section 4.2, the Hamming distance computations of Figure 35-2 are reduced to just the Hamming distances of the information bits of the operands; this section examines avoiding the computation of Figure 35-2 entirely. This is achieved by predetermining the FU assignment for any particular set of instruction operands – without comparison to previous values. Some additional definitions are useful:

**bit(operand):** The information bit of the operand.

**case( $I_j$ ):** The concatenation of  $bit(OP1(I_j))$  with  $bit(OP2(I_j))$ . *case* classifies the instructions into four possible tuples (00, 01, 10, 11).

- least:** The *case* with the lowest frequency, as found by examining Table 35-1 for all four cases, where the commutative and non-commutative rows are combined into one value.
- vector:** The concatenation of ( $\text{case}(I_1), \text{case}(I_2), \dots, \text{case}(I_{\text{Num}(I)})$ ). The size of *vector* is  $2 * \text{Num}(M)$ . If  $\text{Num}(I) < \text{Num}(M)$ , the remaining bit pairs of *vector* are set to the *least* case.

The insight behind an approach of just using present inputs without considering the previous ones is that, by distributing the various instruction cases across the available modules, subsequent instructions to that module are likely to belong to the same case, without needing to check the previous values. For example, if we consider a machine where  $\text{Num}(M) = 4$ , and with an equal probability of each of the four cases (00, 01, 10, and 11), it is logical to assign each of the cases to a separate module. In cycles when no more than one instruction of each case is present, this strategy will place them perfectly, even without checking previous values. When multiple instructions of the same case are present, however, the assignment will be non-ideal.

The algorithm is implemented as follows. During each cycle, *vector* is used as the input address to a look up table, or LUT. The output of that table encodes the assignment of the operations to modules of the given FU. Therefore, the LUT contains the assignment strategy. Although the algorithm is conceptually visualized as using an LUT, the actually implemented circuit may use combinational logic, a ROM, or another method.

The contents of the LUT are determined as follows. The information from Table 35-1 – along with new information from Table 35-2, which lists the probabilities of multiple instructions executing on the given FU type – is used to compute the probabilities of different input patterns. For instance, in the IALU, case 00 is by far the most common ( $40.11\% + 29.38\% = 69.49\%$ ), so we assign three of the modules as being likely to contain case 00, and we use the fourth module for all three other cases (our test machine has 4 modules). For floating point, case 11 is the most common ( $31.00\% + 11.25\% = 42.25\%$ ), but because it is unlikely that two modules will be needed at once (see Table 35-2), the best strategy is to first attempt to assign a unique case to each module.

Whenever the number of instructions of a particular case exceeds the number of modules reserved for that case, then it is necessary to place some

Table 35-2. Frequency that the functional unit uses a particular number of modules for a 4-way machine with 4 IALUs and 4 FPAUs. There is no  $\text{Num}(I) = 0$  column because we only consider cycles which use at least one module – the other case being unimportant to power consumption within a module (ignoring leakage).

	Num(I) = 1	Num(I) = 2	Num(I) = 3	Num(I) = 4
IALU	40.3%	36.2%	19.4%	4.2%
FPAU	90.2%	9.2%	0.5%	0.1%

of them in non-ideal modules. These overflow situations are dealt with in the order of their probability of occurring. The strategy for making non-ideal assignments is to greedily choose the module that is likely to incur the smallest cost.

#### 4.4. Operand swapping

We also propose a method for swapping operands without considering the previous inputs; the intuition of our method is as follows. The most power-consuming integer computations are those where the information bits for both operands fail to match the bits from the previous operands to that FU. There are four ways that this may occur: case 00 follows case 11, case 11 follows case 00, case 10 follows case 01, or case 01 follows case 10. In the last two of these, swapping the operands converts a worst-case situation into a best-case situation, assuming that the operation is commutative.

Therefore, we propose always swapping the operands for one of these cases. To minimize mismatches, the case to swap from should be the one that has the lower frequency of non-commutative instructions. Only non-commutative instructions are considered, because these are the ones that will not be flipped, and will therefore cause mismatches. Table 35-1 shows that for the IALU, the 4th row has a lower frequency than the 6th; for the FPAU, the 6th row is the smaller. Therefore, case 01 instructions will be swapped for the IALU, and case 10 instructions for the FPAU.

*Compiler-based swapping.* An alternative strategy for swapping operands is to perform it in software, by physically changing the machine instructions in the binary executable, using profiling. This approach is not particularly novel, but is studied so as to avoid the hardware cost of dynamic operand swapping. It is also instructional, in that our results show that the benefit of our hardware method is fairly independent of this transformation.

Compiler-based swapping has three advantages over hardware swapping. First, it avoids the overhead of hardware swapping. In fact, it offers some power improvement even when our hardware mechanism is not implemented. Second, the compiler can afford to count the full number of high bits in the operands, rather than approximating them by single bit values. For example, a “1+511” and a “511+1” operation both look like a case 00 to our hardware method. A compile-time method, however, can recognize the difference and swap when beneficial. Third, certain operations are commutable by the compiler but not by the hardware. An example is the “>” operation, which can become the “≤” operation when the operands are swapped. The compiler is able to change the opcode, but the current hardware strategy cannot.

But the compiler approach also has three main disadvantages. First, each instruction’s operands are either always swapped or always not. The decision is made based on the *average* number of high bits for the two operands. In contrast, the hardware technique captures the dynamic behavior of the

instruction. Hardware can choose to swap an instruction’s operands on one cycle, and not to swap them on a later cycle. Second, since the program must be profiled, performance will vary somewhat for different input patterns. Third, some instructions are not commutative in software. One example is the immediate add. While this operation is commutative, there is no way to specify its operand ordering in the machine language – the immediate value is always taken to be the second.

Since both hardware and compiler swapping have their advantages, the best results are achieved with both.

### 5. PRACTICAL CONSIDERATIONS

We now consider the costs of our method. The potential costs are (1) increased power and area due to additional buses and logic, and (2) increased cycle time due to more complex routing logic. In fact, additional buses are not needed and the increase in logic is not large.

In appraising the costs of the method, it is helpful to review how FUs are currently assigned by a superscalar processor. The most common methods are loosely based upon Tomasulo’s algorithm [8], where each FU type has its own *reservation station*, **RS**. On each cycle, those instructions whose operands become ready are removed from RS and assigned to the FUs on a first-come-first-serve basis. Figure 35-3 shows the basic hardware used in Tomasulo’s

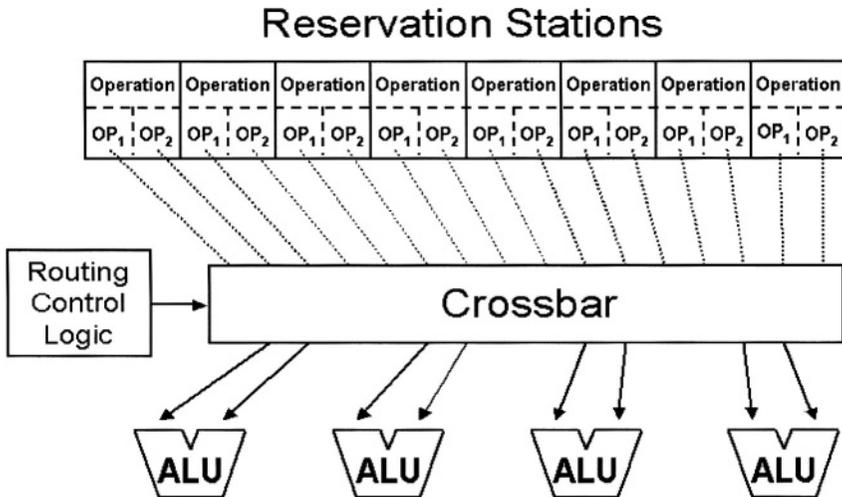


Figure 35-3. Typical Tomasulo hardware. On a particular cycle, 3 operations are shaded to show they are ready to execute. Dotted lines indicate inactive buses, while solid lines represent active ones. Operations indicate their readiness to the routing control logic, which in turn schedules them, in order, to the FUs. The key observation is that the crossbar implies that no new buses will be needed by our method.

algorithm. This algorithm allows instructions to execute out of order, so it requires routing logic and a crossbar. Since, on existing machines, most stations must already be able to map to any module, we do not need to add new wires.

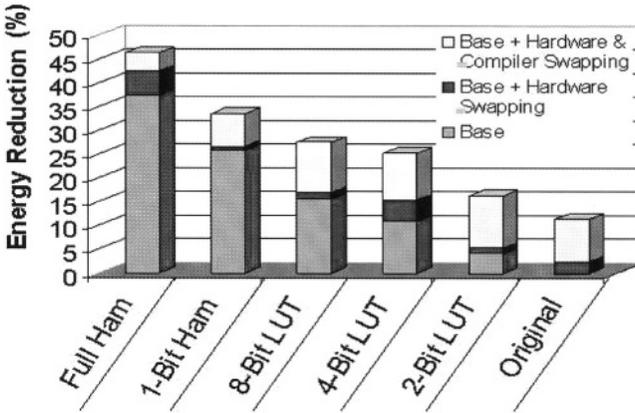
It is unavoidable that our method increases the complexity of the routing control logic, however. We replace the existing simple routing control logic with a not-as-simple LUT as described in Section 4.3. To make the LUT smaller and faster, we propose reducing the size of the vector. Since  $\text{Num}(I) < \text{Num}(M)$  for most cycles, it is reasonable to consider a smaller vector that may not always contain all of the instructions. Reducing the size of the vector makes the proposed hardware modification faster and therefore more attractive. In the results we show that a 4-bit vector yields good performance. With a 4-bit vector, our fully implemented algorithm for the IALU, on a machine with 8 entries in its RS, requires 58 small logic gates and 6 logic levels. With 32 entries, 130 gates and 8 logic levels are needed. This is a negligible fraction of the many thousands of gates present in the IALU. Therefore, the power and delay introduced are small.

A third issue is that the crossbar currently used in Tomasulo's algorithm does not allow for operand swapping. To perform operand swapping in hardware, it would be necessary to include additional wiring after the crossbar of Figure 35-3. This makes the compiler-based swapping methods more attractive.

## 6. EXPERIMENTAL RESULTS

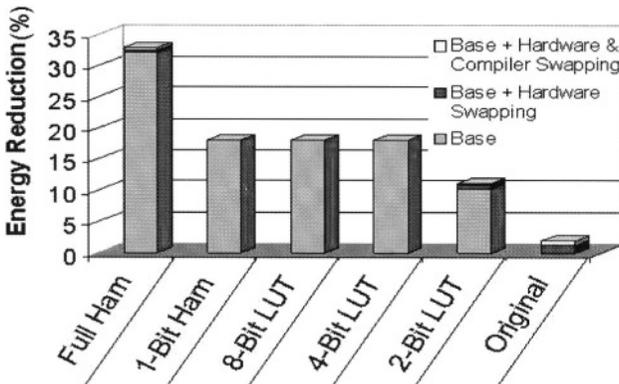
We have implemented the methodology described in section 4 using the *sim-outorder* simulator from the SimpleScalar 2.0 suite [5], with the default configuration of 4 IALUs, 4 FPAUs, 1 integer multiplier, and 1 floating point multiplier. SimpleScalar simulates a MIPS-like machine, with 32-bit integer registers and 64-bit floating point registers. The integer benchmarks used are: 88ksim, jpeg, li, go, compress, cc1, and perl. The floating point benchmarks are: apsi, applu, hydro2d, wave5, swim, mgrid, turb3d, and fpppp. The benchmarks are from SPEC 95 [7], and were run to completion on large input files.

Figure 35-4 displays the power reduction of different schemes, as a fraction of the total power consumption of the IALU. Similarly, Figure 35-5 displays the power reductions for the FPAU. Each bar is a stack of three values, so as to show the effect of operand swapping on the switching activity (which loosely approximates energy). Full Ham (section 4.1) and 1-bit Ham (section 4.2) are cost-prohibitive, but are included for comparison. Full Ham identifies the maximum possible improvement. 1-bit Ham is an upper bound on the improvement possible solely through the information bits. The 8 bit LUT represents the approach of section 3. The 4-bit and 2-bit LUTs represent shortened vectors, as considered in section 5. The Original column



Colour picture

Figure 35-4. Results for IALU.



Colour picture

Figure 35-5. Results for FPAU.

represents a first-come-first-serve assignment strategy. The gain for Original is not zero since swapping benefits it as well.

Figures 35-4 and 35-5 provides five insights. First, a 4-bit LUT is recommended, because its power savings are comparable to the upper bound (1-bit Ham), while being simple to implement. The improvement for the 4-bit LUT with hardware swapping is 18% for the FPAU and 17% for the IALU. With compiler swapping, it is 26%, for the IALU. If no swapping is provided by the hardware or the compiler, the still-rather-simple 8-bit LUT yields a very similar result to the 4-bit LUT with hardware swapping. Second, Figure 35-5 shows that the FPAU does not need operand swapping, due to differences between integers and floats. For integers, the majority of bits are usually the same as the information bit; for floating points, only an information bit of 0 has this property. Thus, for the IALU, a case 01 after a 10 causes most bits to switch; where as a case 01 after another 01 switches few bits. In

contrast, for the FPAU, a case 01 after a 10 switches 1/2 of the bits; where as a case 01 after another 01 still switches 1/4 of the bits. Third, the FPAU is less sensitive to the size of the vector, because the floating point unit is less heavily loaded (Table 35-2). Fourth, profile-based swapping is more effective than hardware-based swapping, because the swap decision is based on the entire value, as opposed to the information bits alone. In fact, “Base + Compiler Swapping” (not shown) is nearly as effective as “Base + Hardware + Compiler”. Fifth, implementing compiler swapping does not reduce the additional benefit of our hardware approach. Rather, the benefit of compiler swapping is slightly higher with an 8-bit LUT than it is for the original processor that has no hardware modifications.

Table 35-3 displays the bit patterns for multiplication data. This table, shows that 15.5% of floating point multiplications can be swapped from case 01 to case 10, certainly resulting in some additional power savings – though not quantifiable since we have no power model for multiplication.

Table 35-3. Bit patterns in multiplication data. (Multiplier power is related to how many 01 cases can become 10.)

Case	Integer			Floating Point		
	Frequency (%)	Operand 1 bit probability	Operand 2 bit probability	Frequency (%)	Operand 1 bit probability	Operand 2 bit probability
00	93.79	0.116	0.056	20.12	0.139	0.095
01	1.07	0.055	0.956	15.52	0.160	0.511
10	2.76	0.838	0.076	21.29	0.527	0.090
11	2.38	0.710	0.909	43.07	0.274	0.271

## 7. CONCLUSIONS

We present a method for dynamically assigning operations to functional units, so as to reduce power consumption for those units with duplicated modules. We also examined operand swapping, both in conjunction with our assignment algorithm, and on its own merits for non-duplicated functional units like the multiplier. Our results show that these approaches can reduce 17% of the IALU and 18% of the FPAU switching, with only a small hardware change. Compiler swapping increases the IALU gain to 26%.

## REFERENCES

1. M. Alidina, J. Monteiro, S. Devadas, and Papaefthymiou. “Precomputation-Based Sequential Logic Optimization for Low Power.” *IEEE Transactions on VLSI Systems*, Vol. 2, No. 4, pp. 426–436, April 1994.

2. L. Benini and G. D. Micheli. "Transformation and Synthesis of FSMs for Low Power Gated Clock Implementation." *IEEE Transactions on Computer Aided Design*, Vol. 15, No. 6, pp. 630–643, June 1996.
3. D. Brooks and M. Martonosi. "Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance." In *Proceedings of the 5th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 13–22, January 1999.
4. D. Brooks, V. Tiwari, and M. Martonosi. "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations." In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA)*, Vancouver, British Columbia, June 2000.
5. D. Burger and T. Austin. "The SimpleScalar Tool Set, Version 2.0." *Technical Report TR 1342*, University of Wisconsin, Madison, WI, June 1997.
6. J.-M. Chang and M. Pedram. "Module Assignment for Low Power." In *Proceedings of the European Conference on Design Automation (EDAC)*, pp. 376–381, September 1996.
7. S. P. E. Corporation. *The SPEC Benchmark Suites*. <http://www.spec.org/>.
8. J. Hennessy and D. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann, Palo Alto, CA, second edition, 1996.
9. M. Johnson, D. Somasekhar, and K. Roy. "Leakage Control with Efficient Use of Transistor Stacks in Single Threshold CMOS." In *Design Automation Conference (DAC)*, pp. 442–445, June 1999.
10. B. Klass, D. E. Thomas, H. Schmidt, and D. E. "Nagle. Modeling Inter-Instruction Energy Effects in a Digital Signal Processor." In *Power-Driven Microarchitecture Workshop, in Conjunction with ISCA*, June 1998.
11. L. Kruse, E. Schmidt, G. Jochenshar, and W. Nebel. "Lower and Upper Bounds on the Switching Activity in Scheduling Data Flow Graphs." In *Proceedings of the International Symposium on Low Power Design*, pp. 115–120, August 1999.
12. T. C. Lee, V. Tiwari, S. Malik, and M. Fuhita. "Power Analysis and Minimization Techniques for Embedded DSP Software." *IEEE Transactions on VLSI Systems*, March 1997.
13. R. Marculescu, D. Marculescu, and M. Pedram. "Sequence Compaction for Power Estimation: Theory and Practice." *IEEE Transactions on Computer Aided Design*, Vol. 18, No. 7, pp. 973–993, 1999.
14. J. Mermet and W. Nebel. *Low Power Design in Deep Submicron Electronics*. Kluwer Academic Publishers, Norwell, MA, 1997.
15. M. Pedram. "Power Minimization in IC Design: Principles and Applications." *ACM Transactions on Design Automation of Electronic Systems*, Vol. 1, No. 1, pp. 1–54, January 1996.
16. V. Tiwari, S. Malik, and P. Ashar. "Guarded Evaluation: Pushing Power Management to Logic Synthesis/Design." In *Proceedings of the ACM/IEEE International Symposium on Low Power Design*, pp. 139–142, April 1994.

*This page intentionally left blank*

## Chapter 36

# ENERGY-AWARE PARAMETER PASSING

M. Kandemir<sup>1</sup>, I. Kolcu<sup>2</sup> and W. Zhang<sup>1</sup>

<sup>1</sup> CSE Department, The Pennsylvania State University, University Park, PA 16802, USA;

<sup>2</sup> Computation Department, UMIST, Manchester M60 1QD, UK;

E-mail: {kandemir, wzhang}@cse.psu.edu

**Abstract.** We present an energy-aware parameter-passing strategy called on-demand parameter-passing. The objective of this strategy is to eliminate redundant actual parameter evaluations if the corresponding formal parameter in a subroutine is not used during execution. Our results indicate that on-demand parameter-passing is very successful in reducing energy consumption of large, multi-routine embedded applications at the expense of a slight implementation complexity.

**Key words:** parameter-passing, compilers, programming languages, energy-efficiency

## 1. INTRODUCTION

Our belief is that addressing ever-increasing energy consumption problem of integrated circuits must span multiple areas. While advances in circuit, architecture, OS, application, and compiler areas are promising, it might also be important to consider programming language support for low power. This issue is very critical because programming language defines the interface between application and the underlying execution environment. The types of optimizations that can be performed by the compiler and possible architectural hooks that can be exploited by the runtime system are also determined and/or controlled by the programming language. Unfortunately, to the best of our knowledge, there has not been a study so far for evaluating different language features from an energy consumption perspective.

Parameter-passing mechanisms are the ways in which parameters are transmitted to and/or from called subprograms [5]. Typically, each programming language supports a limited set of parameter-passing mechanisms. In C, one of the most popular languages in programming embedded systems, all parameter evaluations are done before the called subprogram starts to execute. This early parameter evaluation (i.e., early binding of formal parameters to actual parameters), while it is preferable from the ease of implementation viewpoint, can lead to redundant computation if the parameter in question is not used within the called subprogram.

In this study, we present an energy-aware parameter-passing mechanism that tries to eliminate this redundant computation when it is detected. The

proposed mechanism, called on-demand parameter-passing, computes value of an actual parameter if and only if the corresponding formal parameter is actually used in the subroutine. It achieves this by using compiler's help to postpone the computation of the value of the actual parameter to the point (in the subroutine code) where the corresponding formal parameter is actually used. It should be emphasized that our objective is not just to eliminate the computation of the value of the actual parameter but also all other computations that lead to the computation of that parameter value (if such computations are not used for anything else). Our work is complementary to the study in [3].

In Section 2, we summarize the important characteristics of the parameter-passing mechanisms used in C. In Section 3, we describe energy-aware parameter passing and discuss implementation issues. In Section 4, we introduce our simulation environment and present experimental results. Finally, in Section 5, we give our conclusions.

## 2. REVIEW OF PARAMETER PASSING MECHANISMS

Subroutines are the major programming language structures for enabling control and data abstraction [5]. The interface of the subroutine to the rest of the application code is indicated in its using subroutine name and its parameters. The parameters listed in the subprogram header are called formal parameters. Subprogram call statements must include the name of the subroutine and a list of parameters, called actual parameters, to be bound to the formal parameters in the subprogram header. In a typical implementation, at the subroutine invocation time, the actual parameters are computed and passed (copied) to formal parameters using the parameter access path. After the execution of the subroutine, depending on the parameter return path used, the values of the formal parameters are copied back to actual parameters. The parameter access path and parameter return path implementations depend on the parameter-passing mechanism adopted and discussed below for the C language.

In C, each subroutine parameter is passed using one of two mechanisms: call-by-value (CBV) and call-by-reference (CBR). In CBV (which is the default mechanism), when a parameter is passed, the value of the actual parameter is computed and copied to the formal parameter; that is, the parameter access path involves a value copy operation. This increases the storage overhead as the actual and formal parameters occupy different locations. During the execution of the subroutine, all references to the formal parameter uses only the location allocated for it; that is, the content of the location that holds the actual parameter is not modified. In fact, the said location remains unmodified even after the subroutine returns. In a sense, there is no parameter return path. The CBV mechanism is an excellent parameter-passing method for the cases that require only one-way communication (i.e., from caller to

callee). In comparison, in the CBR mechanism, the actual parameter and the formal parameter share the same location. At subroutine invocation time, a pointer to the actual parameter is passed to the subroutine. This parameter access path speeds up the subroutine invocation. However, during subroutine execution, each access to the formal parameter is slower as it requires a level of indirection.

### 3. ON-DEMAND PARAMETER PASSING

#### 3.1. Approach

As noted in the previous section, early evaluation of an unused formal parameter can lead to performance and energy loss. In this section, we describe an energy-efficient on-demand parameter-passing strategy. In this strategy, the value of an actual parameter is not computed unless it is necessary. Note that this not only eliminates the computation for the value of the parameter, but also all computations that lead to that value (and to nothing else). In developing such a strategy, we have two major objectives. First, if the parameter is not used in the called subroutine, we want to save energy as well as execution cycles. Second, if the parameter is actually used in the subroutine, we want to minimize any potential negative impact (of on-demand parameter-passing) on execution cycles.

We discuss our parameter-passing strategy using the control flow graph (CFG) shown in Figure 36-1. Suppose that this CFG belongs to a subroutine. It is assumed that  $x$ ,  $y$ ,  $z$ , and  $t$  are formal parameters and the corresponding actual parameters are costly to compute from the energy perspective. Therefore, if it is not necessary, we do not want to compute the actual parameter and perform pointer (in CBR) or data value (in CBV) passing. We also assume that  $x$  in Figure 36-1 denotes the use of the formal parameter  $x$  and similarly for other variables. It is assumed that these variables are not referenced in any other place in the subroutine. In this CFG, CBV or CBR strategies would compute the corresponding actual parameters and perform pointer/value passing before the execution of the subroutine starts. Our energy-conscious strategy, on the other hand, postpones computing the actual parameters until they are actually needed.

We start by observing that the CFG in Figure 36-1 has five different potential execution paths from start to end (denoted using I, II, III, IV, and V in the figure). However, it can be seen that not all formal parameters are used in all paths. Consequently, if we compute the value of an actual parameter before we start executing this subroutine and then execution takes a path which does not use the corresponding formal parameter, we would be wasting both energy and execution cycles. Instead, we can compute the actual parameter on-demand (i.e., only if it really needed). As an example, let us focus on the formal parameter  $t$ . As shown in the figure, this parameter is used only in path

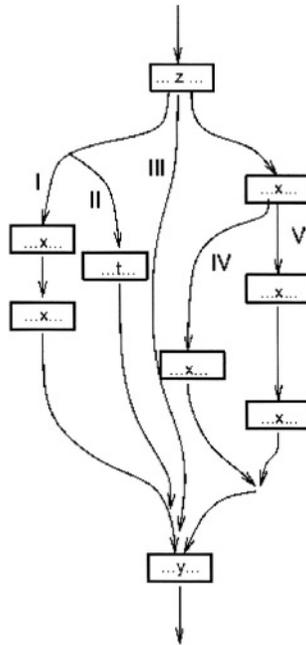


Figure 36-1. An example control flow graph (CFG).

II. So, it might be wiser to compute the corresponding actual parameter only along this path (e.g., just before the parameter is used). When we consider formal parameter  $z$ , however, we see that this parameter is used as soon as the subroutine is entered. Therefore, it needs to be computed when the subroutine is entered, which means that it does not benefit from on-demand parameter passing. A similar scenario occurs when we focus on formal parameter  $y$ . This parameter is used at the very end of the subroutine where all paths merge. Consequently, the corresponding actual parameter needs to be computed irrespective of the execution path taken by the subroutine. Here, we have two options. We can either compute that actual parameter as soon as the subroutine is entered; or, we can postpone it and compute just before it needs to be accessed (at the very end of the subroutine). Our current implementation uses the latter alternative for uniformity. Accesses to parameter  $x$  present a more interesting scenario. This variable is accessed in all but two paths. So, if the execution takes path II or III, the on-demand parameter-passing strategy can save energy. If the execution takes any other path, however, we need to compute the actual parameter. A straightforward implementation would compute the values of actual parameter in six different places (one per each use), which presents a code size overhead over CBV or CBR parameter-passing strategies (as both CBV and CBR perform a single evaluation per parameter). Therefore, to be fair in comparison, this increase in code size should also be accounted for.

### 3.2. Global variables

In this subsection, we show that global variables present a difficulty for on-demand parameter-passing. Consider the following subroutine fragment, assuming that it is called using `foo(c[index])`, where `c` is an array and `index` is a global variable:

```
foo(int x) {
    int y;
    ...
    if (...){
        ...
        index++;
        ...
        y=x+1;
        ...
    } else {
        ...
    }
}
```

It can be seen from this code fragment that a normal parameter-passing mechanism (CBR or CBV) and our on-demand parameter-passing strategy might generate different results depending on which value of the global variable `index` is used. In on-demand parameter evaluation, the actual parameter is computed just before the variable `x` is accessed in statement `y = x + 1`. Since computing the value of `c[index]` involves `index` which is modified within the subroutine (using statement `index++`) before the parameter computation is done, the value of `index` used in on-demand parameter-passing will be different from that used in CBR or CBV. This problem is called the global variable problem and can be addressed at least in three different ways:

- The value of `index` can be saved before the subroutine starts its execution. Then, in evaluating the actual parameter (just before `y = x + 1`), this saved value (instead of the current value of `index`) is used. In fact, this is the strategy adopted by some functional languages that use lazy evaluation [5]. These languages record the entire execution environment of the actual parameter in a data structure (called closure) and pass this data structure to the subroutine. When the subroutine needs to access the formal parameter, the corresponding actual parameter is computed using this closure. While this strategy might be acceptable from the performance perspective, it is not very useful from the energy viewpoint. This is because copying the execution environment in a data structure itself is a very energy-costly process (in some cases, it might even be costlier than computing the value of the actual parameter itself).
- During compilation, the compiler can analyze the code and detect whether

the scenario illustrated above really occurs. If it does, then the compiler computes the actual parameter when the subroutine is entered; that is, it does not use on-demand parameter-passing. In cases the compiler is not able to detect for sure whether this scenario occurs, it conservatively assumes that it does, and gives up on on-demand parameter-passing.

- This is similar to the previous solution. The difference is that when we detect that the scenario mentioned above occurs, instead of dropping the on-demand parameter-passing from consideration, we find the first statement along the path that assigns a new value to the global variable and execute the actual parameter evaluation just before that statement. For example, in the code fragment given above, this method performs the actual parameter computation just before the `index++` statement.

It should be mentioned that Algol introduced a parameter-passing strategy called call-by-name (CBN) [5]. When parameters are passed by call-by-name, the actual parameter is, in effect, textually substituted for the corresponding formal parameter in all its occurrences in the subprogram. In a sense, CBN also implements lazy binding. However, there is an important difference between our on-demand parameter-passing strategy and CBN. In CBN, the semantics of parameter passing is different from that of CBV and CBR. For example, in the subprogram fragment given above, the CBN mechanism uses the new value of `index` (i.e., the result of the `index++` statement) in computing the value of the actual parameter (and it is legal to do so). In fact, the whole idea behind CBN is to create such flexibilities where, in computing the values of actual parameters, the effects of the statements in the called subroutine can be taken into account. In contrast, our on-demand parameter-passing strategy does not change the semantics of CBV/CBR; it just tries to save energy and execution cycles when it is not necessary to compute the value of an actual parameter and the computations leading to it.

### 3.3. Multiple use of formal parameters

If a formal parameter is used multiple times along some path, this creates some problems as well as some opportunities for optimization. To illustrate this issue, we consider the uses of the parameter  $x$  in path I of Figure 36-1. It is easy to see that this parameter is used twice along this path. Obviously, computing the value of the corresponding actual parameter twice would waste energy as well as execution cycles. This problem is called the multiple uses problem. To address this problem, our strategy is to compute the value of the actual parameter in the first use, save this value in some location, and in the second access to  $x$ , use this saved value. Obviously, this strategy tries to reuse the previously computed values of actual parameters as much as possible.

Depending on the original parameter-passing mechanism used (CBV or CBR), we might need to perform slightly different actions for addressing the multiple uses problem. If the original mechanism is CBV, the first use

computes the value and stores it in a new location, and the remaining uses (on the same path) use that value. If, on the other hand, the original mechanism is CBR, the first use computes the value (if the actual parameter is an expression), stores it in a location, and creates a pointer which is subsequently used by the remaining uses (on the same path) to access the parameter. In either case, the original semantics of parameter-passing is maintained.

### 3.4. Problem formulation and solution

To perform analyses on a program, it is often necessary to build a control flow graph (CFG). Each statement in the program is a node in the control flow graph; if a statement can be followed by another statement in the program, there is an edge from the former to the latter in the CFG [1]. In this study, the CFG nodes are individual statements, whereas in most data-flow analysis problem, a CFG node contains a sequence of statements without branch. Note that CFG can contain one or more loops as well. In the following discussion, we use the terms node, block, and statement interchangeably. Each node in the CFG has a set of out-edges that lead to successor nodes, and in-edges that lead to predecessor nodes. The set  $pred(b)$  represents all predecessor nodes for statement  $b$  and the set  $succ(b)$  denotes all successors of  $b$  [1].

Data-flow analysis is used to collect data-flow information about program access patterns [1]. A data-flow analysis framework typically sets up and solves systems of equations that relate information at various points in a program (i.e., in various points in the corresponding CFG). Each point of interest in the code contributes a couple of equations to the overall system of equations. In our context, data-flow equations are used to decide the points at which the actual parameter evaluations for a given formal parameter need to be performed. We define a function called  $USE(\dots)$  such that  $USE(b, x)$  returns true if statement (basic block)  $b$  uses variable  $x$ ; otherwise, it returns false. Using the  $USE(\dots)$  function, we make the following definition:

$$Eval(b, x) = \begin{cases} \text{true, if } USE(b, x) \text{ and } \exists p \in pred(b) \ !Eval(p, x) \\ \text{true, if } !USE(b, x) \text{ and } \forall p \in pred(b) \ Eval(p, x) \\ \text{false, otherwise} \end{cases} \quad (1)$$

In this formulation,  $p$  denotes a predecessor statement for  $b$ . For a given statement  $b$  and formal parameter  $x$  where  $x$  is used in  $b$ ,  $Eval(b, x)$  returns true if and only if an actual parameter computation corresponding to the formal parameter  $x$  needs to be performed to access  $x$  in  $b$ . Such a parameter evaluation would be required if and only if there exists at least a path (coming to statement  $b$ ) along which the actual parameter evaluation in question has not been performed yet. As an example, suppose that statement  $b$  has two predecessors:  $p1$  and  $p2$ . Assume that a formal parameter  $x$  is used in  $b$  and  $p2$ , but not used in  $p1$ ; that is,  $USE(b, x)$ ,  $USE(p1, x)$ ,  $USE(p2, x)$  return true, false, and true, respectively. Assuming that no statement along the path starting from the beginning of the subroutine to  $p1$  and no statement along the path

starting from the beginning of the subroutine to  $p2$  use the parameter  $x$ . In this case,  $EVAL(p1, x)$  computes false and  $EVAL(p2, x)$  computes true. This indicates that the actual parameter evaluation has been performed along the path that contains  $p2$  but not along the path that contains  $p1$ . Since statement  $b$  can be accessed through  $p1$  or  $p2$ , we need to (conservatively) perform the evaluation of actual parameter in  $b$  (see Figure 36-2(a)); that is,  $EVAL(b, x)$  will return true. Suppose now that another formal parameter, all these three statements use  $y$ ,  $b$ ,  $p1$ , and  $p2$ . In this case, both  $EVAL(p1, y)$  and  $EVAL(p2, y)$  return true. Assuming that  $p1$  and  $p2$  are the only statements through which  $b$  can be reached,  $EVAL(b, y)$  will return false. Note that in this last scenario when execution reaches  $b$ , it is guaranteed that the value of the actual parameter (corresponding to  $y$ ) has been computed (see Figure 36-2(b)).

It should be noted that although we also compute the  $EVAL(...)$  function even for the statements that do not access the formal parameter in question, the meaning of this function in such cases is different. Specifically, the  $EVAL(...)$  function for such statements is used only for conveying the value (of  $EVAL(...)$ ) from the previous statements (if any) that use the formal parameter to the successor statements that use it. In technical terms, suppose that  $b'$  is a statement that does not access the formal parameter  $x$ , and  $p1, p2, \dots$ , are its predecessor statements. If there is at least an  $i$  such that  $1 \leq i \leq k$  and  $EVAL(p_i, x)$  is false, then  $EVAL(b', x)$  is set to false; otherwise, it is true.

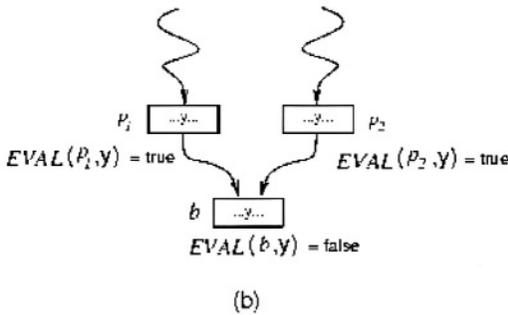
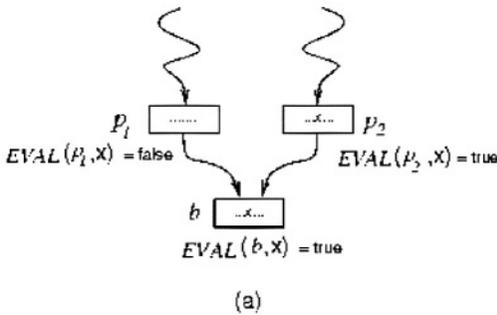


Figure 36-2. Two different scenarios for  $EVAL(...)$  computation.

It should be emphasized that, using  $EVAL(\dots)$ , we can also place actual parameter evaluation code into the subroutine. More specifically, for a given statement  $b$  and formal parameter  $x$ , we have four possibilities:

- $USE(b, x) = \text{true}$  and  $EVAL(b, x) = \text{true}$ : In this case, the actual parameter needs to be computed to access the formal parameter. This means that there exists at least one path from the beginning of the subroutine to  $b$ , which does not contain the actual parameter computation in question.
- $USE(b, x) = \text{true}$  and  $EVAL(b, x) = \text{false}$ : This case means that the statement  $b$  needs to access the formal parameter  $x$  and the value of the corresponding actual parameter has been computed earlier. Consequently, it does not need to be recomputed; instead, it can be used from the location where it has been stored.
- $USE(b, x) = \text{false}$  and  $EVAL(b, x) = \text{false}$ : In this case, the statement  $b$  does not use  $x$  and is not involved in the computation of the value of the corresponding actual parameter.
- $USE(b, x) = \text{false}$  and  $EVAL(b, x) = \text{true}$ : This case is the same as the previous one as far as inserting the actual parameter evaluation code is concerned. No action is performed.

It should be noted that the  $EVAL(\dots)$  function can be computed in a single traversal over the CFG of the subroutine. The evaluation starts with the header statement  $h$ , assuming  $EVAL(h, x) = \phi$  for each formal parameter  $x$ . It then visits the statements in the CFG one-by-one. A statement is visited if and only if all of its predecessors have already been visited and their  $EVAL(\dots)$  functions have been computed. These values are used in computing the value of the  $EVAL(\dots)$  function of the current statement using the expression (1) given above. While it is possible to compute the  $EVAL(\dots)$  function of all variables simultaneously in a single traversal of the CFG, our current implementation performs a separate traversal for each variable. This is a viable option as the number of formal parameters for a given subroutine is generally a small number.

We now give an example to explain how our approach handles a given formal parameter. Consider the CFG in Figure 36-3(a), assuming that  $b1$ ,  $b2$ ,  $b3$ ,  $b4$ ,  $b5$ , and  $b6$  are the only statements that use formal parameter  $x$ . We start by observing that  $EVAL(b1, x)$  should be true as there is no way that the value of the actual parameter might be required before reaching  $b1$  (along the path that leads to  $b1$ ). Having computed  $EVAL(b1, x)$  as true, it is easy to see that  $EVAL(b2, x)$  should be false as  $b1$  is the only predecessor to  $b2$ . Informally, what this means is that, since we compute the value of the actual parameter in  $b1$ , we do not need to re-compute it in  $b2$ . In a similar fashion, it can easily be seen that  $EVAL(b3, x)$ ,  $EVAL(b4, x)$ ,  $EVAL(b6, x)$  should be true, false, and false. The statement  $b5$  presents an interesting case. Since this statement can be reached through two different paths (shown as I and II in the figure), in deciding what  $EVAL(b5, x)$  should be, we need to consider both the paths. If  $b5$  is reached through  $b3$ , we can see that no actual

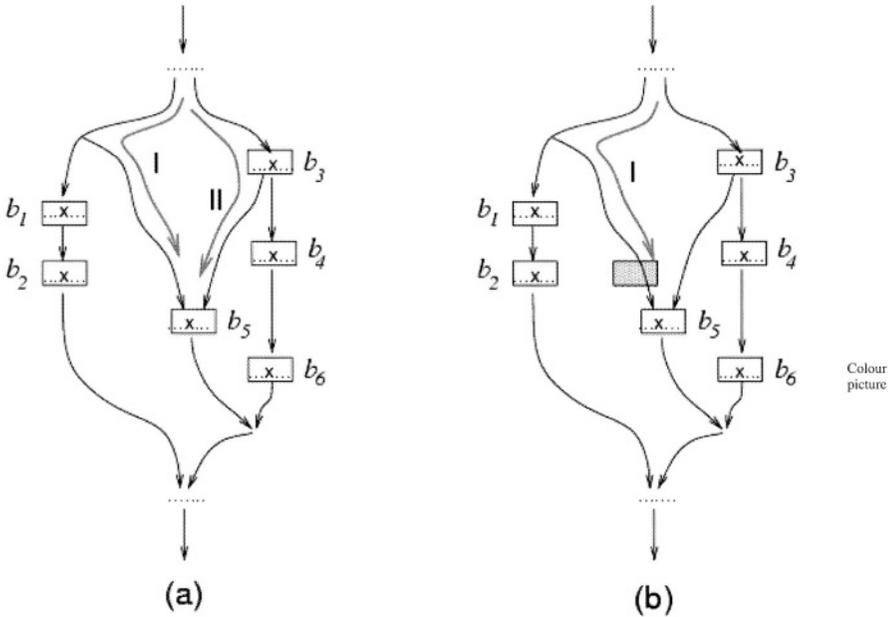


Figure 36-3. (a) An example CFG. (b) Inserting a basic block.

parameter computation (in  $b_5$ ) is necessary. If, however, it is reached through path I, we need to compute the value of the actual parameter. Consequently, we conservatively determine that  $EVAL(b_5, x)$  is true; that is, the value of the actual parameter should be computed before the formal parameter is accessed in  $b_5$ .

### 3.5. Additional optimization

In data-flow analysis, we generally accept safe (or conservative) solutions which might, in some cases, lead to inefficiencies. In our case, the  $EVAL(\dots)$  computation algorithm presented in the previous subsection might re-compute the value of an actual parameter more than once when different execution paths merge at some CFG point. For example, consider the statement  $b_5$  in Figure 36-3(a). If, during execution, path II is taken, then the value of the actual parameter corresponding to  $x$  will be computed twice, wasting energy as well as execution cycles. In this work, we consider two different methods to handle this problem. In the first method, called block insertion, we create a new basic block and put the actual parameter computation code there. Figure 36-3(b) illustrates this solution, which creates a new block (shown shaded) and inserts it along path I just before the statement  $b_5$  is reached. The actual parameter evaluation is performed in this new basic block and the statement  $b_5$  does not perform that evaluation. In this way, we guarantee that when  $b_5$

is reached, the actual parameter has already been computed; so, *b5* can just use the value.

The second method, called path control, is based on the idea of using a variable (or a set of variables) to determine at runtime which path is being executed. For example, assume that we use a variable *p* to determine which path (leading to *b5*) is being executed. Without loss of generality, we can assume that if path I is taken *p* is assigned 1, otherwise *p* is set to zero. Under this method, when *b5* is reached, we can check the value of *p*, and depending on its value, we perform actual parameter value evaluation or not. It should be noted that, as compared to the first method, this method results in a smaller executable size (in general); but, it might lead to a higher execution time due to comparison operation.

#### 4. EXPERIMENTAL RESULTS

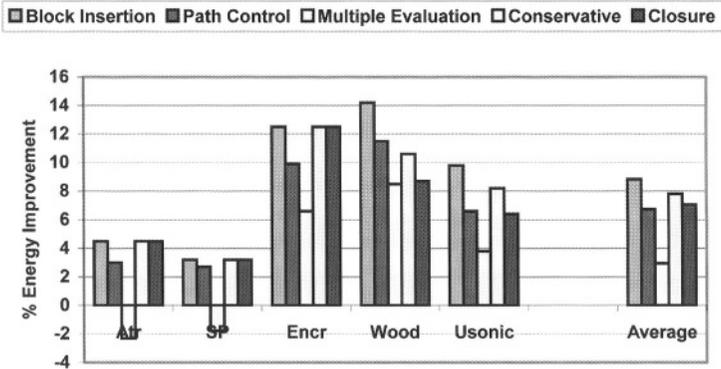
To test the effectiveness of our on-demand parameter-passing in reducing energy consumption, we conducted experiments with five real-life applications from different application domains: Atr(network address translation), SP(all-nodes shortest path algorithm), Encr(digital signature for security), Wood(color-based surface inspection method), and Usonic (feature-based estimation algorithm).

To evaluate the benefits due to on-demand parameter-passing, we modified SUIF [2] to implement different parameter-passing strategies. Our on-demand parameter-passing strategy took less than 1200 lines of C++ code to implement. We then modified the Shade tool-set to collect data about the performance and energy behavior of applications. Shade is an execution-driven ISA (instruction set architecture) and cache memory simulator for Sparc architectures. We simulated an architecture composed of a Sparc-Ilep based core processor (100 MHz), data (8 KB, 2-way) and instruction (16 KB, 2-way) caches, and a banked memory architecture (using SRAM banks). The simulator outputs the energy consumptions (dynamic energy only) in these components and overall application execution time. In particular, it simulates the parameter-passing activity in detail to collect data for comparing different parameter-passing strategies. For computing the energy expended in caches and SRAM banks, the model given by Kamble and Ghose [4] is used. The energy consumed in the processor core is estimated by counting the number of instructions of each type and multiplying the count by the base energy consumption of the corresponding instruction. The base energy consumption of different instruction types is obtained using a customized version of the simulator presented in [6]. All energy numbers have been computed for 0.10 micron, 1 V technology.

Figure 36-4 and Figure 36-5 show the percentage improvements in energy consumption and execution cycles, respectively, when different on-demand

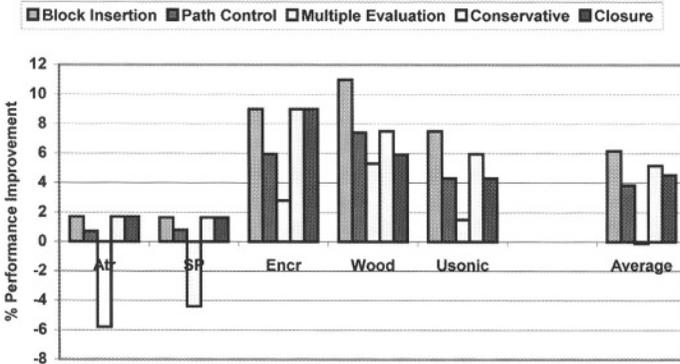
parameter-passing strategies are used. We performed experiments with five different on-demand parameter-passing strategies:

- **Block Insertion (Default):** This is our default on-demand parameter-passing strategy. In this strategy, the multiple uses problem is handled using block insertion. Also, if there is a global variable problem (i.e., the case where a global variable used as part of the actual parameter is assigned in the subroutine before the corresponding formal parameter is assigned in the subroutine before the corresponding formal parameter is reached), this strategy computes the value of the actual parameter just before the global variable assignment.
- **Path Control:** While this strategy handles the global variable problem in the same fashion as the previous one, the multiple uses problem is handled using path control (i.e., using variables to keep track of dynamic execution paths).
- **Multiple Evaluations:** This version also handles the global variable problem



Colour picture

Figure 361-4. Percentage improvement in energy.



Colour picture

Figure 36-5. Percentage improvement in execution cycles.

the same way. However, each access to a formal parameter invokes a fresh computation of the value of the actual parameter.

- **Conservative:** In this strategy, the multiple uses problem is resolved as in the case of the Block Insertion strategy. However, if the global variable problem occurs, this strategy drops on-demand parameter-passing and uses the original parameter-passing strategy.
- **Closure:** This is similar to the previous strategy except that in handling the global variable problem, it uses the closure-based mechanism discussed earlier in Section 3. It should be mentioned that this strategy passes a parameter as the first (default) strategy if the global variable problem does not occur.

We observe from Figure 36-4 that the default on-demand strategy improves energy consumption by 8.78%, on the average. It is most successful in applications Wood and Encr where there is a relatively large difference between the average number of parameters and the average number of active parameters. The Path Control and Multiple Evaluation strategies, on the other hand, achieve less saving in energy. In fact, in two applications, Atr and SP, evaluating the same actual parameter more than once results in energy loss (as compared to the case without on-demand parameter-passing).

The reason that it is relatively successful in remaining applications is the fact that in many subroutine executions the paths where multiple actual parameter evaluations occur are not taken at runtime. The reason that the Path Control strategy is not as successful as the Block Insertion strategy is the extra energy spent in maintaining the values of the path variables and in executing comparison operations. The percentage (average) improvements due to Path Control and Multiple Evaluation strategies are 6.68% and 2.98%, respectively. Conservative and Closure generate the same energy behavior as the default strategy in our three applications. This is because in these applications, the global variable problem does not occur. In Usonic and Wood, on the other hand, the default strategy outperforms Conservative and Closure.

The trends in Figure 36-5 are similar to those illustrated in Figure 36-4. This is not surprising as eliminating redundant computation saves both energy and execution cycles. It should be stressed, however, that the savings in execution cycles are lower than those in energy consumption. This is due to the fact that the processor we consider can execute multiple instructions in the same cycle; consequently, some of the redundant computation can be hidden (which makes the base case – without on-demand parameter-passing – appear stronger). However, unlike execution cycles, it is not possible to hide energy consumption.

## 5. CONCLUSIONS

In this work, we studied the possibility of modifying the parameter-passing mechanism of the language with some help from compiler. Using a set of

five real-life applications and a custom simulator, we investigated the energy and performance impact of an on-demand parameter-passing strategy. In this strategy, the value of an actual parameter is not computed if the corresponding formal parameter is not used within the subroutine. Our experiments show that one implementation of on-demand parameter-passing results in energy savings ranging from 4.42% to 14.11%, averaging in 8.78%.

## REFERENCES

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, 1986.
2. S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and A. W. Lim. "An Overview of a Compiler for Scalable Parallel Machines." In *Proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing*, August, 1993.
3. E.-Y. Chung, L. Benini, and D. De Micheli. "Energy-Efficient Source Code Transformation Based on Value Profiling." In *Proceedings of the 1st Workshop on Compilers and Operating Systems for Low Power*, Philadelphia, PA, 2000.
4. M. Kamble and K. Ghose. "Analytical Energy Dissipation Models for Low Power Caches." In *Proceedings of the International Symposium on Low Power Electronics and Design*, p. 143, August 1997.
5. R. W. Sebesta. *Concepts of Programming Languages*, Addison-Wesley Publishing, 2001.
6. N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. Y. Kim, and W. Ye. "Energy-Driven Integrated Hardware-Software Optimizations Using SimplePower." In *Proceedings of The International Symposium on Computer Architecture*, June 2000.

# LOW ENERGY ASSOCIATIVE DATA CACHES FOR EMBEDDED SYSTEMS

Dan Nicolaescu, Alex Veidenbaum, and Alex Nicolau

*Center for Embedded Computer Systems, University of California, Irvine, USA*

**Abstract.** Modern embedded processors use data caches with higher and higher degrees of associativity in order to increase performance. A set-associative data cache consumes a significant fraction of the total energy budget in such embedded processors. This paper describes a technique for reducing the D-cache energy consumption and shows its impact on energy consumption and performance of an embedded processor. The technique utilizes cache line address locality to determine (rather than predict) the cache way prior to the cache access. It thus allows only the desired way to be accessed for both tags and data. The proposed mechanism is shown to reduce the average L1 data cache energy consumption when running the MiBench embedded benchmark suite for 8, 16 and 32-way set-associate caches by, respectively, an average of 66%, 72% and 76%. The absolute energy savings from this technique increase significantly with associativity. The design has no impact on performance and, given that it does not have mis-prediction penalties, it does not introduce any new non-deterministic behavior in program execution.

**Key words:** set associative, data cache, energy consumption, low power

## 1. INTRODUCTION

A data caches is an important component of a modern embedded processor, indispensable for achieving high performance. Until recently most embedded processors did not have a cache or had direct-mapped caches, but today there's a growing trend to increase the level of associativity in order to further improve the system performance. For example, Transmeta's Crusoe [7] and Motorola's MPC7450 [8] have 8-way set associative caches and Intel's XScale has 32-way set associative caches.

Unfortunately the energy consumption of set-associative caches adds to an already tight energy budget of an embedded processor.

In a set-associative cache the data store access is started at the same time as the tag store access. When a cache access is initiated the way containing the requested cache line is not known. Thus all the cache ways in a set are accessed in parallel. The parallel lookup is an inherently inefficient mechanism from the point of view of energy consumption, but very important for not increasing the cache latency. The energy consumption per cache access grows with the increase in associativity.

Several approaches, both hardware and software, have been proposed to reduce the energy consumption of set-associative caches.

A phased cache [4] avoids the associative lookup to the data store by first accessing the tag store and only accessing the desired way after the tag access completes and returns the correct way for the data store. This technique has the undesirable consequence of increasing the cache access latency and has a significant impact on performance.

A way-prediction scheme [4] uses a predictor with an entry for each set in the cache. A predictor entry stores the most recently used way for the cache set and only the predicted way is accessed. In case of an incorrect prediction the access is replayed, accessing all the cache ways in parallel and resulting in additional energy consumption, extra latency and increased complexity of the instruction issue logic. Also, given the size of this predictor, it is likely to increase the cache latency even for correct predictions.

Way prediction for I-caches was described in [6] and [11], but these mechanisms are not as applicable to D-caches.

A mixed hardware-software approach was presented in [12]. Tag checks are avoided by having the compiler output special load/store instructions that use the tags from a previous load. This approach changes the compiler, the ISA and adds extra hardware.

This paper proposes a mechanism that *determines*, rather than predicts, the cache way containing the desired data before starting an access (called *way determination* from now on). Knowing the way allows the cache controller to only access one cache way, thus saving energy. The approach is based on the observation that cache line address locality is high. That is, a line address issued to the cache is very likely to have been issued in the recent past. This locality can be exploited by a device that stores cache line address/way pairs and is accessed prior to cache access. This paper shows that such a device can be implemented effectively.

## 2. WAY DETERMINATION

Way determination can be performed by a Way Determination Unit (*WDU*) that exploits the high line address locality. The *WDU* is accessed prior to cache access and supplies the way number to use as shown in Figure 37-1.

The *WDU* records previously seen cache line addresses and their way number. An address issued to the cache is first sent to *WDU*. If the *WDU* contains an entry with this address the determination is made and only the supplied cache way is accessed for both tags and data. Since the address was previously seen it is not a prediction and is always correct. If the *WDU* does not have the information for the requested address, a cache lookup is performed with all the ways accessed in parallel. The missing address is added to the *WDU* and the cache controller supplied way number is recorded for it.

Because of the high data cache address locality the number of entries in

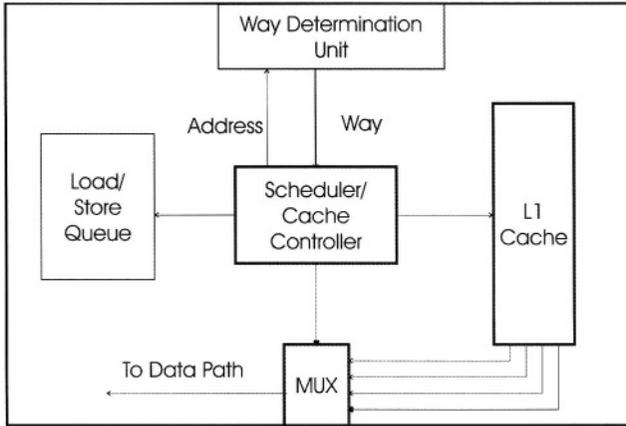


Figure 37-1. Data cache hierarchy with way determination.

the WDU can be small, thus allowing fast access and low energy overhead. WDU lookup can be done in parallel with load/store queues access as it has about the same latency. This should add no extra latency to the cache access.

The way determination system proposed here is based on access history. It has an energy penalty on misses similar to the mis-prediction penalty in a predictor. But it doesn't have the performance penalty of a mis-prediction.

### 3. THE WAY DETERMINATION UNIT DESIGN

The WDU, as shown in Figure 37-2, is a small, cache-like structure. Each entry is an address/way pair plus a valid bit. The tag part is fully associative and is accessed by a cache line address. The address is compared to a tag on lookup to guarantee the correct way number.

There are two types of WDU access: lookup and update. The lookup is cache-like: given a cache line address as input, the WDU returns a way number for a matching tag if the entry is valid. Updates happen on D-cache misses or WDU miss and cache hit. On a miss the WDU entry is immediately allocated and its way number is recorded when the cache controller determines it. If there are no free entries in the WDU the new entry replaces the oldest entry in the WDU. Our initial implementation uses a FIFO entry pointed to by the modulo counter.

One other issue the WDU design needs to address is coherence: when a line is replaced or invalidated in the L1 cache the WDU needs to be checked for the matching entry. The WDU entry can be made invalid. Another possible approach is to allow the access to proceed using the WDU-supplied way and a cache miss to occur when the cache tag access is performed. The way accessed was the only place the line could have been found. The WDU can

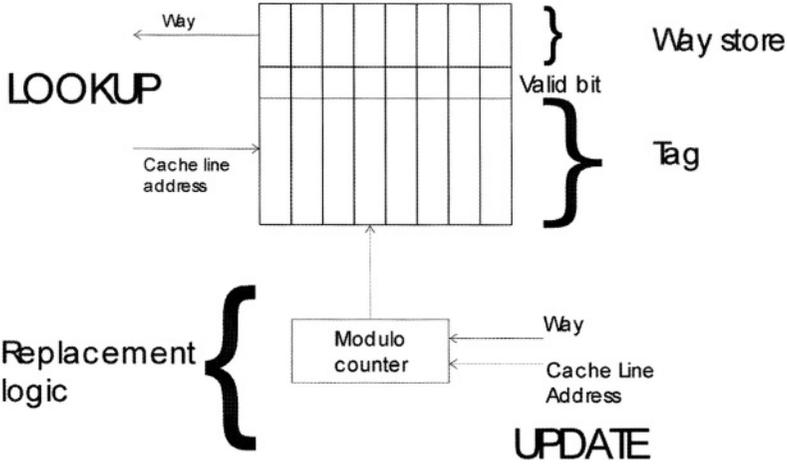


Figure 37-2. Way determination unit.

be updated when a line is allocated again. This is the approach used in the design presented here.

#### 4. EXPERIMENTAL SETUP

To evaluate the WDU design, the Wattch version 1.02 simulator [1] was augmented with a model for the WDU. Based on SimpleScalar [2], Wattch is a simulator for a superscalar processor that can do simulate the energy consumption of all major componets of a CPU. The CMOS process parameters for the simulated architecture are 400 MHz clock and 0.18 μm feature size.

The processor modeled uses a memory and cache organization based on XScale [5]: 32 KB data and instruction L1 caches with 32 byte lines and 1 cycles latency, no L2 cache, 50 cycle main memory access latency. The machine is in-order, it has a load/store queue with 32 entries. The machine is 2-issue, it has one of each of the following units: integer unit, floating point unit and multiplication/division unit, all with 1 cycle latency. The branch predictor is bimodal and has 128 entries. The instruction and data TLBs are fully associative and have 32 entries.

##### 4.1. The WDU energy consumption model

The WDU tags and way storage are modeled using a Wattch model for a fully associative cache. The processor modeled is 32 bit and has a virtually indexed L1 data cache with 32 byte lines, so the WDU tags are  $32 - 5 = 27$  bits wide, and the data store is 1, 2, 3, 4 or 5 bits wide for a 2, 4, 8 or

32-way set associative L1, respectively. The energy consumption of the modulo counter is insignificant compared to the rest of the WDU. The energy consumption model takes into account the energy consumed by the different units when idle.

For a processor with a physically tagged cache the size of the WDU is substantially smaller and so would be the energy consumption of a WDU for such an architecture.

Cacti3 [10] has been used to model and check the timing parameters of the WDU in the desired technology.

## 4.2. Benchmarks

MiBench [3] is a publicly available benchmark suite designed to be representative for several embedded system domains. The benchmarks are divided in six categories targeting different parts of the embedded systems market. The suites are: Automotive and Industrial Control (basicmath, bitcount, susan (edges, corners and smoothing)), Consumer Devices (jpeg encode and decode, lame, tiff2bw, tiff2rgba, tiffdither, tiffmedian, typeset), Office Automation (ghostscript, ispell, stringsearch), Networking (dijkstra, patricia), Security (blowfish encode and decode, pgp sign and verify, rijndael encode and decode, sha) and Telecommunications (CRC32, FFT direct and inverse, adpcm encode and decode, gsm encode and decode).

All the benchmarks were compiled with the `-O3` compiler flag and were simulated to completion using the “large” input set. Various cache associativities and WDU sizes have been simulated, all the other processor parameters were kept constant during this exploration.

## 5. PERFORMANCE EVALUATION

Figure 37-3 shows the percentage of load/store instructions for which a 8, 16, 32 or 64-entry WDU can determine the correct cache way.

An 8-entry WDU can determine the way for between 51 and 98% of the load/store instructions, with an average of 82%. With few exceptions (susan\_s, tiff2bw, tiff2rgba, pgp, adpcm, gsm\_u) for the majority of benchmarks increasing the WDU size to 16 results in a significant improvement in the number of instructions with way determination. The increase from 16 to 32 entries only improves the performance for a few benchmarks, and the increase from 32 to 64 for even fewer benchmarks.

Figure 37-4 shows the percentage data cache energy consumption savings for a 32-way cache when using an 8, 16, 32 or 64-entry WDUs. For space and legibility reasons all other results will only show averages, the complete set of results can be found in [9].

A summary of the average number of instructions for which way determination worked for 2, 4, 8, 16 and 32-way set-associative L1 data cache and

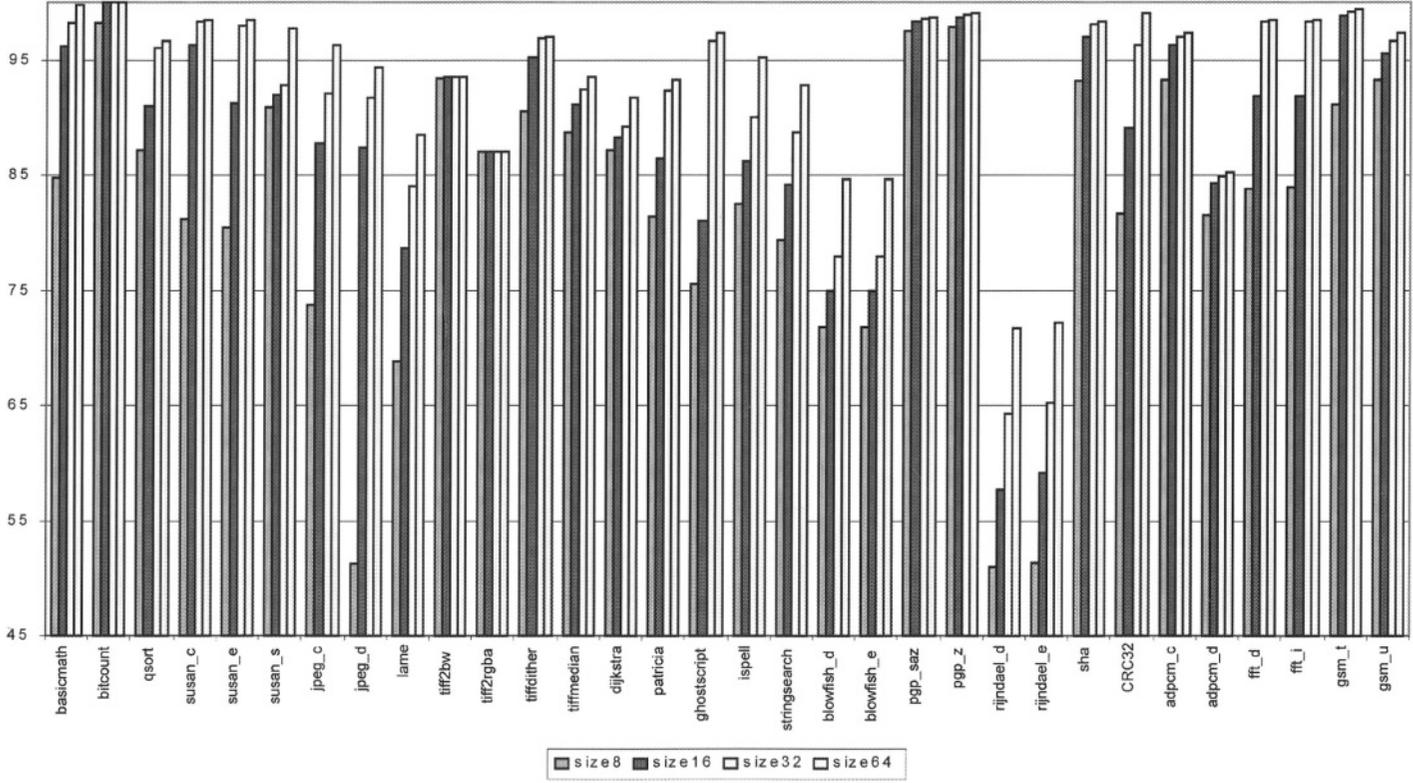


Figure 37-3. Percentage load/store instructions “covered” by way determination for a 32-way set associative cache.

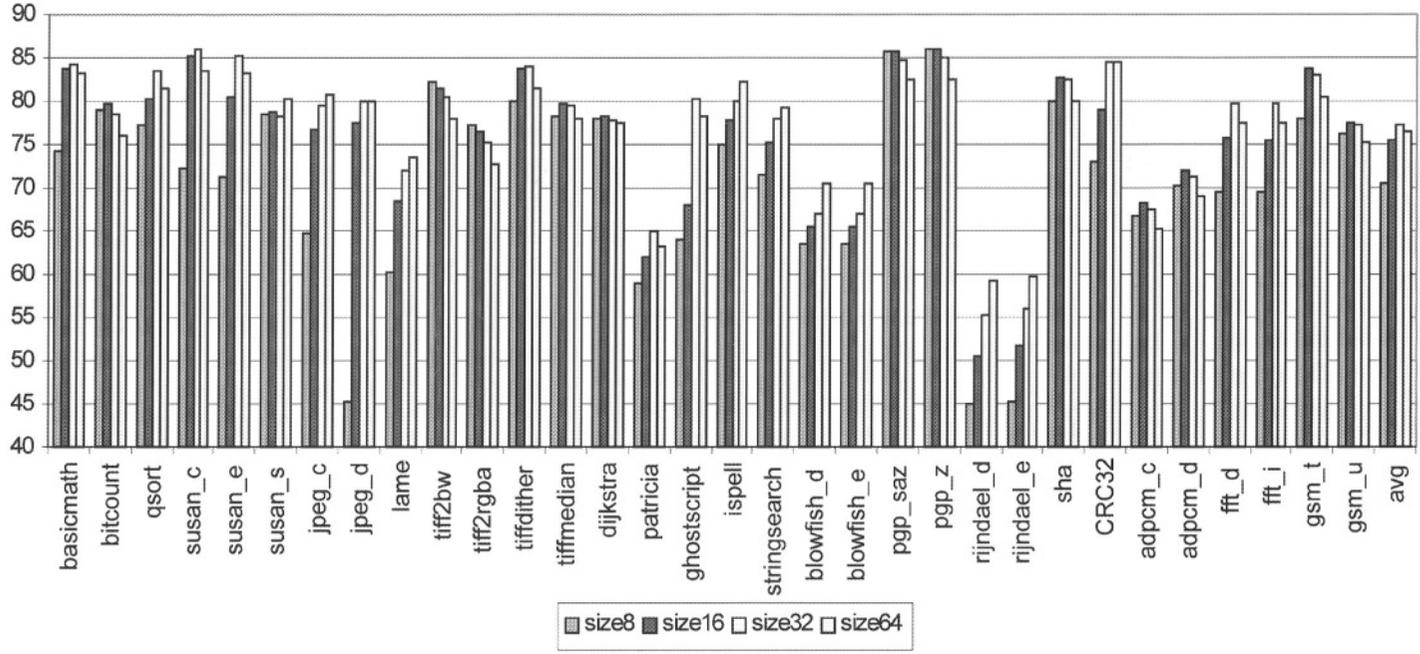


Figure 37-4. Percentage data cache energy consumption reduction for a 32-way set associative cache using different WDU sizes.

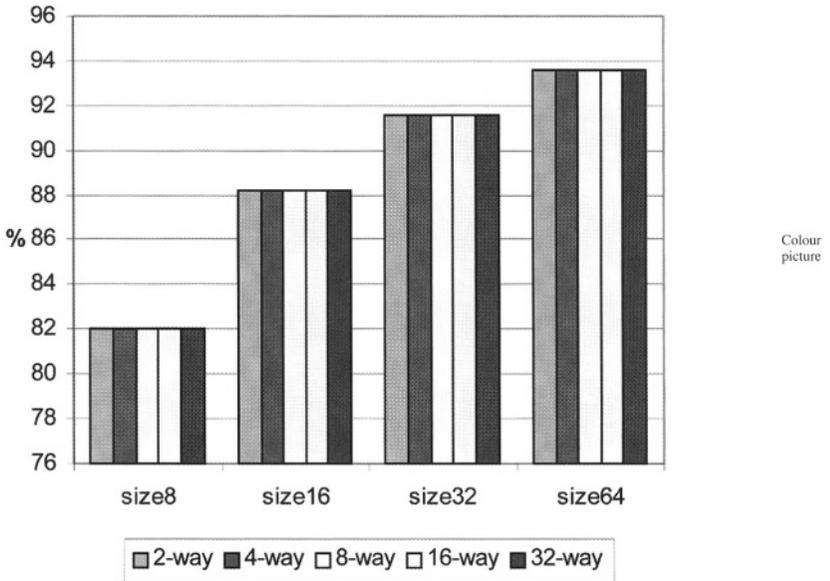
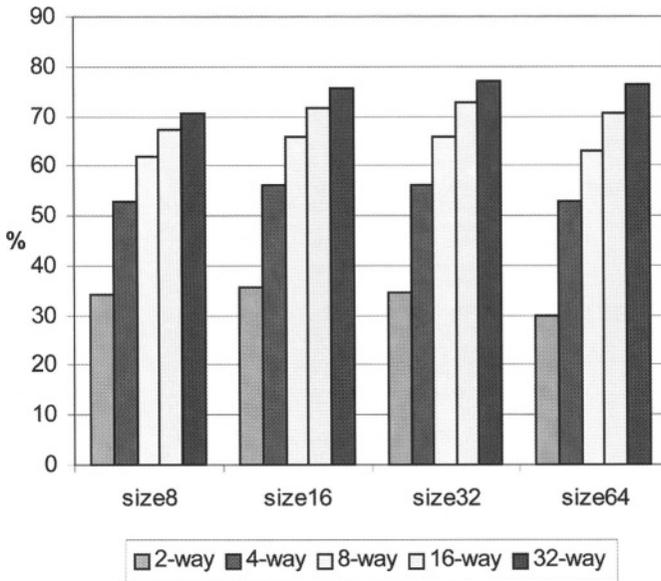


Figure 37-5. Average percented load/store instructions “covered” by way determination.

8, 16, 32 and 64-entry WDU is presented in Figure 37-5. It is remarkable that the WDU detects a similar number of instructions independent of the L1 cache associativity. Increasing the WDU size from 8 to 16 produces the highest increase in the percentage of instructions with way determination, from 82% to 88%. The corresponding values for a 32 and 64-entry WDU are 91% and 93%.

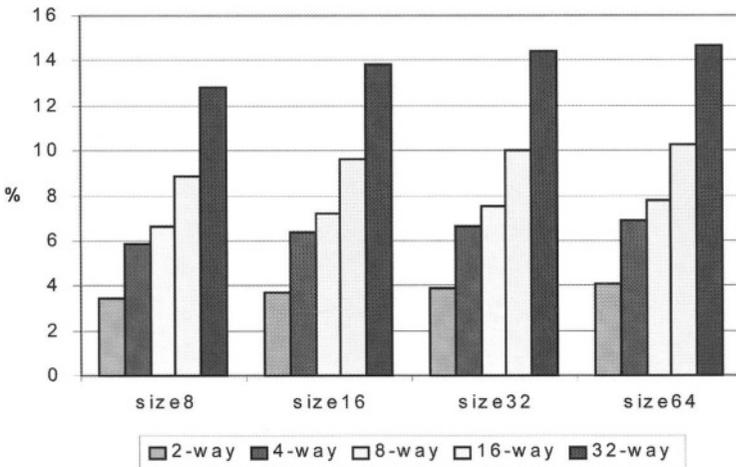
Figure 37-6 shows the average data cache energy consumption savings for the MiBench benchmark suite due to using the WDU, compared to a system that does not have a WDU. When computing the energy consumption savings the WDU energy consumption is added to the D-cache energy consumption. For all the associativities studied the 16-entry WDU has the best implementation cost/energy savings ratio. It’s average D-cache energy consumption savings of 36%, 56%, 66%, 72% and 76% for, respectively, a 2, 4, 8, 16 and 32-way set associative cache are within 1% of the energy consumption savings of a 32-entry WDU for a given associativity. The even smaller 8-entry WDU is within at most 3% of the best case. For the 64-entry WDU the WDU energy consumption overhead becomes higher than the additional energy savings due to the increased number of WDU entries, so the 64-entry WDU performs worse than the 32-entry one for a given associativity.

Figure 37-7 shows the percentage of total processor energy consumption reduction when using a WDU. For a 16-entry WDU the energy consumption savings are 3.73%, 6.37%, 7.21%, 9.59% and 13.86% for, respectively a 2, 4, 8, 16 and 32-way set associative L1 data cache. The total processor energy



Colour picture

Figure 37-6. Average percentage D-cache energy consumption reduction.



Colour picture

Figure 37-7. Total processor energy consumption reduction.

savings are greater for higher levels of associativity due to the increased D-cache energy consumption savings and to the increased share of the D-cache energy consumption in the total energy consumption budget.

The energy consumption savings for a data cache system using a WDU varies significantly with the associativity of the data cache. Figure 37-8 shows the data cache energy consumption savings when using a 32-entry WDU with

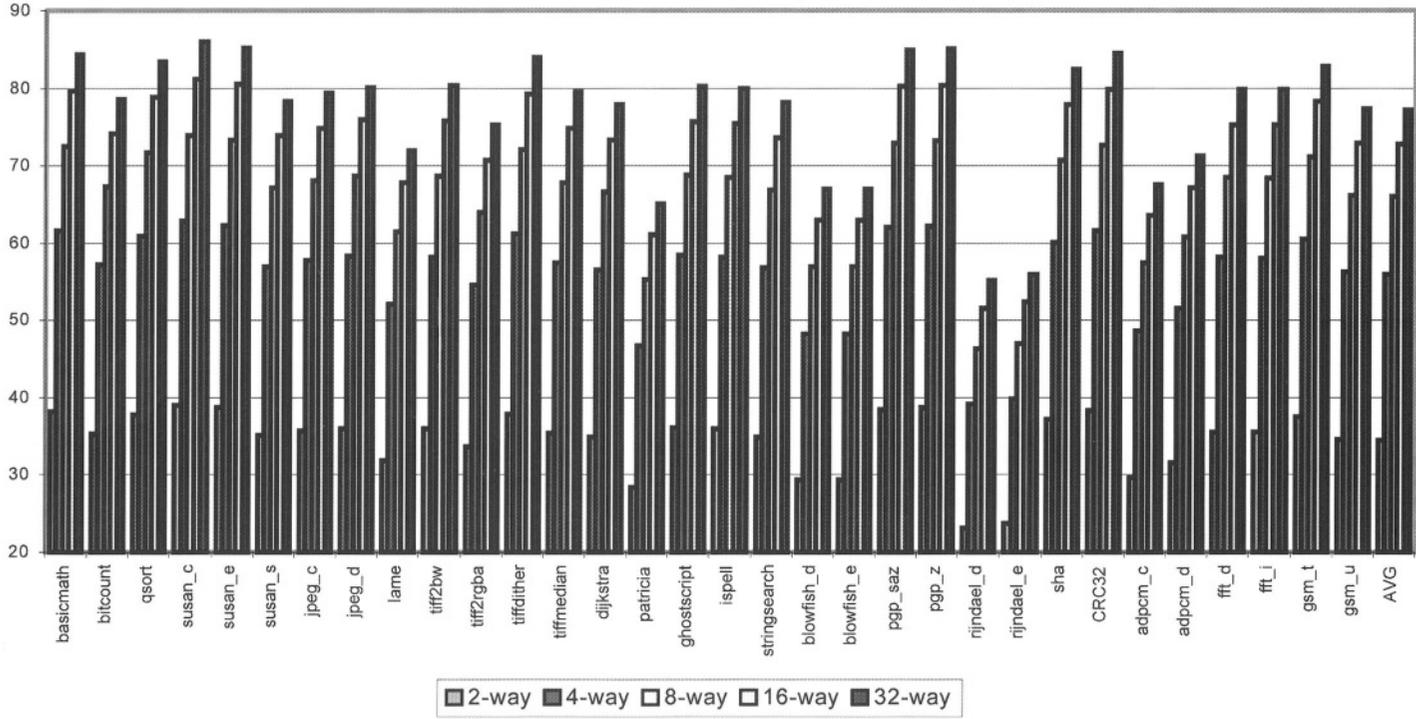


Figure 37-8. Data cache energy consumption savings for a 32-entry WDU.

data caches of different associativities. The size of the cache is the same in all cases 32 KB. For a 4-way data cache the energy savings grow significantly to 56% compared to 34% for a 2-way cache, on average. The savings don't increase as much for 8, 16 and 32-way caches.

5.1. Comparison with way prediction

Figure 37-9 shows the average percentage load/store instructions for which the way can be determined by a WDU of different sizes or can be predicted by a Most Recently Used Way Predictor (MRU) (as presented in [4]). For the MRU way predictor the percentage instructions for which the way is predicted correctly decreases when the cache associativity increases. For 2 and 4-way set associative caches the way predictor has a greater coverage than any WDU. For higher associativities WDU has better coverage, for a 32-way cache the 16-entry WDU already has better performance than the MRU way predictor.

Figure 37-10 shows the data cache energy consumption savings for dif-

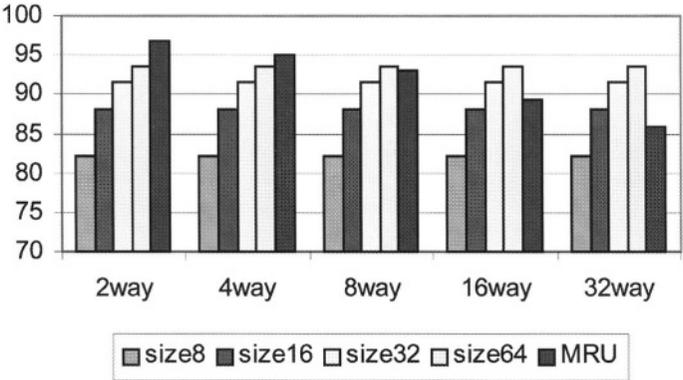


Figure 37-9. Average percentage load/store instructions "covered" by way determination and MRU way prediction.

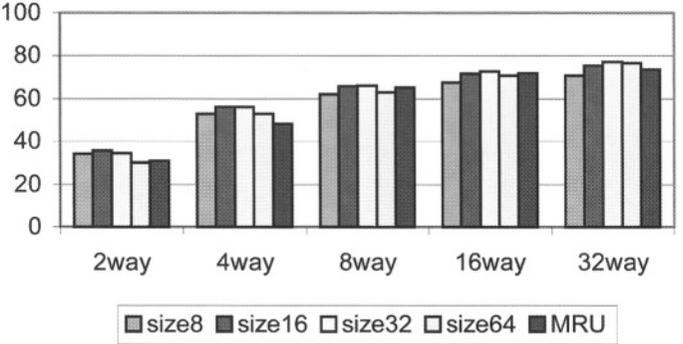


Figure 37-10. Average percentage data cache energy savings.

ferent sizes WDUs and the MRU way predictor. For a 2-way the data cache power savings are smaller for the MRU predictor than for the WDU. Although the MRU predictor has a higher prediction rate, the energy consumption overhead of the predictor MRU reduces the total data cache power savings. For higher associativities the predictor overhead decreases, but so does the prediction rate so, except for small WDU sizes, the WDU has better energy savings.

## 6. CONCLUSIONS

This paper addresses the problem of the increased energy consumption of associative data caches in modern embedded processors. A design for a Way Determination Unit (WDU) that reduces the D-cache energy consumption by allowing the cache controller to only access one cache way for a load/store operation was presented. Reducing the number of way accesses greatly reduces the energy consumption of the data cache.

Unlike previous work, our design is not a predictor. It does not incur mis-prediction penalties and it does not require changes in the ISA or in the compiler. Not having mis-predictions is an important feature for an embedded system designer, as the WDU does not introduce any new non-deterministic behavior in program execution. The energy consumption reduction is achieved with no performance penalty and it grows with the increase in the associativity of the cache.

The WDU components, a small fully associative cache and a modulo counter, are well understood, simple devices that can be easily synthesized. It was shown that very a small (8–16 entries) WDU adds very little to the design gate count, but can still provide significant energy consumption savings.

The WDU evaluation was done on a 32-bit processor with virtually indexed L1 cache. For a machine with a physically indexed cache the WDU overhead would be even smaller resulting in higher energy consumption savings.

## REFERENCES

1. D. Brooks, V. Tiwari, and M. Martonosi. “Wattch: A Framework for Architectural-Level Power Analysis and Optimizations.” In *ISCA*, pp. 83–94, 2000.
2. D. Burger and T. M. Austin. “The SimpleScalar Tool Set, Version 2.0.” *Technical Report TR-97-1342*, University of Wisconsin-Madison, 1997.
3. M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. “Mibench: A Free, Commercially Representative Embedded Benchmark Suite.” In *IEEE 4th Annual Workshop on Workload Characterization*, pp. 83–94, 2001.
4. K. Inoue, T. Ishihara, and K. Murakami. “Way-Predicting Set-Associative Cache for High Performance and Low Energy Consumption.” In *ACM/IEEE International Symposium on Low Power Electronics and Design*, pp. 273–275, 1999.

5. Intel. *Intel XScale Microarchitecture*, 2001.
6. R. E. Kessler. "The Alpha 21264 Microprocessor." *IEEE Micro*, Vol. 19, No. 2, pp. 24–36, March/April 1999.
7. A. Klaiber. "The Technology Behind Crusoe Processors." *Technical Report*, Transmeta Corporation, January 2000.
8. Motorola. *MPC7450 RISC Microprocessor Family User's Manual*, 2001.
9. D. Nicolaescu, A. Veidenbaum, and A. Nicolau. "Reducing Power Consumption for High-Associativity Data Caches in Embedded Processors." In *Proceedings DATE03*, 2003.
10. P. Shivakumar and N. P. Jouppi. *Cacti 3.0: An Integrated Cache Timing, Power, and Area Model*.
11. W. Tang, A. Veidenbaum, A. Nicolau, and R. Gupta. *Simultaneous Way-Footprint Prediction and Branch Prediction for Energy Savings in Set-Associative Instruction Caches*.
12. E. Witchel, S. Larsen, C. S. Ananian, and K. Asanovic. "Direct Addressed Caches for Reduced Power Consumption." In *Proceedings of the 34th Annual International Symposium on Microarchitectures*, 2001

*This page intentionally left blank*

# INDEX

abstract communication  
access latency of cache  
access pattern  
address computation  
AES  
affine condition  
affinity  
AHB multi-layer  
analyzable reference  
APIs  
APOS  
application hotspots  
Application Specific Instruction-Set Processor (ASIP)  
architecture mapping  
architectural model  
ARM core  
array-based applications  
array interleaving  
assembly  
assertion checker  
assertion language  
automatic and application-specific design of HAL  
automation  
automotive  
automotive software

bandwidth  
bank switching  
banked memory  
banking of cache  
behavioral model  
bit patterns in data  
bit-flips  
Bitline scaling  
Buffer Overflows  
Bus Models

cache  
cache bypassing  
cache effects  
cache optimization  
cache performance

call-by-need  
certifiable software  
chip-scale parallelism  
clock gating  
clustered  
co-design  
co-scheduling  
code generation  
communication channels  
communication refinement  
communication scheduling  
communications  
compilers  
computation reuse  
concurrency  
configurable processor  
configurability  
context switching  
control flow  
coprocessor selection  
coroutines  
correct by construction code synthesis  
COTS  
cryptography

data assignment  
data dependency  
data reuse  
data transformation  
data upsets  
data-dominated  
data-flow analysis  
debugging  
design  
design flow  
design space exploration  
detection efficiency  
device driver  
device driver synthesis  
device programming interface  
distributed real-time systems  
dynamic behavior  
dynamic C  
dynamic parallelization  
dynamic power management

- dynamic scheduling
- dynamic voltage scaling
- eCOS
- effect-less
- electronic control unit certification
- embedded benchmarks
- embedded OS
- embedded processor
- embedded processors
- embedded RTOS
- embedded software
- embedded software streaming
- embedded systems
- energy consumption
- energy efficiency
- EPIC
- error containment
- error rate
- Ethernet
- event driven finite state machine
- event handling
- fault containment
- fault tolerance
- filesystem
- flow aggregation
- formal timing analysis
- framework
- functional unit assignment
- genetic algorithm
- HAL standard
- hardware abstraction layer
- hardware dependent software
- hardware detection
- hardware software interface synthesis
- hardware/software co-design
- hardware/software optimization
- hardware/software partitioning
- hash table
- high associativity
- high performance
- HW/SW partitioning
- If-statement
- ILP extraction
- intra-processor communication
- interactive ray tracing
- Intermediate representation
- interrupt handling
- intervals
- instruction reuse
- Instruction selection
- IP based embedded system design
- iSSL
- Kahn process network
- kernel
- libraries
- link to implementation
- load balance
- locality
- locality improvement
- Logic of Constraints (LOC)
- loop iteration
- loop nest splitting
- loop optimization
- loop transformation
- low energy
- low power
- mapping
- marking equation
- memory layouts
- memory recycling
- middleware
- microcontroller
- model refinement
- modeling
- modulo scheduling
- MP SoC
- multi-media applications
- multi-processor,multi-tasked applications
- Multi-rate
- Multimedia
- multiprocessor embedded kernel
- multiprocessor SoC
- multitasking
- multithread
- nested-loop hierarchy
- network processors
- network-on-chip
- networking
- on-chip communication
- on-chip communicationanalyzable reference
- on-chip generic interconnect
- on-chip multiprocessing
- on-demand computation
- on-line checkpointing
- on/off instructions,
- operating systems
- optimization
- OS-driven Software

- packet flows
- parameter passing
- pareto points
- partitioning
- performance analysis
- performance constraint
- performance estimation
- peripheral device modeling
- petri nets
- physical layer
- picture-in-picture (PIP)
- pipeline
- pipeline latches
- pipelined cache
- pipeline stall
- platform independence
- platform-based HW/SW co-design
- polytope
- posix threads
- porting
- power estimation
- predication
- process
- process scheduling
- product line
- programming languages
- protected OS
  
- quantitative constraint
- quasi-static scheduling
  
- rabbit
- reachability graph
- reactive systems
- real time operating systems (RTOS)
- real-time
- real-time systems
- region detection
- reliability model
- Rijndael
- RMC2000
- resource conflict
- RTOS
- RTOS characterization
- RTOS model/modeling and abstraction
- RTOS modeling
- runtime optimization
  
- safety critical
- safety-critical system
- Satisfiability
- schedulability analysis
- scheduling analysis
- s dram
  
- selfishness
- Sequence-Loss
- SET
- SEU
- SIMD Reconfigurable architecture
- SimpleScalar
- simulation
- simulation model of HAL
- simulation monitor
- simulation speedup
- simulation verification
- simultaneous multithreading
- SoC
- sockets
- software engineering
- software generation
- software integration
- software performance validation
- software synthesis
- software-detection
- source code transformation
- speculation
- speech recognition
- specification methodology
- static/dynamic energy
- stochastic communication
- synchronization Errors
- system-level design
- system-level design language (SLDL)
- (system-level) design methodology
- system-level modeling
- System-on-Chip
- System on Chip Bus
- SystemC
- SWCD
- system-on-a-chip
- software integration
- Software Architectural Transformations
- Software Streaming
- SoC
- software reuse
- State space compression
- Software synthesis
  
- task management
- task migration
- task synchronization & communication
- TCP
- thread
- time-sharing
- time-triggered architecture
- time/delay modeling
- timing back-annotation
- transaction Level Modeling TLM

trace analysis  
tracking device

ultra-dependable computer applications  
UML

virtual chip interconnect  
Voice encoder/decoder  
VLIW

way determination

X-by-Wire