Lecture Notes in Computer Science 4697

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison
Lancaster University, UK
Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA
Josef Kittler
University of Surrey, Guildford, UK
Jon M. Kleinberg
Cornell University, Ithaca, NY, USA
Friedemann Mattern
ETH Zurich, Switzerland
John C. Mitchell
Stanford University, CA, USA
Moni Naor
Weizmann Institute of Science, Rehovot, Israel
Oscar Nierstrasz
University of Bern, Switzerland
C. Pandu Rangan
Indian Institute of Technology, Madras, India
Bernhard Steffen
University of Dortmund, Germany
Madhu Sudan
Massachusetts Institute of Technology, MA, USA
Demetri Terzopoulos
University of California, Los Angeles, CA, USA
Doug Tygar
University of California, Berkeley, CA, USA
Moshe Y. Vardi
Rice University, Houston, TX, USA
Gerhard Weikum
Max-Planck Institute of Computer Science, Saarbruecken, Germany

Lynn Choi   Yunheung Paek
Sangyeun Cho (Eds.)

Advances in Computer Systems Architecture

12th Asia-Pacific Conference, ACSAC 2007
Seoul, Korea, August 23-25, 2007
Proceedings

Springer

Volume Editors

Lynn Choi
Korea University
School of Electrical Engineering
Anam-Dong, Sungbuk-Ku, Seoul, Korea
E-mail: lchoi@korea.ac.kr

Yunheung Paek
Seoul National University
School of Electrical Engineering
Seoul, Korea
E-mail: ypaek@snu.ac.kr

Sangyeun Cho
University of Pittsburgh
Department of Computer Science
Pittsburgh, PA 15260, USA
E-mail: cho@cs.pitt.edu

Library of Congress Control Number: 2007932678

CR Subject Classification (1998): B.2, B.4, B.5, C.2, C.1, D.4

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN 0302-9743
ISBN-10 3-540-74308-1 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-74308-8 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

Springer is a part of Springer Science+Business Media

springer.com

© Springer-Verlag Berlin Heidelberg 2007
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12109494 06/3180 5 4 3 2 1 0

Preface

On behalf of the program and organizing committee members of this conference, we are pleased to present you with the proceedings of the 12th Asia-Pacific Computer Systems Architecture Conference (ACSAC 2007), which was hosted in Seoul, Korea on August 23-25, 2007. This conference has traditionally been a forum for leading researchers in the Asian, American and Oceanian regions to share recent progress and the latest results in both architectural and system issues. In the past few years the conference has become more international in the sense that the geographic origin of participants has become broader to include researchers from all around the world, including Europe and the Middle East.

This year, we received 92 paper submissions. Each submission was reviewed by at least three primary reviewers along with up to three secondary reviewers. The total number of completed reviews reached 333, giving each submission 3.6 reviews on average. All the reviews were carefully examined during the paper selection process, and finally 26 papers were accepted, resulting in an acceptance rate of about 28%. The selected papers encompass a wide range of topics, with much emphasis on hardware and software techniques for state-of-the-art multicore and multithreaded architectures. In addition to the regular papers, the technical program of the conference included eight invited papers from world-class renowned researchers and featured two keynotes by Pen-Chung Yew (University of Minnesota) and Kunio Uchiyama (Hitachi), addressing *a compiler framework for speculative multithreading* and *power-efficient heterogeneous multicore chip development,* respectively. We sincerely hope that the proceedings will serve as a valuable reference for researchers and developers alike.

Putting together ACSAC 2007 was a team effort. First of all, we would like to express our special gratitude to the authors and speakers for providing the contents of the program. We would also like to thank the program committee members and external reviewers for diligently reviewing the papers and providing suggestions for their improvements. We believe that you will find the outcome of their efforts in this book. In addition, we extend our thanks to the organizing committee members and student volunteers, who contributed enormously to various aspects of conference administration. Finally, we would like to express special thanks to Chris Jesshope and Jinling Xue for sharing their experience and offering fruitful feedback in the early stages of preparing the conference.

June 2007

Lynn Choi
Yunheung Paek
Sangyeun Cho

Conference Organization

General Co-chairs

Lynn Choi	Korea University, Korea
Sung Bae Park	Samsung Electronics, Korea

Program Co-chairs

Yunheung Paek	Seoul National University, Korea
John Morris	University of Auckland, New Zealand
Sangyeun Cho	University of Pittsburgh, USA

Publicity Chair

Ki-Seok Chung	Hanyang University, Korea

Publication Chair

Hwangnam Kim	Korea University, Korea

Local Arrangement Chair

Sung Woo Chung	Korea University, Korea

Finance Chair

Yunmook Nah	Dankook University, Korea

Registration Chair

Youngho Choi	Konkuk University, Korea

Steering Committee

Jesse Z. Fang	Intel, USA
James R. Goodman	University of Auckland, New Zealand
Gernot Heiser	National ICT, Australia

Kei Hiraki	Tokyo University, Japan
Chris Jesshope	University of Amsterdam, Netherlands
Feipei Lai	National Taiwan University, Taiwan
John Morris	University of Auckland, New Zealand
Amos Omondi	Yonsei University, Korea
Ronald Pose	Monash University, Australia
Stanislav Sedukhin	University of Aizu, Japan
Mateo Valero	Universitat Politecnica de Catalunya, Spain
Jingling Xue	University of New South Wales, Australia
Pen-Chung Yew	University of Minnesota, USA

Program Committee

Jin Young Choi	Korea University, Korea
Bruce Christianson	University of Hertfordshire, UK
Sung Woo Chung	Korea University, Korea
Oliver Diessel	University of New South Wales, Australia
Colin Egan	University of Hertfordshire, UK
Skevos Evripidou	University of Cyprus, Cyprus
Wong Weng Fai	National University of Singapore, Singapore
Michael Freeman	University of York, UK
Guang G. Gao	University of Delaware, USA
Jean-Luc Gaudiot	University of California at Irvine, USA
Alex Gontmakher	Technion, Israel
Gernot Heiser	National ICT, Australia
Wei-Chung Hsu	University of Minnesota, USA
Suntae Hwang	Kookmin University, Korea
Chris Jesshope	University of Amsterdam, Netherlands
Jeremy Jones	Trinity College, Ireland
Norman P. Jouppi	Hewlett Packard, USA
Cheol Hong Kim	Chonnam University, Korea
Doohyun Kim	Kunkook University, Korea
Feipei Lai	National Taiwan University, Taiwan
Hock Beng Lim	Nanyang Technological University, Singapore
Philip Machanick	University of Queensland, Australia
Worawan Marurngsith	Thammasat University, Thailand
Henk Muller	University of Bristol, UK
Sukumar Nandi	Indian Institute of Technology Guwahati, India
Tin-Fook Ngai	Intel China Research Center, China
Amos Omondi	Yonsei University, Korea
L M Patnaik	Indian Institute of Science Bangalore, India
Andy Pimentel	University of Amsterdam, Netherlands
Ronald Pose	Monash University, Australia
Stanislav G. Sedukhin	University of Aizu, Japan
Won Shim	Seoul National University of Technology, Korea
Mark Smotherman	Clemson University, USA

K. Sridharan	Indian Institute of Technology Madras, India
Rajeev Thakur	Argonne National Laboratory, USA
Mateo Valero	Universitat Politecnica de Catalunya, Spain
Lucian N. Vintan	University of Sibiu, Romania
Chengyong Wu	ICT, Chinese Academy of Sciences, China
Zhi-Wei Xu	ICT, Chinese Academy of Sciences, China
Jingling Xue	University of New South Wales, Australia
Pen-Chung Yew	University of Minnesota, USA

External Reviewers

Nidhi Aggarwal	Kai Hwang	Naveen Muralimanohar
Nadeem Ahmed	Lei Jin	Sudha Natarajan
Christopher Ang	Jonghee Kang	Venkatesan Packirisamy
Elizabeth M. Belding-Royer	Kamil Kedzierski	Chanik Park
Darius Buntinas	Daeho Kim	Jagdish Patra
Francisco Cazorla	Jinpyo Kim	Vladimir Pervouchine
José M. Cela	John Kim	Vinod Prasad
Yang Chen	Chung-Ta King	Ken Robinson
Doosan Cho	Tei-Wei Kuo	Esther Salamí
Peter Chubb	Ihor Kuz	Oliverio J. Santana
Ian Clough	Koen Langendoen	Michael Schelansker
Toni Cortés	Robert Latham	Bill Scherer
Kyriacou Costas	Sanghwan Lee	Bertil Schmidt
Adrián Cristal	Heung-No Lee	Ahmed Sherif
Abhinav Das	Hyunjin Lee	Todor P. Stefanov
Amitabha Das	Graham Leedham	Mark Thompson
Michel Dubois	Binghao Li	Jordi Torres
Bin Fan	Huiyun Li	Nian-Feng Tzeng
Jinyun Fang	Kuan-Ching Li	Lei Wang
Yu-Chiann Foo	Wei Li	Yulu Yang
John Glossner	Adam Postula	Jia Yu
Sandeep K. Gupta	Chen Liu	Patryk Zadarnowski
Rubén Conzález	Shaoshan Liu	Ahmed Zekri
Rogeli Grima	Jie Ma	Ge Zhang
Jizhong Han	Luke Macpherson	Jony Zhang
Paul Havinga	Pramod K. Meher	Longbing Zhang
Michael Hicks	Neill Miller	Youtao Zhang
Houman Homayoun	Miquel Moreto	

Student Volunteers

Yong-Soo Bae	Hyun-Joon Lee	Keunhee Yeo
Jae Kyun Jung	Kiyeon Lee	Jonghee Youn
Daeho Kim	Sang-Hoon Lee	

Table of Contents

A Compiler Framework for Supporting Speculative Multicore Processors

Pen-Chung Yew

University of Minnesota at Twin Cities

As multi-core technology is currently being deployed in computer industry primarily to limit power consumption and improve throughput, continued performance improvement of a *single application* on such systems remains an important and challenging task. Because of the shortened on-chip communication latency between cores, using thread-level parallelism (TLP) to improve the number of instructions executed per clock cycle, *i.e.*, to improve ILP performance, has shown to be effective for many *general-purpose* applications. However, because of the program characteristics of these applications, effective speculative schemes at both thread- and instruction-level are crucial.

Processors that support speculative multithreading have been proposed for sometime now. However, efforts have only begun recently to develop compilation techniques for this type of processors. Some of these techniques would require efficient architectural support. The jury is still out on how much performance improvement could be achieved for general-purpose applications on this kind of architectures.

In this talk, we focus on a compiler framework that supports thread-level parallelism with the help of control and data speculation for general-purpose applications. This compiler framework has been implemented on the Open64 compiler that includes support for efficient data dependence and alias profiling, loop selection schemes, as well as speculative compiler optimizations and effective recovery code generation schemes to exploit thread-level parallelism in loops and the remaining code regions.

© Springer-Verlag Berlin Heidelberg 2007

Power-Efficient Heterogeneous Multicore Technology for Digital Convergence

Kunio Uchiyama

Hitachi, Ltd

In recent mobile phones, car navigation systems, digital TVs, and other consumer electronic devices, there is a trend toward digital convergence in which a single device has the ability to process various kinds of applications. At the same time, considering the processing of media content, these devices must be capable of encoding and decoding video images and audio data based on MPEG2, MPEG4, H.264, VC-1, MP3, AAC, WMA, RealAudio, and other formats. Moreover, the latest DVD recorders have the ability to automatically generate digests of video images by using audio and image recognition technology. These kinds of digital convergence devices must be able to flexibly process various kinds of data—media, recognition, data, communications, and so on—and the SoC (System-on-Chip) that is embedded in the devices must deliver superior performance while consuming very small power.

To meet these needs, a power-efficient heterogeneous multi-core technology for the SoC used in consumer electronic devices has been developed. Primary objectives in developing this technology are to: (1) establish a robust heterogeneous multicore architecture that integrates a number of different types of power-efficient processor cores; (2) incorporate dynamic reconfigurable processors to leverage parallelism at the operation level; and (3) create a new software development environment for efficiently developing programs tailored for the heterogeneous multicore architecture. This combination of attributes will give us the superior performance/power ratio and flexibility, while satisfying the enormous demand for digital convergence devices.

The power-thrifty processors used in the heterogeneous multicore architecture essentially include a local memory and an intelligent data transfer unit. Each local memory functions as a distributed shared memory for the entire chip. Processing is speeded up by enabling operations within processors in parallel with data transfers between processors. Dynamic reconfigurable processors called Flexible Engines (FEs) have been implemented as a special type of processor core. The FE executes various arithmetic algorithms fast while dynamically changing the functions and interconnections among 32 arithmetic elements.

A prototype heterogeneous multicore chip has been developed using 90nm technology based on the architecture described above. Four low-power CPU cores are integrated along with two FEs on the 96mm^2 chip. The CPU core operates at 600 MHz and has a performance of 1.08 GIPS or 4.2 GFLOPS, while the FE operates at 300 MHz, and can perform up to 19.2 GOPS. The chip as a whole delivers a performance of 4.32 GIPS, 16.8 GOPS, and 38.4 GOPS with a power dissipation of less than several watts.

When a program is executed on the heterogeneous multicore chip, the program is divided up into sub-programs, which are processed by the processor cores on the chip

© Springer-Verlag Berlin Heidelberg 2007

that are best suited to the task based on the attributes of each sub-program part. Multimedia programs such as encoding audio data have been executed on various combinations of CPUs and FEs, and the performance and the power consumption of the various configurations have been evaluated.

A new software development environment has been created for the efficient development of programs specifically tailored for the heterogeneous multicore architecture. Using the new platform, programs are broken up into sub-program parts. The object code for the portions executed by CPU cores is generated using a usual compiler. An FE compiler has been developed for the sub-program parts executed by FEs, and the compiler generates configuration data and sequence control codes tailored for the FEs. A graphical interface editor for optimizing FE libraries has also been developed. It not only enables programmers to write FE programs directly but also enables the programs to be verified by simulation.

A part of the introduced research has been supported by NEDO “Advanced heterogeneous multiprocessor,” “Multicore processors for real-time consumer electronics,” and “Heterogeneous multi-core technology for information appliances.”

StarDBT: An Efficient Multi-platform Dynamic Binary Translation System

Cheng Wang, Shiliang Hu, Ho-seop Kim, Sreekumar R. Nair, Mauricio Breternitz Jr., Zhiwei Ying, and Youfeng Wu

Programming Systems Lab, Intel Corporation
2200 Mission College Blvd
Santa Clara, CA 95052, USA
{cheng.c.wang,shiliang.hu,ho-seop.kim,sreekumar.r.nair, mauricio.breternitz.jr,victor.ying,youfeng.wu}@intel.com

Abstract. This paper describes the design and implementation of a research dynamic binary translation system, StarDBT, which runs many real-world applications. StarDBT is a multi-platform translation system that is capable of translating application level binaries on either Windows or Linux OSes. A system-level variant of StarDBT can also run on a bare machine by translating the whole system code. We evaluate performance of a user-mode system using both SPEC2000 and some challenging Windows applications. StarDBT runs the SPEC2000 benchmark competitively to other state-of-the-art binary translators. For Windows applications that are typically multi-threaded GUI-based interactive applications with large code footprint, the StarDBT system provides acceptable performance in many cases. However, there are important scenarios in which dynamic translation still incurs significant runtime overhead, raising issues for further research. The major overheads are caused by the translation overhead of large volume of infrequently-executed code and by the emulation overhead for indirect branches.

Keywords: Dynamic binary translation, performance evaluation.

1 Introduction

Dynamic binary translation (DBT) has many attractive applications in computer system designs. For example, it can be used to support legacy binary code [4]; support ISA virtualization [1]; enable innovative co-designed microarchitectures [7], [13], and many others [3], [10], [14], [15], [19], [20]. However, DBT technology also comes with its costs: translation overhead, emulation overhead and potentially other runtime overheads. It is an interesting research topic to obtain insights for designing systems featuring binary translation.

To evaluate DBT design and application, we developed a multi-platform DBT system, named StarDBT. StarDBT translates from IA (Intel Architecture, a.k.a 'x86') to IA, including from IA32 to Intel64. As a multi-platform system, StarDBT can run as a user-mode module that resides in user process space. Currently, we have OS-specific support for user-mode DBT on both Linux and Windows x64 platforms. Furthermore, StarDBT can also serve as a system level DBT that runs directly on

© Springer-Verlag Berlin Heidelberg 2007

hardware and boots commercial OSes. Abstraction and modularization is the key for StarDBT to support multiple platforms. Specifically, the generic part of the DBT system is separated, modularized and shared. The platform-dependent modules of the DBT are separated from each other and interface with the DBT generic part through an internal API. The DBT system has been stabilized enough to run many real-world applications. Hence, we currently focus on performance evaluation, tuning and obtaining insight on DBT runtime behavior.

For real world deployment, it is important that a new technology handles all kinds of cases and performs well on representative applications. Pathological cases should be identified and handled gracefully. Therefore, we need to evaluate performance on a wide variety of workloads. In this paper, we report our DBT performance for client-side workloads, namely, SPEC2000 and some popular Windows applications.

For the SPEC2000 suite, our StarDBT performs competitively to state-of-the-art DBT systems [4], [5], [17]. Arguably its performance is comparable to native runs (e.g. about 12% performance difference for SPEC2000 on Pentium 4 systems). For Windows applications that are less-benchmarked on most DBT systems, we observed acceptable performance in many cases (e.g. ranging from 10% to 40% slowdowns) and also found interesting performance issues in some others. Performance for Windows applications is critical to our DBT design. On one hand, Windows applications are more representative workloads for most computer users. On the other hand, Windows applications are more challenging for DBT systems -- they are typically multi-threaded, interactive GUI workloads that show different runtime behaviors than the frequently-benchmarked SPEC2000. Our StarDBT system has encountered severe translation overhead for some Windows applications due to their larger code footprint and less code reuse. Meanwhile, more indirect branches (e.g. more function and DLL calls/returns) in Windows workloads also adversely stresses the inefficient indirect branch emulation schemes of software-only DBT systems. In this paper, we strive for better understanding on DBT runtime behavior for Windows workloads and try to gain insights for DBT system design.

This paper describes the StarDBT system, briefing highlights on its design and implementation. Preliminary performance evaluation is also presented. The rest of the paper is organized as follows. Section 2 discusses related work and summarizes the state-of-the-art DBT technology. Section 3 presents the design and implementation of the StarDBT system. Section 4 reports performance evaluation of the system. Section 5 concludes the paper.

2 Related Work

There are a few product systems that adopted the DBT technology [1], [4], [8], [13]. There are also many research papers studying DBT technology [2], [5], [9], [12], [16], [17]. However, DBT runtime overhead has long been a concern for the industry. Performance results are mostly published using hotspot (frequently executed code) dominant workloads such as the SPEC CPU 2000 [11] (referred to as SPEC2000 in this paper). In general, current DBT systems perform from comparably to competitively when compared with native runs for hotspot code. However, non-hotspot (cold, infrequently executed code) performance is less studied.

DynamoRIO [5] is an IA32 to IA32 DBT system that supports flexible binary inspection and instrumentation. Performance numbers are published for the SPEC2000 and four Windows applications. Pin [14] is an instrumentation tool primarily on Linux, with limited Windows support so far. HDTrans [17] is a simple fast Linux-based binary translator. Its simplicity speeds up its cold code translation performance and it shows competitive performance among DBT systems that do not optimize hotspots. Performance is not reported for interactive GUI workloads. FX!32 [6] employs runtime interpretation coupled with off-line static binary translation and optimization. It runs Windows applications, but it does not fit into the DBT category. IA32-EL [4] runs Windows applications on Intel IPF platforms, and published SYSMARK2000 performance results. Although it observed less efficient execution for Windows applications, little is discussed about the underlying dynamics. VMware [1] virtualizes the IA32 instruction set using dynamic binary translation to scan and translate certain resource-sensitive instructions. Therefore, VMware VMMs only translate 3% of the code (OS kernel code) and cause about 4% slowdown due to translation overhead. Transmeta [13] used a code morphing software (CMS) to implement IA32 processors and run IA32 binaries on mobile platforms. However, few performance results are published.

3 StarDBT Design and Implementation

The full spectrum of potential DBT applications motivated our system design to support multiple platforms. The binary translator evaluated in this paper targets translation from IA32 into Intel64 at user level. Namely, it transparently translates 32-bit IA32 application binaries into 64-bit Intel64 code at runtime, enabling full advantages of 64-bit computing for legacy 32-bit applications.

3.1 Multi-platform DBT Architecture

StarDBT runs on both Windows and Linux platforms. StarDBT for Windows platform runs popular Windows desktop applications such as Microsoft Office and Internet Explorer. The Linux version runs command-line Linux applications and also runs on top of a cycle-accurate simulator that simulates at Linux ABI level. Additionally, StarDBT also runs as a system-level virtual machine monitor that boots and runs commercial OSes. StarDBT is designed to boot various commercial OS kernels. At this point, it can boot Linux kernels.

To support such a multi-platform system, we separated the DBT-generic (platform-independent) part from DBT-platform (platform-dependent). The interface between these modules is an internal API named DBT-platform API. DBT-generic requests services such as resource allocation through this API. A DBT-platform module notifies the DBT-generic of certain events such as exceptions or callbacks also through this API.

There are two major components of the DBT system: the binary translator and the runtime system that manages and controls the execution of the entire system. The binary translator components are mostly DBT-generic. However, the runtime system has to be divided into a generic part and a platform (-specific) part. The generic part of the runtime dispatches execution, and manages the code cache and other important DBT resources in a platform-abstract way. The platform part of the runtime system provides platform-specific services such as initialization, finalization, platform interaction etc. Each platform needs its own specialized implementation of the platform part of the runtime. For example, the Linux platform runtime inserts a kernel module to load the DBT system, which is packed as a Linux user-mode dynamic library. The Windows x64 version DBT-platform runtime rewrites some part of the Microsoft wow64 runtime system to integrate our binary translators.

3.2 Runtime Translation Design

Dynamic binary translation incurs significant runtime overhead; this is especially true for the complex x86 instruction set. Therefore, like most other sophisticated DBT systems, we apply an adaptive translation strategy. It uses a simple fast translator for cold code translation and once a workload hotspot is detected, it applies optimizations.

The simple and fast translation tries to generate target code with minimal runtime overhead. For most IA32 computation instructions (e.g. ALU ops and data moves), the DBT simply decodes and recognizes them. Then the translator copies these instructions to generate Intel64 code. Some IA32 instructions are no longer available in Intel64. These cases are further detailed in Subsection 3.3. Control transfer instructions such as branches, function calls and returns need to be rewritten. This involves some translation lookup and dispatch code (subsection 3.4).

For program hotspots, we form straight-line traces (called regions) to optimize. Because this is conducted at run-time, we only implement a few effective optimizations such as code layout, registerization (for more registers in Intel64) and partial redundancy elimination.

Translators place generated code in code caches for later reuse. The runtime dispatcher maintains a translation lookup table. The execution of the original IA32 code is achieved via emulating in code caches and dispatching in the runtime system. Additionally, we use a 64K-entry translation lookup table as in [5] and we allocate 16MB of memory for code cache. Once the lookup table or the cache is full, we simply flush the code cache.

Generally, it is difficult to maintain true compatibility for user-mode virtual machines due to their memory footprint inside the application process. However, on 64-bit systems, the larger 64-bit memory space offers extra memory space beyond the original 32-bit application space. Thus, our runtime system maintains true compatibility by allocating DBT memory above (outside) the 32-bit user space.

3.3 IA32 to Intel64 Translation

As aforementioned, some IA32 instructions are no longer supported in Intel64. For example, push and pop instructions only support 64-bit operands in 64-bit mode, making it non-straightforward to emulate the 32-bit stack efficiently. In our translator, we generate 64-bit code to emulate 32-bit stack instructions. For example, push32 eax is translated into the following code sequence:

```
lea   esp, [rsp - 4]
mov   [rsp], eax
```

It is important that the generated code for stack manipulation never uses space beyond its stack pointer, `rsp`. Some OS kernels may use the space beyond the stack to throw exception and context records for exception handling.

Some IA32 instructions are aliased. However, in Intel64, they have only one opcode available. For example, all eight instructions (as a group) with opcode 82H are the same instructions as those with opcode 80H in IA32. There is a need to remap their opcode to the only opcode available in Intel64: 80H.

There are also IA32 instructions that are missing in Intel64. For example, some BCD arithmetic and bit manipulation operations are no longer supported in Intel64. These instructions can be emulated with a sequence of Intel64 instructions. A tricky part is the segment manipulation instructions in IA32. The 64-mode does not support segmentation directly. Our user-mode StarDBT ignores the segmentation manipulation instructions (except for GS and FS segments which are still supported in Intel64). However, inside the kernel (as seen by our system-mode StarDBT), segmentation are processed in its true architectural sense.

Table 1 summarizes some of our translation from IA32 to Intel64.

Clearly, translating IA32 legacy instructions into Intel64 causes code expansion. The Intel64 REX prefix causes additional code expansion. However, in Intel64, more registers are available and we can exploit these extra registers to improve hotspot code performance.

Table 1. Translation of Some IA32 Instructions

Original IA-32 instruction	Translated Intel64 Instruction
PUSH ESP	REX. MOV R8/32, ESP LEA ESP, [RSP – 4] REX. MOV [RSP], R8/32
PUSH imm/32 (imm/8)	LEA ESP, [RSP – 4] MOV [RSP], imm/32 (imm/32*) // sign extend imm/8 to imm/32
PUSH r/32	LEA ESP, [RSP – 4] MOV [RSP], r/32
PUSH m/32	REX. MOV R8/32, M/32 LEA ESP, [RSP – 4] REX. MOV [RSP], R8/32
POP ESP	MOV ESP, [RSP]
POP r/32	MOV r/32, [RSP] LEA ESP, [RSP + 4]
POP m/32	REX. MOV R8/32, [RSP] REX. MOV m/32, R8/32 LEA ESP, [RSP + 4]
ENTER imm16, imm8	LEA ESP, [RSP – 4] MOV [RSP], EBP LEA ESP, [RSP – imm16] (imm16 > 0) REX. MOV R8/32, [EBP – 4] REX. MOV [RSP + imm16 – 4], R8/32 … REX. MOV R8/32, [EBP – 4 * imm8 + 4] (imm8 > 1) REX. MOV [RSP + imm16 – 4 * imm8 + 4], R8/32 (imm8 > 1) LEA EBP, [RSP + imm16] MOV [RSP + imm16 – 4 * imm8], EBP (imm8 > 0)
LEAVE	MOV ESP, EBP MOV EBP, [RSP] LEA ESP, [RSP + 4]

3.4 Control Transfers in Code Cache

Control transfers between translated code in the code cache can cause significant runtime overhead if handled inefficiently, because of the need to look up translated addresses. StarDBT eliminates direct jumps and (eventually) chains direct conditional branches together, maintaining execution in the code cache. Indirect branches are inlined with dispatch code to lookup the translation mapping table for their translated targets. The translation mapping table for large footprint applications can be quite big. We speed up this dispatch code for indirect branch target lookup with a special cache table. Return instructions are handled similarly to indirect branches.

4 Evaluation and Characterization

4.1 Evaluation Methodology

We evaluate our StarDBT system using both SPEC2000 and widely-used Windows GUI-based interactive applications (mostly from the SYSMARK 2004 SE [18]). We run SPEC2000 as a whole to collect the baseline performance numbers. These data can be compared with prior DBT systems and they are good performance indicators for batch-mode and long-run applications. We selected seven Windows applications to study DBT performance for interactive workloads. Unfortunately, the SYSMARK 2004 SE benchmark is too large (and out of our script control) for performance characterization tools. Therefore, we developed our own automation scripts to run these applications. Whenever possible, we develop scripts that run similar scenarios as the original SYSMARK 2004. Otherwise, we develop scenarios that reflect common usage of the benchmarked software. The workload scenarios will be described shortly.

We use the Intel VTune Performance Analyzer v8.02 to study performance issues. In our experiments, we emphasize the measurement of response time for interactive workloads. In many cases, this can be measured by a timer. However, this is more or less subjective and imprecise. For example, sometimes the GUI response looks ready for user's interaction; however some background processing may be still active. A timer measurement in this situation is not well-defined. In our study, we use the number of duty cycles (collected by VTune) to measure the response time. The duty cycle in VTune is defined as the number of cycles that processors use to execute instructions from the application(s) being monitored.

We use two generations of machines to study platform sensitivity in our study. The first machine is a 3.60GHz (Pentium 4) Xeon based dual-processor system that runs Windows 2003 x64 server. It has 1MB L2 cache per core and 2GB memory. The second machine is a 3.0GHz Intel Xeon 5160 (Woodcrest) based Dell Precision 490 workstation, also running Windows 2003 x64 server. The Dell Woodcrest system has two dual-core Woodcrest processors (4 processor cores) and 4GB of memory.

4.2 Windows Interactive Workloads

The criteria for developing our workloads are: (1) The scenarios must be representative of user interactive applications, using widely-used software; (2) the benchmark scripts should run long enough to be realistic, but short enough to be characterized (using VTune). We use the same BAPCO SYSMARK 2004 scenarios whenever possible. And eventually we developed the following workload scenarios in the test: (Non-Microsoft software in SYSMARK 2004 could not be launched outside the SYSMARK scripts)

- ***Access*****:** the script opens the SYSMARK 2004 sales database, performs two queries and then exports the result to Excel format.
- ***Excel*****:** the script opens the worksheets from the SYSMARK 2004. It performs two sorting operations and calculates values for large columns based on formulas. Finally, it does some auto formatting.
- ***Power Point (PPT)*****:** this script opens the car sales and commercial presentation from SYSMARK 2004. Then it shows slides of the 70-second presentation that contains figures, animations and a short video clip.
- ***Word*****:** this script opens the Tom Sawyer document from the SYSMARK 2004. It performs a set of formatting, editing operations before a final print-preview.
- ***Internet Explorer (IE)*****:** this script activates the IE browser, searches on Google for SYSMARK 2004 and its white paper. It opens the white paper, flips through and closes it. Then it surfs several well-known websites such as CNN and ESPN. It also searches OLE automation documents from MSDN and finally it checks our local weather from weather.com and a stock page from Google Finance.
- ***Media Player (MPlayer)*****:** this script loads and plays the SYSMARK 2004 car commercial video clip (68 seconds).
- ***Visual Studio (VStudio)*****:** this is our own benchmark scenario. It opens Microsoft Platform SDK and builds our entire StarDBT system. In a sense, it is similar to self compilation tests for many compilation systems.

Table 2 shows some basic characteristics of our benchmark scenarios when they run natively on the Dell 490 Woodcrest system without DBT. The first data column shows the instruction count for each benchmark run. The second data column shows the duty cycles for each run, collected by VTune. And the third data column shows the elapsed wall time for the benchmark scenario, including user thinking time. It is clear that most benchmarks run tens of seconds and tens of billions duty cycles.

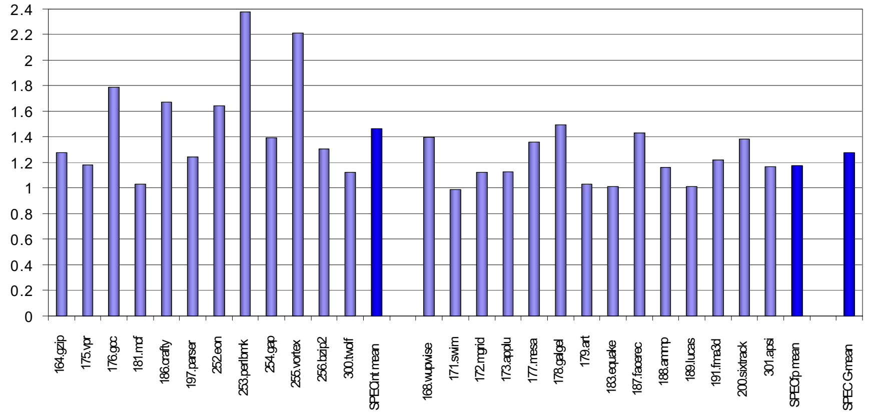
Table 2. Benchmark basic characteristics

BENCH MARK	INSTR COUNT (BILLIONS)	DUTY CYCLE (BILLIONS)	WALL TIME (SECOND)
Access	65.607	52.281	76
Excel	8.178	11.038	69
PPT	123.117	159.169	77
Word	6.765	11.253	69
IE	15.369	25.501	83
MPlayer	43.113	44.463	70
VStudio	48.576	69.114	43

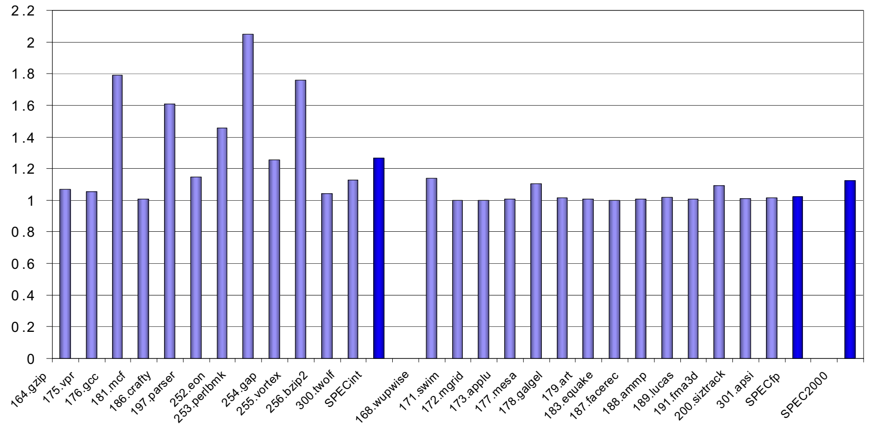
4.3 SPEC2000 Performance

Although we have developed hotspot optimizations in our StarDBT system, they are not particularly helpful in our GUI interactive scenarios and sometime degrade performance. Furthermore, to compare with prior projects, it is important to have a consistent setting. Therefore, in our baseline performance measurement using SPEC2000, we disabled hotspot optimizations except hot trace re-layout.

Given the StarDBT settings above, Figure 1 shows the SPEC2000 performance running on our StarDBT system on the Woodcrest system. The input binaries are compiled with Intel ICC compiler 9.0 at the highest optimization level (O3) with profile feedback. Because hotspot optimizations are not invoked, StarDBT's slowdowns are due to DBT translation and emulation overhead. The average slowdown for



(a) SPEC2000 Performance on Woodcrest platform



(b) SPEC2000 performance on Pentium 4 platform

Fig. 1. SPEC2000 performance

integer benchmarks is about 46% while the FP benchmarks show an average slowdown of 17%, making the aggregate slowdown for the entire SPEC2000 approximately 27%. However, the same performance test on the Pentium 4 machine shows respectively 25%, 2% and 12% slowdowns for the SPEC2000 integer, FP and entire suite. The performance difference between the two generations of processors reflects the different processor efficiency and the memory hierarchy efficiency of the test platforms. The new Woodcrest processors are more efficient and thus are more sensitive to extra instructions caused by DBT dynamic instruction count expansion. This effect can be seen even from the same platform while running applications with different memory behavior. For example, in Figure 1, memory intensive (thus pipeline less efficient) programs such as mcf and FP benchmarks cause less slowdown than CPU intensive programs on both platforms.

DynamoRIO [5] reported slightly better performance (17% slowdown for SPEC2000 integer, with a different experimental setting) than our StarDBT when only simple translation is enabled. HDTrans [17] also translates from IA32 to IA32 and reported slightly better performance than DynamoRIO. Although the quality of the input binaries and the experimental machines may influence the performance differences, all these experiments demonstrate that current DBT systems can work fairly well on CPU intensive workloads where program footprint is small and the translated code is heavily reused.

4.4 Interactive Application Performance

To better understand how DBT affects user experience of Windows applications, we collect responsiveness data when running Windows applications on top of our StarDBT. The responsiveness is measured both by timing elapsed wall clock time and by collecting the duty cycle numbers using VTune. Figure 2 graphs the slowdown factor (when compared with native runs on Woodcrest) of the scenarios described in subsection 4.2.

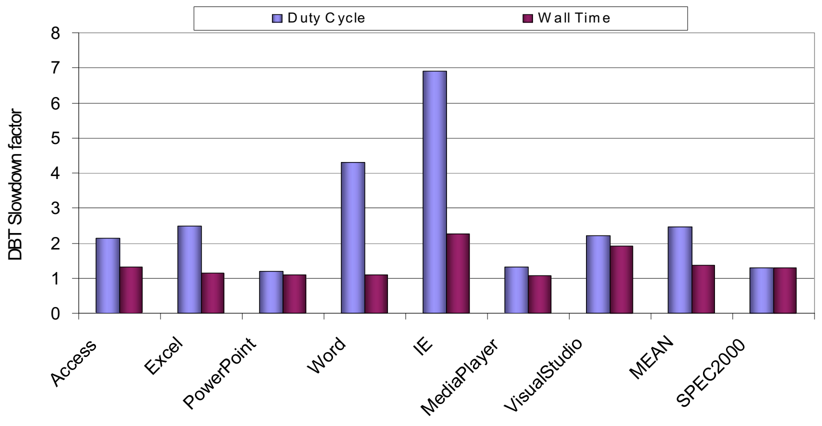


Fig. 2. Window application responsiveness slowdown

It is found that the responsiveness in terms of wall time is not too bad, 35% slowdown on average, slightly worse than the SPEC2000 slowdown. In our actual user experience, there is little noticeable difference between DBT runs and native runs for four out of the seven scenarios (MPlayer, PPT, Excel and Word). However, there are cases, like the IE scenario, which are much slower and users clearly experience the slowdown while loading web pages. On the other hand, the responsiveness measured by duty cycles is worse than what the wall time slowdown suggests. On average, DBT runs are executing 2.4 times more duty cycles than native runs. For reference, the SPEC2000 DBT runs show the same duty-cycle expansion factor as the slowdown factor. The significantly more duty cycles for interactive workloads have both performance and power efficiency implications.

Specifically for each individual benchmark, we observed different runtime behavior. For example, PPT and MPlayer scenarios render real time presentations at certain pace. Their duty cycle expansion can hardly be noticed by users. Rather, it appears as slightly heavier CPU utilization. Excel and Word scenarios are more interactive. They exercise many different features of the tested software for editing or formatting. However, on today’s processors, the extra duty cycles they caused are a relatively short period of real time for end-user experience. The Visual Studio scenario is an interesting case. It invokes Microsoft C/C++ compiler many times to compile the files in our StarDBT project. Each file is compiled by a separate compiler process. Since the StarDBT is a user-mode module inside each process, cross-process translation sharing is not supported. Thus our DBT caused many repeated re-translations during the entire project build. Since the DBT build is more like a batch job, the more duty cycles clearly show as wall time slowdown. Although user-mode DBT causes serious overhead for such cases, it should be in a much better position when translation is done at system level, where translation is managed at the physical memory level. The worst user experience in our benchmarks is IE surfing. In a sense IE is a software framework that loads many other software modules to process different web document sections. This is especially true for today’s webpage designs. Many webpages involve dynamic scripting and plug-in technology that invokes JVM, flash player, PDF readers, and many other document/content process modules. Consequently IE causes the worst slowdown (7X) in terms of duty cycles and more than 2X in terms of wall clock time.

5 Conclusion

Binary translation has many potential applications and it enables new features to computer systems cost-effectively. However, its runtime overhead has long been a concern. Many recent DBT systems have been fairly successful in running CPU intensive workloads. However, performance characterization has not been clear for GUI-based interactive Windows applications.

We have developed a state-of-the-art dynamic binary translation system, StarDBT, which runs on multiple platforms. The user-mode StarDBT demonstrates comparable performance to other cutting-edge runtime translation systems [4], [5], [16], [17] when running the SPEC2000 benchmarks. Additionally, we study performance using popular Windows applications.

In general, we found that current software binary translation technology can provide acceptable responsiveness in many cases. However, clear slowdowns are still experienced for some common user applications. In our evaluation, we collected and analyzed detailed performance data to achieve insight into this subject. We found that dynamic instruction count expansion is the primary performance factor for the DBT runs. This expansion is due to (1) emulation overhead for unsupported instructions and control transfers (for steady state performance in code cache) and (2) translation overhead for big footprint workloads and interactive workloads. We are using the insight from this performance characterization to guide future research that will address these performance issues.

Acknowledgements

We would like to thank Jesse Fang for his strong support of the StarDBT project. We would also like to thank our colleagues at Programming Systems Lab for their contributions to the StarDBT infrastructure.

References

1. Adams, K., Agesen, O.: A Comparison of Software and Hardware Techniques for Virtualization. In: 8th Int'l Symp. on Architecture Support for Programming Languages and Operating System, pp. 2–13 (2006)
2. Altman, E.R., Gschwind, M., Sathaye, S., Kosonocky, S., Bright, A., Fritts, J., Ledak, P., Appenzeller, D., Agricola, C., Filan, Z.: BOA: the Architecture of a Binary Translation Processor. IBM Research Report RC 21665 (2000)
3. Bala, V., Duesterwald, E., Banerjia, S.: Dynamo: A Transparent Runtime Optimization System. In: ACM SIGPLAN Conf. on Programming Language Design and Implementation, pp. 1–12 (2000)
4. Baraz, L., Devor, T., Etzion, O., Goldenberg, S., Skalesky, A., Wang, Y., Zemach, Y.: IA-32 Execution Layer: A Two Phase Dynamic Translator Designed to Support IA-32 Applications on Itanium®-based Systems. In: 36th Int'l Symp. on Microarchitecture, pp. 191–202 (2003)
5. Breuning, D.L.: Efficient, Transparent, and Comprehensive Runtime Code Manipulation. Ph.D thesis, Massachusetts Institute of Technology (2004)
6. Chernoff, A., Hookway, R.: DIGITAL FX!32 Running 32-Bit x86 Applications on Alpha NT. In: USENIX (1997)
7. Cmelik, R.F., Ditzel, D.R., Kelly, E.J, Hunter, C.B., et al.: Combining Hardware and Software to Provide an Improved Microprocessor, US Patent 6,031,992 (2000)
8. Dehnert, J.C., Grant, B., Banning, J., Johnson, R., Kistler, T., Klaiber, A., Mattson, J.: The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges. In: 1st Int'l Symp. on Code Generation and Optimizations, pp. 15–24 (2003)
9. Ebcioglu, K., Altman, E.R.: DAISY: Dynamic Compilation for 100% Architectural Compatibility. In: IBM Research Report RC 20538 (1996) Also: 24th Int'l Symp. on Computer Architecture (1997)
10. Edson, B., Wang, C., Wu, Y., Araujo, G.: Software-Based Transparent and Comprehensive Control-Flow Error Detection. In: 4th Int'l Symp. on Code Generation and Optimizations, pp.333–345 (2006)

11. Henning, J.L.: SPEC CPU 2000: Measuring CPU Performance in the New Millennium. IEEE Computer 33(7), 28–35 (2000)
12. Horspool, R.N., Marovac, N.: An Approach to the Problem of Detranslation of Computer Programs. Computer Journal (August 1980)
13. Klaiber, A.: The Technology Behind Crusoe Processors, Transmeta Technical Brief (2000)
14. Luk, C., Cohn, R., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V., Hazelwood, K.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In: ACM SIGPLAN Conf. on Programming Language Design and Implementation, pp. 190–200 (2005)
15. Qin, F., Wang, C., Li, Z., Kim, H.-S., Zhou, Y., Wu, Y.: LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In: 39th Int'l Symp. on Microarchitecture, pp. 135–148 (2006)
16. Scott, K., Kumar, N., Velusamy, S., Childers, B., Davidson, J., Soffa, M.: Retargetable and Reconfigurable Software Dynamic Translation. In: 1st Int'l Symp. on Code Generation and Optimizations, pp. 36–17 (2003)
17. Sridhar, S., Shapiro, J.S., Northup, E., Bungale, P.: HDTrams: An Open Source, Low-Level Dynamic Instrumentation System. In: 2nd Int'l Conf. on Virtual Execution Environments, pp. 175–185 (2006)
18. SYSMARK 2004, Second Edition, http://www.bapco.com/products/sysmark2004/
19. Wu, Q., Reddi, V., Wu, Y., Lee, J., Conners, D., Brooks, D., Martonosi, M., Clark, D.:Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance. In: 38th Int'l Symp. on Microarchitecture, pp. 271–282 (2005)
20. Ying, V., Wang, C., Wu, Y., Jiang, X.: Dynamic Binary Translation and Optimization of Legacy Library Code in a STM Compilation Environment, WBIA06 in conjunction with ASPLOS (2006)

Unbiased Branches: An Open Problem

Arpad Gellert[1], Adrian Florea[1], Maria Vintan[1], Colin Egan[2], and Lucian Vintan[1]

[1] Computer Science Department, “Lucian Blaga” University of Sibiu, Emil Cioran Street, No. 4, 550025 Sibiu, Romania
{arpad.gellert,adrian.florea,lucian.vintan}@ulbsibiu.ro
[2] School of Computer Science, University of Hertfordshire, Hatfield, College Lane, AL10 9AB UK
c.egan@herts.ac.uk

Abstract. The majority of currently available dynamic branch predictors base their prediction accuracy on the previous *k* branch outcomes. Such predictors sustain high prediction accuracy but they do not consider the impact of unbiased branches, which are difficult-to-predict. In this paper, we evaluate the impact of unbiased branches in terms of prediction accuracy on a range of branch difference predictors using prediction by partial matching, multiple Markov prediction and neural-based prediction. Since our focus is on the impact that unbiased branches have on processor performance, timing issues and hardware costs are out of scope of this investigation. Our simulation results, with the SPEC2000 integer benchmark suite, are interesting even though they show that unbiased branches still restrict the ceiling of branch prediction and therefore accurately predicting unbiased branches remains an open problem.

Keywords: Branch prediction, unbiased branch, branch difference value prediction.

1 Introduction

In a previous paper [1] we showed that a branch in a certain dynamic context is difficult-to-predict when that branch is unbiased and its outcomes are non-deterministically shuffled. A branch is unbiased if its behaviour does not demonstrate a tendency to either the taken or the not taken path. We quantified and demonstrated that the percentages of difficult-to-predict branches in the SPECcpu2000 benchmarks suite [2] are significant (averaging between 6% and 24%, depending on the type of branch prediction context and the prediction context length). We considered the ceiling of history context-based prediction to be around 94% if the feature set length of 28 bits is used. Furthermore, we showed that many current state-of-the-art conventional branch predictors are unable to accurately predict these unbiased branches. This is because current branch predictors only use a limited amount of prediction information, such as local- or/and global-correlations and path-based information. The use of such limited information means that unbiased branches cannot be predicted to a high degree of accuracy. Consequently, other information is required to predict branches which have been classified as unbiased. In this paper we investigate the use of a branch condition sign. The condition sign can be either

© Springer-Verlag Berlin Heidelberg 2007

positive, negative or zero. The condition sign is the difference between the data operands held within each source register. For example, a positive condition sign is computed if the datum in the first source register is greater than the datum in the second source register, and vice-versa for a negative condition sign, and zero if the data show equality. We show that branch behaviour is predictable by predicting the condition sign because branch's output is deterministically correlated with the condition's sign, but the impact of unbiased branches remains significantly high.

2 Related Work

Smith [3] showed that the majority of mispredicted branches come from few static branches. He also showed that a context-predictor where the last '*n*' (as low as 2) data values produced or consumed are used in combination with a closing outer-loop counter can achieve better prediction accuracy than a conventional *gshare* predictor.

Heil [4] introduced the idea of a Branch Difference Predictor (BDP) which simply holds branch source register differences. Heil used these data-value differences as inputs into a Rare Event Predictor (REP). The Rare Event Predictor was used to predict difficult-to-predict branches and the majority of easy-to-predict branches were predicted with a conventional *gshare* predictor. In Heil's study a difficult-to-predict branch was a branch that was mispredicted by a conventional *gshare* predictor. In contrast to Heil, we define in [1] a difficult-to-predict branch to be a branch with a low degree of polarisation since that tends to shuffle between taken and not-taken and is therefore unbiased. Heil used the differences in the register data values as inputs to the REP (up to a maximum of 3 value differences), whereas in our study we use the sign of the differences (up to a history of 256 sign differences) between the register data values. We therefore use less storage and our simulation results show that we achieve better prediction accuracy.

In [5], González introduced the concept of branch prediction through value prediction (BPVP). The idea was to pre-compute a branch outcome by speculatively predicting the source operand as each branch is dynamically encountered. González's prediction strategy was to use a conventional *gshare* in conjunction with a BPVP. The inclusion of the BPVP was to predict the branches that were difficult-to-predict by the conventional *gshare* predictor. González therefore has a similar approach to Heil.

Vintan [6] proposed pre-computing branches by determining a branch outcome as soon that branch's operands were available. The basis behind such pre-computation was that the instruction that produced the last branch source operand would also trigger the branch condition estimation. This means that as soon as this operation was completed then the branch outcome could be immediately resolved. Even though this concept would provide (almost) perfect prediction accuracy, there was a heavy timing penalty in the case when a branch instruction is dynamically executed immediately after the last source operand has been computed, in fact this is a common case.

Gao [7] implemented a Prediction by Partial Matching (PPM) predictor that predicts branch outcomes by combining multiple partial matches through an adder tree. The Prediction by combining Multiple Partial Matches (PMPM) algorithm selects up to L confident longest matches and sums the corresponding counters that are used to furnish a prediction. A bimodal predictor is used to predict branches that are completely biased (either always taken or always not taken) and the PMPM predictor is used to furnish a prediction when a branch is not completely biased. In this study we also implement a PPM predictor, but our PPM predictor has three significant differences. First, our Branch Difference Prediction by Combining Multiple Partial Matches (BPCMP) furnishes predictions for unbiased branches identified in our previous work [1, 8] instead of not completely biased branches. Second, in Gao's study global branch history information was used, whereas we use a combination of global and local branch difference history information. Finally, Gao used an adder tree algorithm to combine multiple Markov predictions, we use one of two voting algorithms.

Jiménez [9] proposed a neural predictor that uses fast single-layer perceptrons. In his first perceptron-based predictor the branch address is hashed to select the perceptron, which is then used to furnish a prediction based on global branch history. Jiménez [10] furthered his work by developing a perceptron-based predictor that uses both local and global branch history information. We also evaluate a perceptron-based predictor, but unlike Jiménez our inputs are based on global and local branch operand difference information. In [11] Jiménez developed a piecewise linear predictor using a piecewise linear function the idea being to exploit different paths leading to the branch undergoing prediction. We have also evaluated a piecewise linear predictor on the unbiased branches as described in [12].

3 Unbiased Branches

In [1] we define an unbiased branch to be a branch that does not demonstrate a bias to either the taken or the not taken path which means unbiased branches show a low degree of polarisation towards a specific prediction context (by which we mean, a local prediction context or a global prediction context or a path-based prediction context) and are therefore difficult-to-predict by that particular prediction context.

We also identified branches that were unbiased on their local and global history contexts and, on their global history XORed with the branch address. Our results showed that even with a feature set length of 28 bits the number of unbiased branches remained significantly high at just over 6%. We therefore considered the ceiling of history context-based prediction to be around 94%.

3.1 Condition-History-Based Branch Prediction Using Markov Models

A context-based predictor [13] predicts the next datum value based on a particular stored pattern that is repetitively generated in the values' sequence. This means that, a context-based predictor could predict any stochastic repetitive sequence. Value predictors that implement the PPM algorithm represent an important class of context-based predictors. In a PPM predictor, if a prediction cannot be furnished by order *k* then the pattern length is shortened and the Markov predictor of order *k-1* is used to furnish the prediction and if this order cannot furnish a prediction the order is further reduced to *k-2* and so on until either a prediction is furnished or the Markov predictor is of the order *0*.

3.1.1 Local Branch Difference Predictor

In Figure 1 we show the mechanism of our local PPM Branch Difference Predictor. The Branch Difference History Table (BDHT) is indexed by the branch address (B_0). In the case of a hit in the BDHT, the last *h* dynamic source operand differences are furnished. To save storage space, sign operand differences are recorded as +1, -1 or 0. For each dynamic branch encountered, a positive difference is recorded if the first source operand is greater than the second, a negative difference is recorded if the first source operand is less than the second source operand and zero is recorded if both operands are the same. The *h* difference fields of the BDHT entry are then used as inputs into our complete-PPM predictor. The PPM predictor furnishes the predicted sign value of the branch undergoing execution (B_0) of order *k*, where $k<h$. Speculative execution of the branch (B_0) only occurs in the case that the pattern length *k* is repeated in the last *h* differences with a frequency greater than or equal to a threshold value.

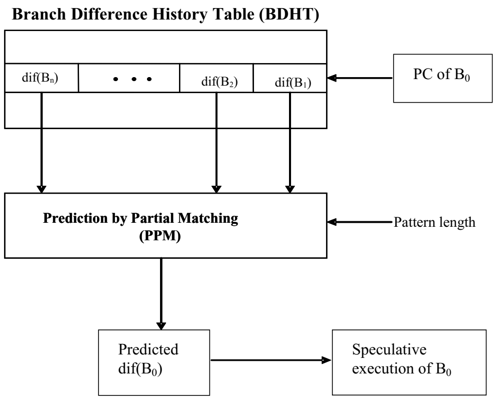

Branch Difference History Table (BDHT)
dif(Bn)
• • •
dif(B2)
dif(B1)
PC of B0
Prediction by Partial Matching (PPM)
Pattern length
Predicted dif(B0)
Speculative execution of B0

Fig. 1. A local complete-PPM branch-difference predictor

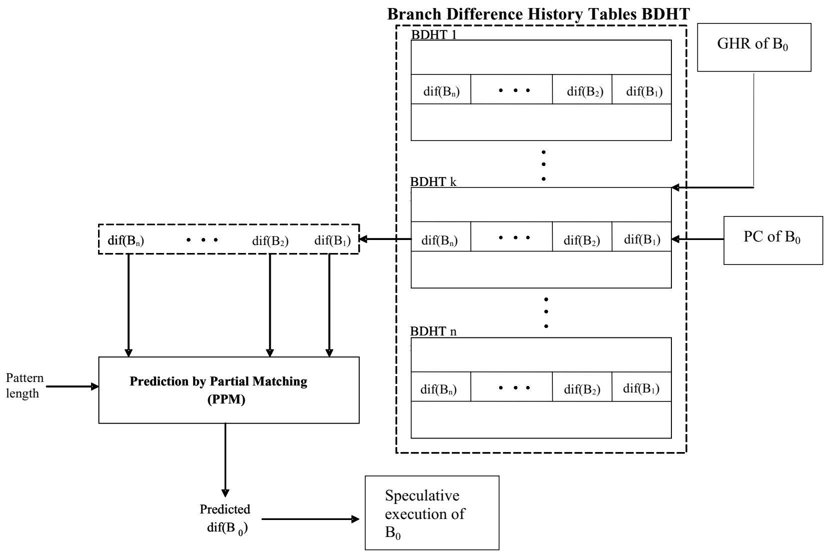

Branch Difference History Tables BDHT
BDHT 1
dif(Bn)
• • •
dif(B2)
dif(B1)
GHR of B0
BDHT k
PC of B0
BDHT n
Pattern length
Prediction by Partial Matching (PPM)
Predicted dif(B 0)
Speculative execution of B0

Fig. 2. A global and local complete-PPM branch-difference predictor

3.1.2 Combined Global and Local Branch Difference Predictor

Figure 2 shows the branch prediction mechanism using a combined global and local PPM-based branch-difference predictor. The Global History Register (GHR) contains the global branch difference history pattern. Every global branch history pattern has its own BDHT and the GHR history pattern is used as an index to its BDHT. Each BDHT is configured as a local BDHT and is accessed as described in section 3.1.1.

3.1.3 Branch Difference Prediction by Combining Multiple Partial Matches

Figure 3 shows our branch prediction mechanism using the Branch Difference Prediction by Combining Multiple Partial Matches (BPCMP). An entry in the BDHT is accessed by the method described in section 3.1.1, but now the *h* branch differences are used as inputs into multiple Markov predictors of different orders (*n* where $n < h$). Each Markov predictor furnishes a predicted sign value (+1, -1 or 0) and these multiple predictions are passed to a voter. The final value prediction is then furnished as the greatest sign frequency that was input into the voter.

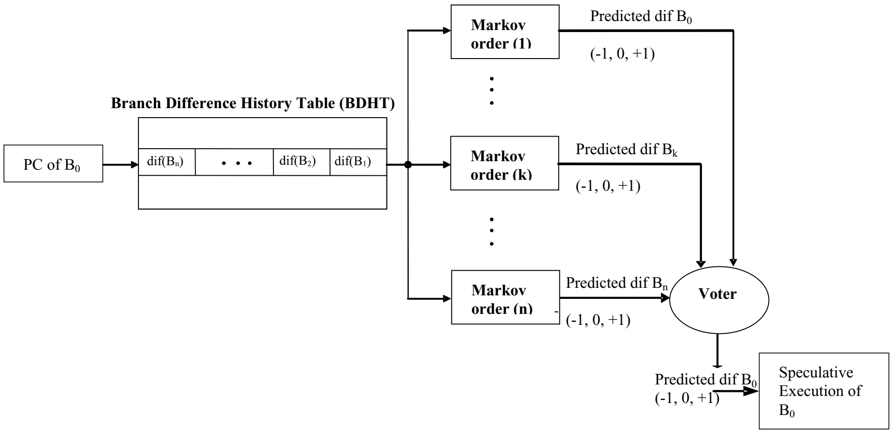


Fig. 3. Multiple Markov branch-difference prediction

We have also investigated a confidence-based voting mechanism. The function field of each entry in the BDHT holds *n* saturated confidence counters, in the range -4 to +4, which are associated with the *n* Markov predictors. For a pattern length *k*, where $1 \leq k \leq n$, the Markov predictors will furnish a value prediction if that repeating pattern is stored at least once in its *h* history values. In the case of a correctly predicted branch, its confidence saturating counter is incremented and decremented in the case of a misprediction. The Markov prediction is then replicated to match its confidence counter, so long as that confidence counter is >0. These multiple value predictions are then passed to the voter, which furnishes the most frequent value prediction.

4 Simulations

We have developed a number of simulators (as described in section 3) which extend the sim-bpred simulator provided in SimpleSim-3.0 [14]. We also include implementations to identify unbiased branches as presented in [1, 8]. We have evaluated our simulators using the unbiased branches we identified in [1] on the SPEC2000 benchmark suite [2]. All simulation results are reported on 1 billion dynamic instructions skipping the first 300 million instructions. We emphasise that our investigation is about the identification and the impact that unbiased branches have on dynamic branch prediction and therefore realistic hardware costs and timings are out of scope of this investigation.

4.1 Local Branch Difference Prediction

We set out to determine the optimal local branch difference predictor. We asked ourselves 5 questions. Would the operand sign value difference algorithm achieve better prediction accuracy than the operand value difference? Which local history register length would provide the best prediction accuracy? Which pattern length would achieve the best prediction accuracy? What is the most suitable threshold value? What is the ideal number of local BDHT entries?

In Figure 4 we answer the first two questions: What would be the most suitable sign algorithm to use and, which history register length achieves the best prediction accuracy? We identified unbiased branches the same way as in our previous work [1], and we evaluated the impact of these unbiased branches using a complete PPM predictor with a local BDHT. The BDHT we used was sufficiently large to ensure that every static branch had its own entry thereby eliminating any possibility of collisions. The pattern length was set to 3, the threshold value was set to 1, and the local history register length was varied from 8-signs to 64-signs in increments of 8. Our results show that better prediction accuracy is achieved by the operand sign difference algorithm rather than the operand value difference algorithm and that beyond a local history register length of 24-signs there is only marginal improvement in prediction accuracy. The reason that the operand sign difference algorithm outperforms the operand value difference algorithm is due to the increased amount of correlation information used by the sign difference algorithm. The frequency of information used

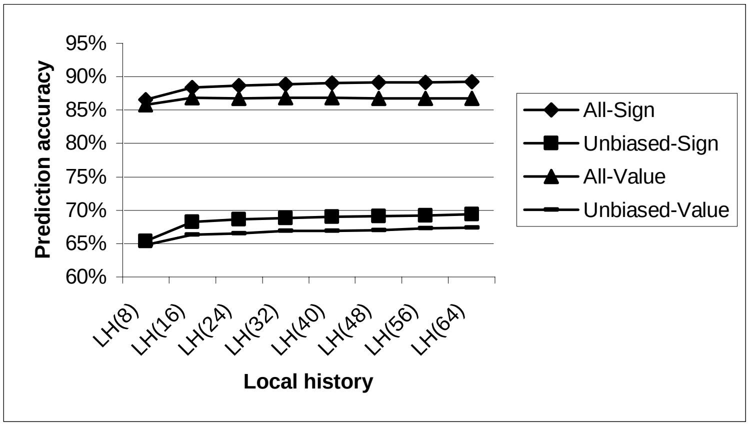


Fig. 4. Average difference prediction accuracy with increasing local history register length of the sign difference and operand difference algorithms

by the operand value difference algorithm is low and therefore correlation is low, whereas the frequency of information used by the operand sign difference algorithm is high and therefore correlation is high.

In Figure 5 we answer the third question: Which pattern length would achieve the best prediction accuracy? We used a complete PPM predictor with the operand sign

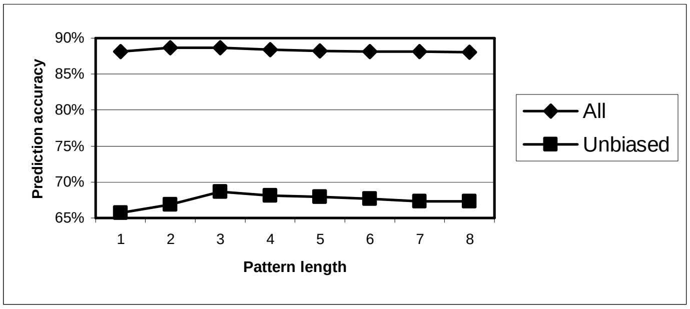


Fig. 5. Average difference prediction accuracy with increasing pattern length

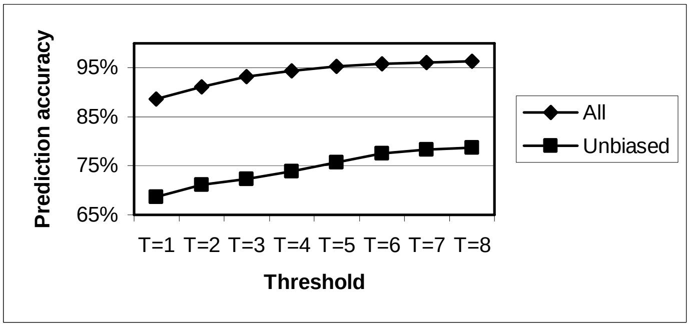


Fig. 6. Average difference prediction accuracy with increasing threshold value

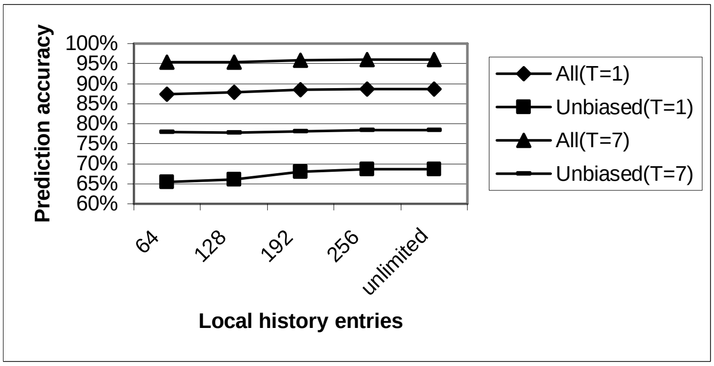


Fig. 7. Average difference prediction accuracy with an increase in the number of local BDHT entries

difference algorithm, a local history register length of 24-signs and the threshold value was set to 1. Our results show that initially prediction accuracy improves with increasing pattern length and then decreases and these results confirm that our original pattern length of 3 achieves the best prediction accuracy.

In Figure 6 we answer the fourth question: What is the most suitable threshold value? We used the same parameters as Figure 5, but the pattern length was now set to 3 and the threshold value varied. Our results show that prediction accuracy improves with an increasing threshold value, but there is marginal, if any, benefit of increasing the threshold value beyond 7.

In Figure 7 we answer the final question: What would be the optimal number of entries in the local BDHT? We used the same parameters as Figure 6, and the number of entries in the local BDHT was varied from 64 entries to 256 entries in increments of 64. We also include an unlimited local BDHT. Our results show that the impact of the so called 3Cs (capacity, collisions and cold-start) to be minimal with a 256 entry local BDHT and that there is minimal prediction accuracy gain by increasing the number of entries beyond 256 entries where the increased number of cold-start mispredictions may impact on prediction accuracy.

We investigated the branch prediction accuracies of the individual SPEC2000 benchmarks using our optimal local branch difference predictor. We used the operand sign difference algorithm, with a local history register length of 24-signs, a pattern length of 3, and we use a local 256 entry BDHT. In our results we compare two threshold values, 1 and 7. When the threshold value is 1, we achieve an average branch prediction accuracy of 90.55% and the unbiased branches have an average branch prediction accuracy of 71.76%. When the threshold value is increased to 7, we achieve an average branch prediction accuracy of 96.43% and the unbiased branches have a prediction accuracy of 76.69%. These results show the significance of the threshold value on prediction accuracy and the impact of unbiased branches. Consequently, unbiased branches in this local context remain difficult-to-predict.

4.2 Combined Global and Local Branch Difference Prediction

We consider the high number of unbiased branches and their impact on prediction accuracy to be due to their high degree of shuffling. To alleviate the problem of shuffled branch behaviour of unbiased branches we have developed a combined global and local branch difference predictor which would convert an unbiased branch in a local context into a biased branch in a global context, and therefore a difficult-to-predict branch in a local context would be an easy-to-predict branch in a global context.

In our global and local branch difference predictor, each global history register pattern is used to point to its own local BDHT as described in section 3.1.2 and shown in Figure 2. Consequently, we restrict the global history register length to a maximum of 4-signs. The parameters of each of the local BDHTs were the same as those which achieved the results shown in Figure 7, except we used a 256 entry BDHT.

In Figure 8 the global history register length of 0 represents the optimal local branch difference predictor whose results are provided in Figure 7, with a 256 entry BDHT. With the combined global and local difference predictor, as the global history

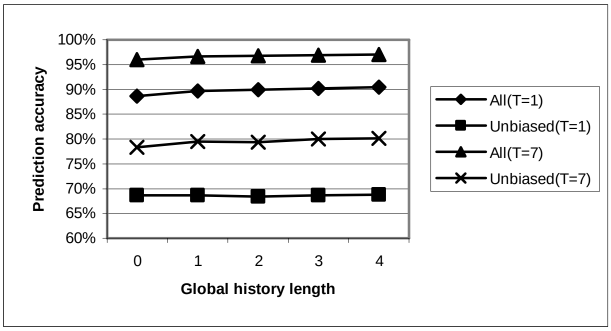


Fig. 8. Combined global and local difference prediction accuracy

register length is increased there is a marginal improvement in prediction accuracy. With a global history register length of 4-signs and a threshold value of 1, the combined global and local branch difference predictor achieves an average prediction accuracy of 90.47%, but the unbiased branches only achieve an average prediction accuracy of 68.81% showing a marginal improvement over the local branch difference predictor. When the threshold value is increased to 7, the average prediction accuracy improves to 97.44% and the average prediction accuracy of unbiased branches is still significant at 81.25%. Even though there is some improvement in prediction accuracy, these results show that the impact of unbiased branches still remains significant and therefore implies that alternative approaches are required.

4.3 Branch Difference Prediction by Combining Multiple Partial Matches

Our first alternative approach was to develop a branch difference predictor using five Markov predictors of orders ranging between 1 and 5 (as described in section 3.1.3 and shown in Figure 3). Again, we use a 256 entry local BDHT, a local history register length of 24-signs; we compare the prediction accuracy of two voting

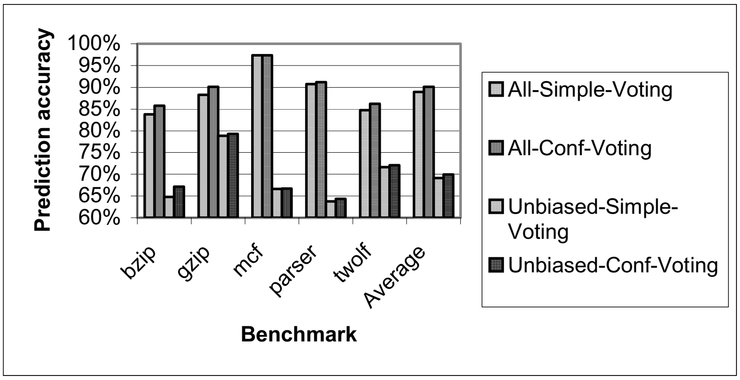


Fig. 9. Difference prediction accuracies by combining multiple partial matches through simple voting and confidence-based voting

algorithms, a simple voting algorithm and a confidence voting algorithm. Our results show that the average prediction accuracy of the confidence voting algorithm is marginally better than the simple voting algorithm, as shown in Figure 9.

4.4 Neural-Based Branch Difference Global and Local Prediction

In our second alternative approach we developed a family of neural-based branch difference predictors. Our neural predictors are fast single-layer perceptron predictors similar to those developed by Jiménez [9]. For a fair comparison with our 256 entries local BDHT we use a perceptron table with 256 entries. Our single-layer perceptron predictors use global history information only or local history information only or a combination of global and local history information.

To determine our optimal single-layer perceptron predictor, we vary the input history register lengths. Not surprisingly, the combination of global and local history information outperforms the other two predictors. We found that the best average prediction accuracy of 92.58% was achieved with a 40-global history signs combination with 28-local history signs. However, the unbiased branches still have a significant impact with an average prediction accuracy of 73.46%.

Finally, we considered the impact of unbiased branches on a piecewise linear predictor based on [11]. We dynamically changed the global history input from 18- to 48-bits combined with local history input from 1- to 16-bits. We achieved an average prediction accuracy of 94.2% on all branches but the impact of unbiased branches still remained significant at 77.3%.

4.5 Comparing All of the Optimal Predictors

In Figure 10, we bring together the impact that unbiased branches have on all of the optimal predictors we have developed (local-PPM, combined-PPM and multiple Markov combined- perceptron and the piecewise linear branch predictor). Our results show that unbiased branches have a severe impact on all branch predictors and in all cases unbiased branches only have an average branch prediction accuracy of between 71.54% (local-PPM) and 77.3% (piecewise linear branch predictor).

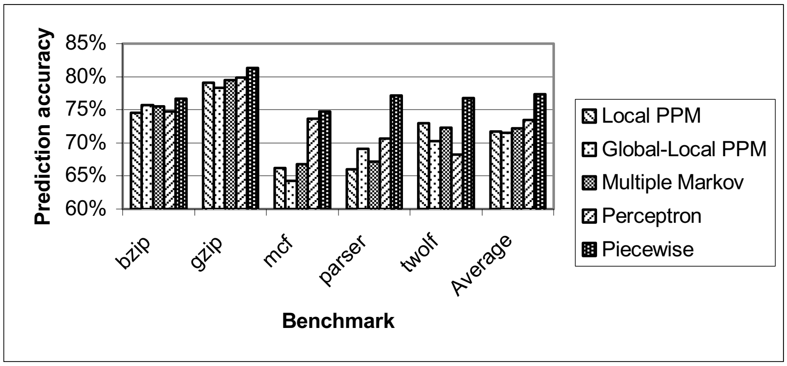


Fig. 10. Branch prediction accuracy on unbiased branches

5 Conclusion

In this study we have validated our previous findings in [1, 8] that current state-of-the-art branch predictors correlate either insufficient information or wrong information in the prediction of unbiased branches. This led us to consider alternative approaches: the branch difference predictors using PPM and multiple Markov predictors and neural-based perceptron predictors. Our results show that unbiased branches still limit prediction accuracy even with these alternative approaches. The most effective branch predictor was the piecewise linear branch predictor, but even this predictor only achieved a prediction accuracy of 77.3% on the unbiased branches.

However, we have shown that the sign difference algorithm achieves better prediction accuracy than the operand difference algorithm. We also show that combined global and local information achieves better prediction accuracy than global information alone or local information alone.

In our opinion, the most optimal local branch difference predictor uses the operand sign difference algorithm, with a local history register length of 24, a pattern length of 3, a threshold value of 7 and a local BDHT with 256 entries. This predictor achieves an average prediction accuracy of 96.43% on all branches but on the unbiased branches only achieve a prediction accuracy of 79.69%.

Also in our opinion, the impact of unbiased branches significantly restricts prediction accuracy. This means that accurate branch prediction of unbiased branches remains an open problem and such branches will continue to limit the ceiling of dynamic branch prediction. Perhaps an alternative mechanism might be to hand-shake scheduler support with dynamic branch prediction. The idea of the scheduler would be to remove as many branch instructions from the static code as possible and leave the remaining branches to be dynamically predicted. Yet another alternative could be to pursue the concepts of micro-threading [15] where small fragments of code are executed concurrently and the branch problem is no longer a major concern.

References

[1] Vintan, L., Gellert, A., Florea, A., Oancea, M., Egan, C.: Understanding Prediction Limits through Unbiased Branches, Lecture Notes in Computer Science. In: Jesshope, C., Egan, C. (eds.) ACSAC 2006, vol. 4186-0480, pp. 480–487. Springer, Heidelberg (2006)

[2] SPEC2000, The SPEC benchmark programs http://www.spec.org

[3] Smith, Z.: Using Data Values to Aid Branch-Prediction, MSc Thesis, Wisconsin-Madison, USA, pp. 1–103 (1998)

[4] Heil, T.H., Smith, Z., Smith, J.E.: Improving Branch Predictors by Correlating on Data Values. In: The 32nd International Symposium on Microarchitecture, pp. 28–37 (1999)

[5] González, J., González, A.: Control-Flow Speculation through Value Prediction. IEEE Transactions on Computers 50(12), 1362–1376 (2001)

[6] Vintan, L., Sbera, M., Mihu, I.Z., Florea, A.: An Alternative to Branch Prediction. Pre-Computed Branches, ACM SIGARCH Computer Architecture News 31(3), 20–29 (2003)

[7] Gao, H., Zhou, H.: Prediction by Combining Multiple Partial Matches. In: The 2nd Journal of Instruction-Level Parallelism Championship Branch Prediction Competition (CBP-2), Orlando, Florida, USA, pp. 19–24 (2006)

[8] Gellert, A.: Prediction Methods Integrated into Advanced Architectures, Technical Report, Computer Science Department, ”Lucian Blaga” University of Sibiu, pp. 37–70 (2006)
[9] Jiménez, D., Lin, C.: Dynamic Branch Prediction with Perceptron. In: Proceedings of the Seventh International Symposium on High Performance Computer Architecture (HPCA-7), pp. 197–206 (2001)
[10] Jiménez, D., Lin, C.: Neural Methods for Dynamic Branch Prediction. ACM Transactions on Computer Systems 20(4), 369–397 (2002)
[11] Jiménez, D.: Idealized Piecewise Linear Branch Prediction. Journal of Instruction-Level Parallelism 7, 1–11 (2005)
[12] Gellert, A.: Integration of some advanced prediction methods into speculative computing systems, Technical Report, Computer Science Department, ”Lucian Blaga” University of Sibiu, pp. 1–40 (2007)
[13] Sazeides, Y., Smith, J.E.: The Predictability of Data Values. In: Proceedings of the 30th Annual International Symposium on Microarchitecture, pp. 248–258 (1997)
[14] Simplescalar. The SimpleSim Tool Set ftp://ftp.cs.wisc.edu/pub/sohi/Code/simplescalar
[15] Bousias, K., Hasasneh, N., Jesshope, C.: Instruction Level Parallelism through Microthreading—A Scalable Approach to Chip Multiprocessors. The Computer Journal 49(2), 211–232 (2006)

An Online Profile Guided Optimization Approach for Speculative Parallel Threading

Yuan Liu, Hong An, Bo Liang, and Li Wang

Department of Computer Science and Technology
University of Science and Technology of China
Hefei, Anhui, China
Key Laboratory of Computer System and Architecture
Chinese Academy of Sciences
Beijing, China
naga@mail.ustc.edu.cn,han@ustc.edu.cn

Abstract. Speculative parallel threading technique has been proposed to speed up hard-to-parallelize applications running on multi-core chips. Traditionally offline profiling approach provides necessary information for the optimizations used in speculative parallelization. However, the offline profiling can't address the applications without appropriate training input. We propose an online profile guided optimization approach to address this problem, which performs profiling and optimizing at runtime and doesn't need an individual profiling pass as well as good training inputs. In our design, programs run in a manner of two continuous phases which are profiling phase and optimized execution phase respectively. Furthermore, our approach can also detect at runtime the behavior change of programs parallelized speculatively. Next the execution flow will be transferred to a new optimized version more suitable to this change. The evaluation shows that the ability of this approach is comparable to the traditional offline implementation. So we believe that this approach is able to serve as an individual guide to speculatively parallelize the applications when traditional offline profiling is unavailable.

Keywords: Thread-level parallelization, profile guided dynamic optimization.

1 Introduction

1.1 Profile Guided Speculative Parallelization

Speculative parallel threading technique [1][2][4] has been proposed to enhance the performance of hard-to-parallelize programs on the CMP platforms. It allows to parallelize regions of code in the presence of ambiguous data dependence. Speculative threads run in parallel, and commit sequentially in the original order of the program. Other threads will check whether the read-after-write dependence is violated when an earlier thread commits, and restart when suffering dependence/break violations. Since the complete dependence relations don't need to

© Springer-Verlag Berlin Heidelberg 2007

be decided ahead of time, the compiler can partition easily some code regions into threads, such as iterations of loops, even if this partition may bring a lot of violations at runtime. Transactional memory [9] can effectively support the speculative mechanism. Entire codes or parts of them in the speculative thread are taken as transactions, and the transactions commit in the logical order among threads. As each transaction completes, it writes all of its newly produced states to shared memory atomically.

When using transaction memory, codes of threads can be split into two regions, transaction region and ordered region. Codes in transaction are speculatively parallel, while those in ordered region are executed sequentially in logical order among threads. The memory accesses which frequently cause data dependence violation can be moved into ordered region of threads, so the corresponding dependence violation will be avoided. This optimization is called transaction partition, which reduces the restart rate of speculative threads at the cost of parallelism loss. To gain the optimal effect, it must first identify which references are frequent violation candidates (FVC), thus to move them. Static compiling can tell only a small proportion of FVCs. Profiling technique can assist the compiler to overcome this problem [6][10]. It collects information about the behavior of a program from its past execution. Information about the crossing-iteration data dependence probabilities could effectively guide the optimizations such as transaction partition.

The traditional profiling method [6] is offline, i.e. it needs first to run the program once to collect information, afterwards feed back the results to compiler for optimization. The offline method requires a training input in this trial run. But it is not easy to get the representative training input for a program. If the training input doesn't represent the actual workload, the program may be optimized incorrectly. For example, many commercial programs have multiple functions to fit various requirements, and there isn't a general training input for profiling. This offline profiling method prevents the speculative parallel threading from being applied on applications without appropriate training input, even though the behaviors of these programs are predicable actually when they are running.

1.2 Related Work

Du et al. proposed a cost-driven compilation framework [1][5][6] to select and to transform the profitable loop for speculative parallel threading. It is implemented by a static compiler with aids of profiling in advance. The offline profiling process provides data dependence information and control flow reaching probabilities which are used to annotate the cost graph of a loop. Compiler selects the optimal partition through calculating the mis-speculation cost of every possible partition. Compared with it, our design selects the optimal transaction partition at runtime, so using the optimized codes needn't wait until the next execution of programs.

Java runtime parallelizing machine (JRPM) [3] is a dynamic compilation system supported by hardware profiler. It is based on CMP with thread-level speculating supports. This system uses two-phase profile guided optimization to boost the performance of speculative threads. It collects information from initial execution to identify those good loops to parallelize. Only loops with high predicted speedup are recompiled into speculative threads.

1.3 Objective of This Study

The goal of this study is to design a more flexible profile guided optimization mechanism to improve the performance of speculative multi-threading execution when the offline method is unavailable. The online profiling method [3][7]is promising because it doesn't need a training input or a separate trial run. It profiles only an initial short phase of program execution to predict behaviors of the whole program, and its results are used immediately at runtime to optimize the rest of the execution.

In this paper, we propose a continuous two-phase profile guided optimization framework on a speculative multithreading execution platform based on transaction memory. The framework includes online profiling and dynamic optimization for speculative parallelization of loops. To reduce runtime overhead, it generates possible optimized loop versions in advance, and at runtime selects an appropriate version to execute according to initial profiling results. This design supports dynamic transaction partition for speculative parallel threading, and other optimizations such as value prediction and triple regions partition.

The rest of this paper is organized as follows. Section 2 describes the methodology and implementation of our framework in detail. Section 3 evaluates the design using several benchmarks. Section 4 concludes this paper.

2 Continuous Two-Phase Profile Guided Optimization Approach

Our approach of online profile guided optimization is a continuous two-phase course. In the initial phase, only a thread executes the program's sequential code version which contains profiling codes. After initial profiling phase, it halts and incurs an optimizing routine. The optimizing routine predicts the following behavior of the program through the initial profiling result, thus to perform corresponding optimizations on original version. Afterwards the program enters the next phase and resumes the execution using this new optimized code version without profiling. If a change of program behavior is observed in the optimized execution phase, the program will reenter the profiling phase, and repeat the above process. In continuous two-phase optimization, *two-phase* refers that the thread is executed in either of two phases: profiling phase or optimized execution phase, while *continuous* refers that a new profiling-optimizing cycle could be triggered again with the change of runtime characteristic of the program.

This section first briefly introduces the speculative multithreading execution model used in our design. Next a detailed description about our online profile guided optimization framework is given. Additionally, two problems this framework encounters and the solutions are provided in the following subsection. The last subsection describes some extensions to the original design.

2.1 Speculative Parallel Execution Model

The speculative execution model used by threaded programs is depicted in Fig. 1. Iterations of loops are interleavingly allocated into every thread, and codes of an iteration are divided into transaction region and ordered region. A thread first executes the transaction region, and then commits the writes of transactions at the beginning of ordered region. It can't enter the ordered region until the threads executing the earlier iterations have finished this region and passed the token to it. If dependence violations are detected during commitment of other threads, the thread will re-execute the transaction region.

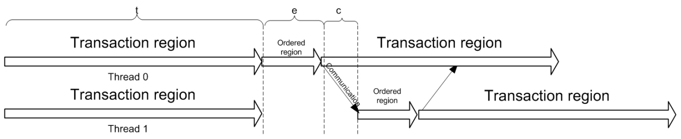


Fig. 1. Speculative execution model with 2 threads

2.2 Dynamic Optimization Framework

An overview of the continuous two-phase profile guided optimization framework is given in Fig. 2.

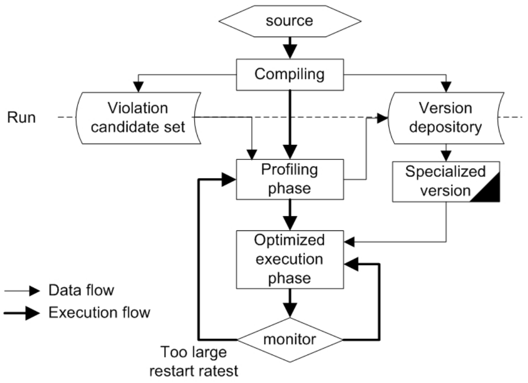


Fig. 2. Continuous two-phase profile guided optimization framework

Example of a loop for speculative parallel threading

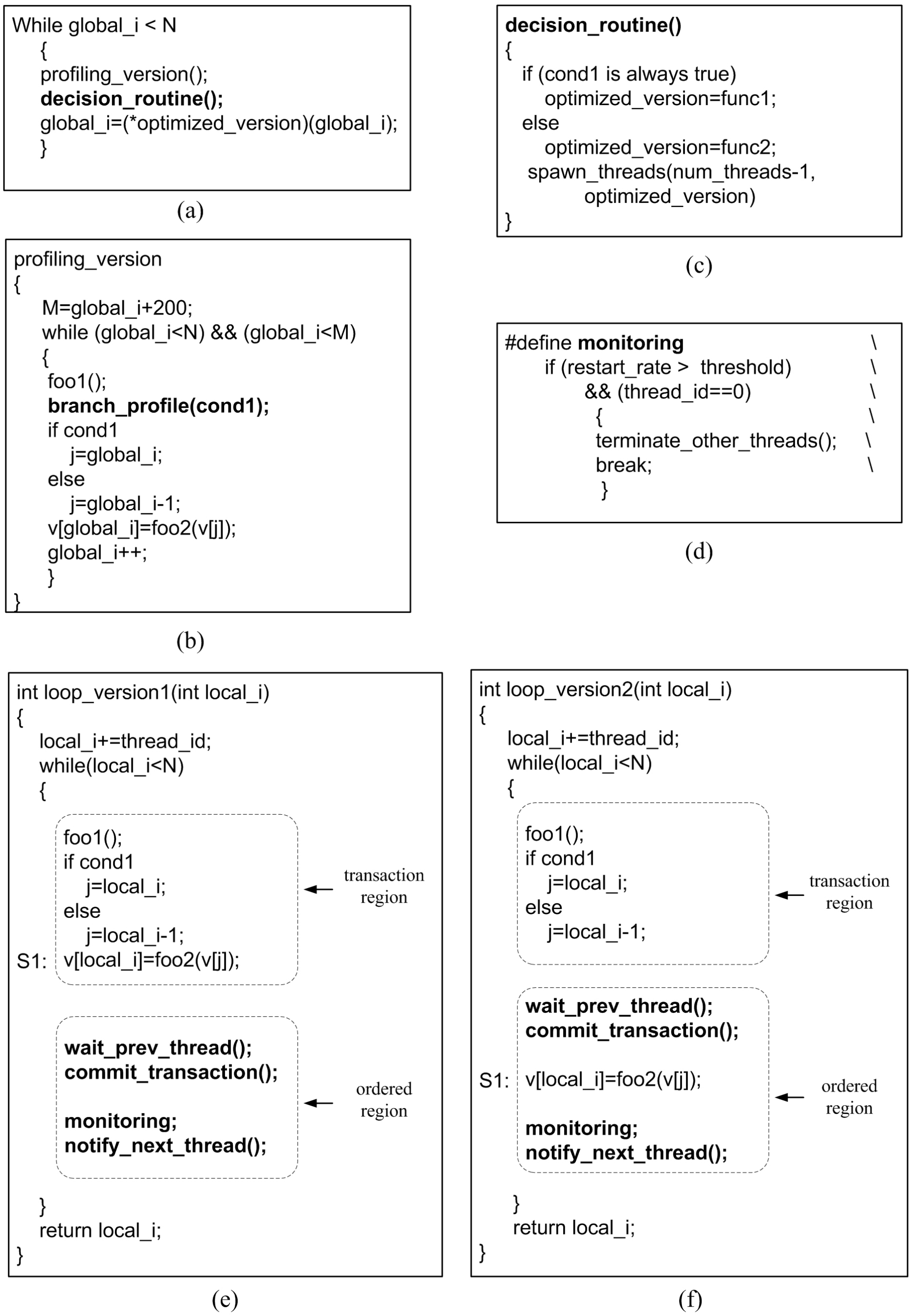
```
while (i<N)
  {
   foo1();
   if cond1
     j=i;
   else
     j=i-1;
S1:v[i]=foo2(v[j]);
   i++;
   }
```

First, the compiler identifies the loop candidates for speculative execution and their potential violation candidates. We take the loop code in the above example to show the code transformation in static compiling and the dynamic optimization at runtime. In this loop, function `foo1` is iteration-independent, but the statement `S1` will carry the crossing-iteration dependence if `cond1` is false. According to the probability that this condition is true, it could be decided whether moving `S1` into the transaction region. But this probability is unknown in static compiling stage.

Afterward compiler inserts profiling code into the original loop body to form the profiling code version which is shown in Fig. 3(b), and generates ahead-of-time optimized multithreading code versions shown in Fig. 3(e)(f). The details about ahead-of-time optimizing technique will be described in the following subsection.

As Fig. 3(a) shows, Two-phase profile guided optimization process is driven by a main loop. In the beginning, profiling version function is executed by thread 0 alone, i.e. the execution is sequential. Besides the original computing, the extra inserted profiling codes collect information about intra-iteration control flow and crossing-iteration data dependence. In Fig. 3(b), the function `branch_profile` will record the times when the `cond1` is true. This initial profiling phase lasts a fixed amount of iterations, such as 200.

When profiling phase ends, a runtime decision routine like the function `decision_routine` in Fig. 3(c) is called. It analyzes the collected profiling results to decide current frequent violation candidate sets (FVC). According to the mapping correlation between different FVC sets and ahead-of-time optimized code versions, it chooses a currently optimal version for the execution of rest iterations. The version may be a speculative parallel version or the original sequential version. The latter means that this loop can't boost performance through the speculative execution, and the rest of its execution will be sequential. As an example, `decision_routine` decides whether the `cond1` is always true. If so, the reference `v[j]` is not a FVC, and the statement `S1` could be moved into transaction. Thus, the function `loop_version1` in Fig. 4(e) will be called in the main loop. Otherwise, the function `loop_version2` in Fig. 3(f) with smaller transaction region will be selected.



```
While global_i < N
    {
    profiling_version();
    decision_routine();
    global_i=(*optimized_version)(global_i);
    }
```

(a)

```
profiling_version
{
    M=global_i+200;
    while (global_i<N) && (global_i<M)
    {
     foo1();
     branch_profile(cond1);
     if cond1
        j=global_i;
     else
        j=global_i-1;
     v[global_i]=foo2(v[j]);
     global_i++;
     }
}
```

(b)

```
decision_routine()
{
  if (cond1 is always true)
     optimized_version=func1;
  else
     optimized_version=func2;
  spawn_threads(num_threads-1,
          optimized_version)
}
```

(c)

```
#define monitoring                        \
     if (restart_rate >  threshold)       \
          && (thread_id==0)               \
            {                             \
            terminate_other_threads();    \
            break;                        \
            }
```

(d)

```
int loop_version1(int local_i)
{
   local_i+=thread_id;
   while(local_i<N)
   {
      foo1();
      if cond1
         j=local_i;
      else
         j=local_i-1;
  S1: v[local_i]=foo2(v[j]);

      wait_prev_thread();
      commit_transaction();

      monitoring;
      notify_next_thread();

   }
   return local_i;
}
```

(e)

```
int loop_version2(int local_i)
{
   local_i+=thread_id;
   while(local_i<N)
   {
      foo1();
      if cond1
         j=local_i;
      else
         j=local_i-1;

      wait_prev_thread();
      commit_transaction();

  S1: v[local_i]=foo2(v[j]);

      monitoring;
      notify_next_thread();
    }
    return local_i;
}
```

(f)

Fig. 3. Loop codes transformed for speculative threading execution: (a) describes the main loop; (b) describes the code version executed in profiling phase; (c) describes the decision routine; (d) describes monitoring code; (e) and (f) give two optimized version with different transaction partitions

If a speculative parallel version is selected, the loop will enter optimized execution phase. In this phase the loop will run as the execution model in Fig. 1. However, a piece of additional monitoring codes is inserted into ordered region of iterations. Although initial profiling results have supported the speculative execution, the execution manner still needs to be adjusted when the behavior of the loop changes significantly. The monitoring codes check this change by calculating current restart rate, which is shown in Fig. 3(d). If the restart rate exceeds the threshold that current code version can bear, it is very possible that loop behavior characteristic identified in last profiling phase has changed. The current version is not suitable any more, so the loop will perform a new profiling-optimized execution cycle again for the rest iterations. In the Fig. 3, the function `optimized_version` returns after the monitoring code breaks the loop, So the main loop will perform the next iteration. This way endows the framework with the ability to optimize the loop with phase-changed behavior.

2.3 Accuracy of Initial Profiling and Its Effective Implementation

We evaluated the ability of initial profiling on predicting data dependence hazards against the offline one. These dependence relations are unable to be identified in static compiling stage because of unsolved condition branches and ambiguous addresses from alias. Crossing-iteration dependence probabilities in loops can be gained directly by dependence profiling based on address comparison, or indirectly through the branch probabilities from control flow profiling.

For control flow profiling, recent studies [7] have already indicated that for SPEC CPU2000 benchmark even very short initial profiling has comparable accuracy to the traditional full profiling when predicting the hot path in program execution. Furthermore, we consider about the matching degree of the results from initial direct data dependence profiling and those from full profiling. Mismatch rates for some applications in SPEC CPU2000 benchmark are given in Fig. 4, which shows a mean value of 5.6%. It is small enough to believe in the ability of the initial profiling.

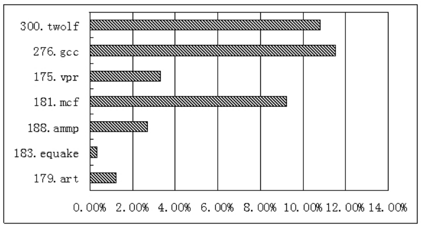


Fig. 4. Mismatch rates for frequent violation candidate

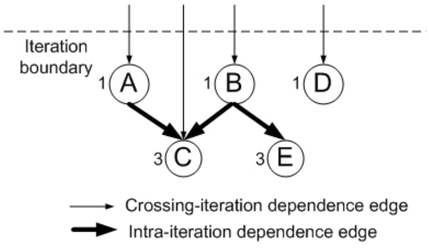

Iteration boundary
1 A
1 B
1 D
3 C
3 E
Crossing-iteration dependence edge
Intra-iteration dependence edge

Fig. 5. A dependence graph example for computing partition cost

We use both edge profiling and data dependence profiling to obtain the data dependence probability. For direct data dependence profiling, the instrumentation-based software implementation is adopted, which approximates to the tool proposed in [8].

2.4 Speculative Code Versions Generating in Static Compiling

After gaining the initial profiling results, how to use them at runtime to optimizing codes is another challenge. Overhead introduced by a complicated optimizing procedure will easily spend the performance gains from the rest of codes execution it generates; even the total effect of optimizing procedure and its output are negative.

We attempt the technique of ahead-of-time optimization to solve this problem. In static compiling stage every possible runtime characteristic is predicted, and individual specialized code versions are generated for them. At runtime, the version with respect to the actual program behavior will be triggered to run.

As shown in Fig. 3. compiler generates multiple loop versions with different transaction partitions corresponding to possible FVC sets, and the decision routine will select the corresponding version for the rest of the execution according to actual FVC sets given by initial profiling phase. This technique which at runtime decides to use some partition is called dynamic transaction partition.

For a violation candidate, the following code transformations may be performed in static compiling stage:

R1: *fully parallelize the loop, without explicit ordered region, if the violation rarely occurs;*
R2: *move the candidate and its dependence descendants into ordered region, if the violation is frequent and the size of ordered region isn't beyond the threshold;*
R3: *under other conditions, abandon this loop for speculatively parallelization.*

To calculate the threshold in rule R2, an approximate formula estimating the performance gain of the speculative parallelized loop is given as follows:

$$speedup = \begin{cases} \dfrac{lp}{t+e+c+l\alpha}, & t > (p-1)e + pc \quad \text{(1a)} \\ \dfrac{l}{e+c+l\alpha}, & t \le (p-1)e + pc \quad \text{(1b)} \end{cases}$$

Where
t : execution time of transaction region,
e : execution time of order region,
l : original sequential execution time of the iteration,
c : communication delay to pass committing token,
p : number of current threads,
α : restart rate of the loop.
t, e and c have been denoted in Fig. 1.

When there is only one violation candidate and it has been moved into ordered region, restart rate α is 0. So we can conclude that the threshold of order region's execution time e is $l-c$ or $lp-c-t$, which is used by rule R2. If a FVC introduces ordered region with size beyond this threshold, the rest iterations of the loop have to execute the original sequential version.

Now we consider about the case when the initial profiling gives a result with a combination of FVCs. Theoretically, N FVCs will generate 2^N combinations and every combination should have one individual version, and lead to a code explosion. But this is not true, because most combinations lead to the original sequential version due to large size of the ordered region. For the actual applications, it is enough to consider the combination containing only 2 FVCs, in some cases, more than 2.

Figure 5 shows a loop dependence graph example, where each node is annotated with its own size (execution time). The threshold is 5. According to the above rules, only FVCs combinations A, C and C, D require generating a new version. As listed in table 1, FVCs set {A,C} corresponds a version with ordered region {A,C}, While all combinations containing B lead to the sequential version due to its large partition cost.

Table 1. Mapping from FVC sets to partition versions

A	B	C	D	ordered region version
*	1	*	*	{A,B,C,D,E}
1	0	1	0	{A,C}
0	0	1	1	{D,C}
...	...	...	...	{A,B,C,D,E}

Table 2. mapping from FVC sets to partition versions with value prediction

A	B	C	D	ordered region version
0	1	0	0	profile B
1	0	1	0	{A,E}
0	0	1	1	{D,C}
0	predicable	0	0	predict B
...	...	...	...	{A,B,C,D,E}

2.5 Extensions

We have also implemented software value prediction as a complement to dynamic transaction partition optimization. Value prediction plays a very important role among many efforts to improve speculative performance, which has been proved in [5].

If the value used by a FVC is predicable, the predicted value could be speculatively used in transaction region. And the thread doesn't need to read this reference in ordered region. This method could solve the problem that FVCs with large partition cost reduce parallel degree, and could enable the sequential version to become a speculative parallel one. Whether the value of a FVC is predicable and what its change pattern is are identified usually by value profiling. Value profiling and value prediction have been embedded into our optimization framework, which can deal with last-value change pattern and stride pattern.

For example in Fig. 5, the correlations of FVCs and versions are modified to table 2. If B is identified as a FVC, an additional value profiling procedure will be performed by the prolonged profiling phase. Next if B is identified predicable, new version with value prediction will replace original sequential version to run in optimized execution phase.

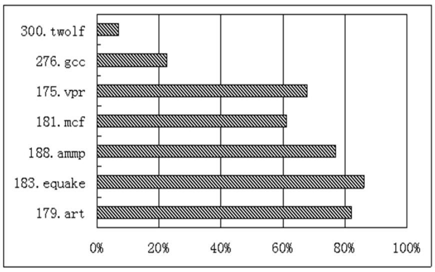
Another extension to our design is to add an extra ordered region similar to the pre-fork region in [6] during the speculative execution of an iteration. If a FVC results in a version with too large ordered region, even the sequential version, its dependence source will be considered about to move. If the source has enough small partition cost, the pre-fork region will be built for it.

3 Experiments and Results

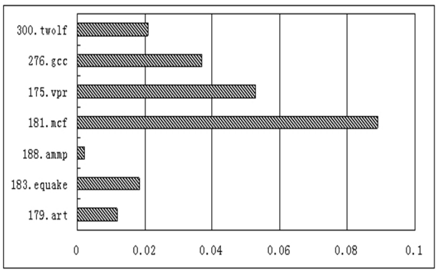
To evaluate the design, we generate speculative parallel threaded codes and run them on a simulator. The simulator models a CMP whose core executes one instruction per cycle. The reason of using this structure is to emphasize the effect of thread level parallelism on performance and to weaken the influence of instruction level parallelism or memory hierarchical details. A simulator with 1 IPC is competent to achieve information about speculative threads behavior, such as restart rates, ordered region ratios and so on.

Four integer applications and three float pointing applications in SPEC CPU2000 benchmark suite are chosen for the test. The potential violation candidates information is obtained through the conservative analysis of the compiler. The multi-version transformation of the loop codes is performed manually on the source level. Since the optimal transaction partition is the common optimization target for our design and that in [6], the results are compared with those given in it. The design proposed in [6] can support only two parallel threads, so we also provide the results on two threads. However, our design supports arbitrary numbers of threads.

First the ability of our approach to find good speculative parallel loops is checked. Figure 6(a) shows the runtime coverage of the loops that are speculative executed. Our framework can speculatively parallelize the loops that cover over 70% total execution time for float pointing application. For integer ones, vpr (place) and mcf give the results of over 60%. Coverages for gcc and twolf fall down, but they are not so bad comparing with the results in [6].



(a)



(b)

Fig. 6. (a) Runtime coverages of speculative parallel loops; (b) Average ratios of ordered region size to loop body size

Figure 6(b) describes the average results of dynamic transaction partition, and Fig. 7(a) also gives the total effect of two kinds of optimization transaction partition and value prediction on reducing the restart rate. For the float pointing applications,which could be easily parallelized, ordered region occupying average 1.07% of the loop body size reduces the restart rate to 0.1%. For integer applications, mcf has the largest ordered region, which provides restart rate 0.35% . It is as if twolf should restart more frequently, but the actual rate is still small. This proves that our framework is competent to improve the behaviors of speculative threads with little parallel degree loss.

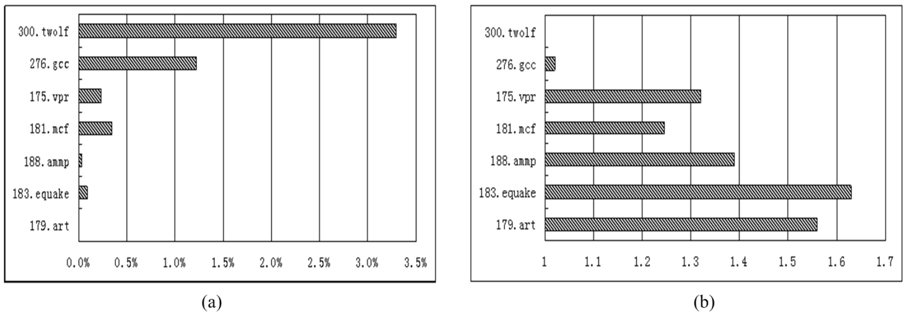


Fig. 7. (a) Restart rates of speculative threads; (b) Speedups of speculative parallel execution

Finally the speedup results are given to depict the effect of our approach on program performance in Fig. 7(b). The speedup values are based on comparing the original sequential program execution time against the time of speculative execution which includes profiling phase and optimized execution phase. For all float pointing and two integer applications, significant performance improvement has been observed. The optimized execution eliminates the negative effect of initial profiling phase. But for the gcc and twolf, this is not true. Especially for twolf, the gain is nearly zero. Through analyzing the execution of twolf, we find that there are many loops with insufficient iteration number. These bad loops increase the profiling time, but not provide enough rest optimized execution to balance the overhead. Compared with the offline method, our online profiling method can't be aware of the entire behavior of the program in advance, such as the trip count. However, the resulting performance loss can be alleviated by conservatively selecting loop candidates.

4 Conclusion

From the above results it can be concluded that the two-phase online profiling approach has the comparable ability to the offline profiling when it is taken as a guide to transaction partition optimization and identifying the loops suitable to be speculatively parallelized. This approach can guide the speculative optimization of the applications lacking appropriate training inputs, while the traditional offline method doesn't work for them. In addition, it can effectively deal with the applications with phase-changed behavior due to its continuous optimizing ability.

An interesting idea is to combine the online and offline profiling methods. A fast offline profiling provides some global but coarse information about program behavior, and using this information the online one can focus the profiling and optimization efforts on the more valuable targets. We will attempt this idea in future work. And a specializing codes tool with low overhead at runtime will also be developed to replace the current multiple versions way.

Acknowledgments. We would like to thank Chen Yang and Tin-Fook Ngai for the important advice and guidance from their experience in [6]. We also thank Feng Wang and Rui Guo for their contributions to the experiments. This research is supported by the grant from Intel Education (PO#4507176412), the National Natural Science Foundation of China (60373043 and 60633040) and the National Basic Research Program of China (2005CB32160).

References

1. Li, X.-F., Du, Z.-H., Yang, C., Lim, C.-C., Ngai, T.-F.: Speculative Parallel Threading Architecture and Compilation. In: ICPPW'05 (2005)
2. Steffan, J.G., Colohan, C.B., Zhai, A., Mowry, T.C.: A Scalable Approach to Thread-Level Speculation. ISCA 27 (2000)
3. Chen, M., Olukotun, K.: The Jrpm System for Dynamically Parallelizing Java Programs ISCA 30 (2003)
4. Hammond, L., Hubbert, B., et al.: The Stanford Hydra CMP. IEEE Micro 20(2), 71–84 (2000)
5. Li, X.-f., Yang, C., Du, Z.-H., Ngai, T.-F.: Exploiting Thread-Level Speculative Parallelism with Software Value Prediction. ACSAC05
6. Du, Z.-H., Lim, C.-C., Li, X.-F., Yang, C., Zhao, Q., Ngai, T.-F.: A Cost-Driven Compilation Framework for Speculative Parallelizing Sequential Program. In: PLDI'04 (2004)
7. Wu, Y., Breternitz, M., Quek, J., Etzion, O., Fang, J.: The Accuracy of Initial Prediction in Two-Phase Dynamic Binary Translators. In: CGO'04 (2004)
8. Chen, T., Lin, J., Dai, X., Hsu, W.C., Yew, P.C.: Data Dependence Profiling for Speculative Optimization.CC 13, 57–72 (2004)
9. Hammond, L., Wong, V., et al.: Transactional Memory Coherence and Consistency. ISCA 31 (2004)
10. Gupta, R., Mehofer, E., et al.: Profile Guided Compiler Optimizations. The Compiler Design Handbook: Optimizations & Machine Code Generation. Auerbach Publications

Entropy-Based Profile Characterization and Classification for Automatic Profile Management*

Jinpyo Kim[1], Wei-Chung Hsu[1], Pen-Chung Yew[1], Sreekumar R. Nair[2], and Robert Y. Geva[2]

[1] Department of Computer Science and Engineering, University of Minnesota, Twin-Cities, Minneapolis, MN 55455, USA
{jinpyo,hsu,yew}@cs.umn.edu
[2] Intel, Santa Clara, CA, USA
{sreekumar.r.nair,robert.geva}@intel.com

Abstract. The recent adoption of pre-JIT compilation for the JVM and .NET platforms allows the exploitation of continuous profile collection and management at user sites. To support efficient pre-JIT type of compilation, this paper proposes and studies an entropy-based profile characterization and classification method. This paper first shows that highly accurate profiles can be obtained by merging a number of profiles collected over repeated executions with relatively low sampling frequency for the SPEC CPU2000 benchmarks. It also shows that simple characterization of the profile with *information entropy* can be used to guide sampling frequency of the profiler in an autonomous fashion. On the SPECjbb2000 benchmark, our adaptive profiler obtains a very accurate profile (94.5% similar to the baseline profile) with only 8.7% of the samples that would normally be collected using a 1M instructions sampling interval. Furthermore, we show that entropy could also be used for classifying different program behaviors based on different input sets.

1 Introduction

JIT compilers have been widely used to achieve near native code performance by eliminating interpretation overhead [1,2]. However, JIT compilation time is still a significant part of execution time for large programs. In order to cope with this problem, recently released language runtime virtual machines, such as Java Virtual Machine (JVM) or Microsoft Common Language Runtime (CLR), provides Ahead-Of-Time (AOT) compiler, sometimes called pre-JIT compiler, to generate native binaries and store in a designated storage area before the execution of some frequently used applications. This approach could mitigate runtime compilation overhead [3].

A pre-JIT compiler [4] can afford more time-consuming profile-guided optimizations (PGO) compared to a JIT compiler because compilation time in a

* This research was supported in part by NSF grant no. EIA-0220021, Intel and HP.

© Springer-Verlag Berlin Heidelberg 2007

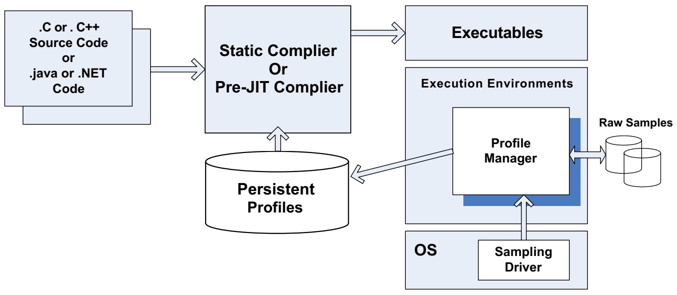


Fig. 1. Continuous Profile-Guided Optimization Model

pre-JIT compiler may not be a part of execution time. With the deployment of Pre-JIT compilers, automatic continuous profiling and re-optimization, as shown in Figure 1, becomes a viable option for the managed runtime execution environment. For example, with the introduction of a Pre-JIT compiler on the recent Microsoft CLR [3], continuous PGO framework as shown in Figure 1 has become a feasible optimization model. The Pre-JIT compiler compiles MSIL code into native machine code and stores it on the disk. The re-compilation does not occur during program execution but, instead, is invoked as an offline low priority process. The re-compilation process relies on accurate HPM (Hardware Performance Monitor)-sampled profiles that are accumulated over several executions of the application program. HPM-sampled profiles could provide more precise runtime performance events, such as cache misses and resource contentions, that allow more effective runtime or offline optimizations [5,6,7].

Due to the statistical nature of sampling, the quality of sampled profiles is greatly affected by the sampling rate. A high sampling frequency would cause more interrupts, require more memory space to store sampled data, and more disk I/O activities to keep persistent profile data. With a fixed number of runs, a challenge to the runtime profiler is how to determine the ideal sampling rate so that we can still obtain high quality profiles with minimum sampling overhead.

In this paper, we propose an "*information entropy*" to determine an adaptive sampling rate for automated profile collection and processing that can efficiently support continuous re-optimization in a pre-JIT environment. The information entropy is a good way to summarize the frequency distribution into a single number [8]. Since the sampled profile in our study is the frequency profile of collected PC addresses, the information entropy of the profile is well suitable for characterizing program behaviors we are interested in.

In practice, a program has multiple input data sets and exhibits a different program behavior for each particular input. Hence, the entropy of a profile could be different according to the input used. We show that the information entropy can be used to classify profiles with similar behaviors. This classified profiles allow the optimizer to generate specially optimized for that particular input sets.

The primary contributions of this paper are as follows:

- We show that highly accurate profiles can be obtained efficiently by merging a number of profiles collected over repeated executions with low sampling rates. We demonstrate this approach by using the SPEC2000 benchmarks.
- We also show that a simple characterization of profiles using *information entropy* can be used to automatically set the sampling rate for the next profiling run. On SPECjbb2000, our adaptive profiler obtains very accurate profile (94.5% match with the baseline profile) with only 8.7% of the samples needed using 1M-instruction sampling intervals.
- We show that the entropy of a profile could be used to classify different program behaviors according to different input sets and to generate classified profiles for targeted optimizations.

The rest of this paper is organized as follows. Section 2 describes the methodology used in evaluating the accuracy of sampled profiles. Section 3 describes entropy-based profile characterization to adaptively select the sampling rate for efficient profile collection and management. Section 4 describes entropy-based profile classification to characterize sampled profiles from multiple input sets. Section 5 provides experimental setups and results. Section 6 discusses related works. Finally, Section 7 summarizes our work.

2 Similarity of Sampled Profiles

Sampling-based profiler collects the frequency profile of sampled PC addresses instead of edge count profiles using instrumentation. Zhang et al. [9] show that an accurate edge profile can be deduced from the frequency profile of sampled PC addresses.

2.1 Similarity Metric

In order to evaluate the similarity between a sampled profile and the "complete profile", we define a "*similarity comparison metric*" between the two profiles. Since our profile is a list of frequency counts of distinct PC addresses sampled, it can be represented as an 1-dimensional vector. To compute the linear distance between two vectors, we used the Manhattan distance shown in the following equation as a similarity comparison metric.

$$S = \sum_{i=1}^{n} \frac{(2.0 - |a_i - b_i|)}{2.0} \qquad a_i, b_i : \text{relative freq. of } i^{th} \text{ distinct PC addr.}$$

The PC addresses of a_i and b_i are the same. The a_i or b_i could be zero when no matching samples are collected. If two profiles are identical, S becomes 1.

Baseline Profile to Determine Similarity of Sampled Profiles. Instead of using instrumented profiles, we use a merged profile generated from very high frequency sampling rates over multiple runs as the baseline profile. We collected sampled profiles three times using each of the six different sampling rates (one sample every 634847, 235381, 87811, 32973, 34512, 32890 instructions), and generate a baseline profile by merging the 18 profiles for each benchmark program. In every sampling interval, one PC address is collected. Each sampled profile has a list of frequency count for each distinct PC address. Hence, we could compute normalized frequency for each distinct PC address by dividing its frequency count by the total sample counts. We mask off 4 low-order bits of the PC address to approximate a basic block, i.e. use the masked address as the starting address of an approximated basic block instead of distinct PC addresses within the approximated basic block. The obtained frequency is the frequency of the approximated basic block. The obtained baseline profile is very close to the instrumentation-based complete profile. It ranges from 0.95 to 0.98 using our similarity comparison metric for SPEC CPU2000 benchmarks.

2.2 Accuracy of Persisted Profiles

As the similarity between the baseline profile and the instrumented complete profile reflects how "accurately" the baseline profile could mimic the complete profile, we use "accuracy" and "similarity" interchangeable in the rest of the paper. Intuitively, the accuracy of merged profiles improves as the number of samples increases. Figure 2 show that merged profiles are more accurate (compare to the baseline profile) than a single instance of profile, the R2R (Run-to-Run) in the figure, on *176.gcc* with *200.i* input.

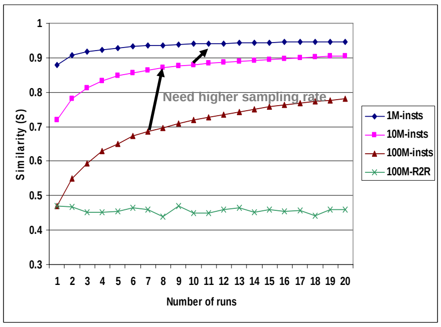


Fig. 2. Convergence of merged profiles of *gcc* with *200.i* input set

In Figure 2, sampled profiles are cumulatively merged along repeated runs on the same input set. For example, at the 10th runs, the merged profile is the summation from 1st profile to 10th profile. The y-axis shows the similarity (S) between the baseline profile and the merged profile with three different sampling rates (one sample every 1M, 10M, 100M instructions).

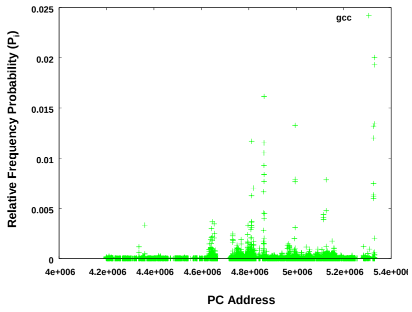
We could observe two interesting facts. First, most of the improvement in accuracy came from the first three to six runs. Second, after that the curve of improvement becomes flattened. Since we cannot afford too many runs at high sampling rates, we need to adapt sampling rates according to the program behavior. We address how to automatically reduce the sampling rate through profile characterization in the next section.

3 Entropy-Based Profile Characterization

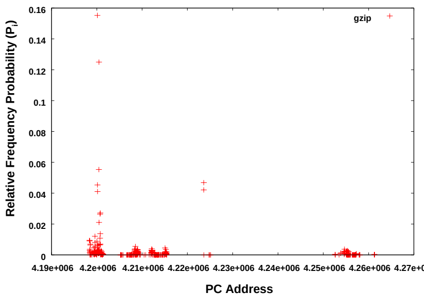
Information Entropy: A Metric for Profile Characterization. An appropriate sampling rate could be determined according to the program behavior. Since our frequency profile can be represented as a statistical distribution, where each distinct PC address has a probability that is the number of its occurrences divided by the overall number of samples, we can use the equation to quantify the shape of statistical distribution. In this paper, we propose to use *information entropy* as defined by Shannon [8] to characterize the sampled profiles (for example, "flat" or "skewed"). The information entropy is defined as follows:

$$E = \sum_{i=1}^{N} P_i \cdot log \frac{1}{P_i} \qquad P_i : \text{relative freq. prob. of } i^{th} \text{ distinct PC addr.}$$

If the program has a large footprint and a complex control flow, a large number of distinct PC addresses will be collected. On the other hand, if the program has a small number of hot spots (or hot loops), sampled profile will have a small number of distinct PC addresses. It leads to a low entropy number. This property could be used in determining an appropriate sampling rate for the next run. The two example programs shown in Figure 3 clearly show that entropy distinguish "flat" profiles from "skewed" profiles. The *gcc* in Figure 3(a) shows a relatively "flat" distribution and a high entropy number (E=10.12). In contrast, the *gzip* in Figure 3(b) shows a "skewed" distribution and a low entropy number (E=5.67).



(a) gcc (E=10.12)



(b) gzip (E=5.67)

Fig. 3. Relative frequency distribution of PC address samples (*gcc, gzip*)

Entropy-Based Adaptive Profiler. The application of entropy heuristics in the adaptive profiler is as follows:

1. When an application is loaded and ready to run, check if there is already a profile file for this application. If not, i.e. this is the first time the application executes, start with a low sampling rate.
2. After the program terminates, compute the information entropy of the profile. Categorize the profile based on one of the three ranges of entropy values. Our data shows that the following three ranges are sufficient: low [0-5], medium [5-8], and high [8 or higher] entropy.
3. When an application is loaded, if the entropy is known, set the sampling rate according to the entropy: a high entropy uses a high sampling rate, a medium entropy uses a medium sampling rate and a low entropy uses a low sampling rate.

4 Entropy-Based Profile Classification

The entropy of their profiles will also change along with the changed input sets. For multiple inputs, this section describes how entropy can be used to classify different profiles.

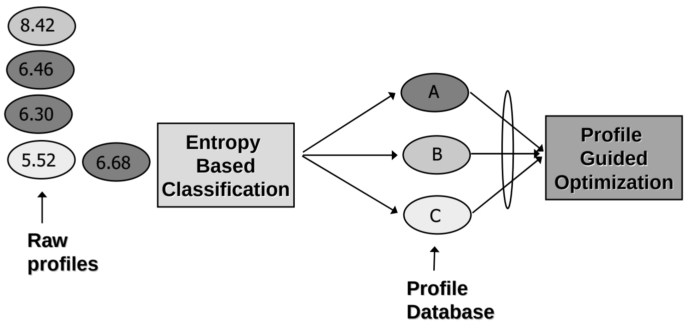


Fig. 4. Entropy-based profile classification

Figure 4 shows the workflow of entropy-based profile classification. In our profile classification framework, we use k-mean clustering technique to identify similar profiles. If the maximum number of clusters sets to 3 ($k = 3$), incoming profiles will be classified and merged into three persistent profiles (A, B, C). In Figure 4, the three profiles with their entropy in a similar range ($E = 6.46, 6.30, 6.68$) are classified and merged into a single persistent profile A. One profile with entropy ($E = 8.42$) is classified and merged into persistent profile B. Another one with entropy ($E = 5.52$) is classified and merged into persistent profile C.

When the recompilation for PGO is invoked, the controller determines whether the classified profiles are merged into one profile or one particular profile is selected for PGO. If the similarity (S) of classified profiles is very low, it means that the profiles come from disjoint code regions. In this case, it is better to combine the profiles. *vpr* is such a case. The profile from *place* and the profile from *route* are disjoint from each other. Otherwise, we have to choose one profile that is merged from majority of runs.

5 Experiments

5.1 Experimental Setup

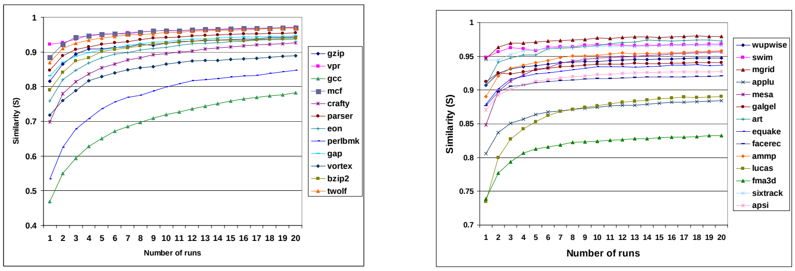
For our experiments, we collected the sampled profiles with Intel SEP(Sampling Enabling Products) tool ver. 1.5.252, running on 3.0 GHz Pentium 4 with Windows XP Operating System. The SPEC CPU2000 INT and FP benchmarks are used to evaluate the convergence of merged profiles and the effectiveness of using entropy to characterize different sampled profiles. The SPEC CPU2000 benchmarks are compiled with Intel icc compiler ver. 8.0 with O3 optimization level. The SpecJBB 1.01 benchmark is used to show how the entropy could be effectively used in an adaptive profile manager.

In the SPEC CPU2000 INT benchmarks, *vpr, vortex, gzip, gcc, perlbmk* and *bzip2* have multiple input sets. We used these six benchmarks for our experiments. These benchmarks are compiled with the Intel icc compiler ver. 8.0 with O3 optimization level, and measured on 1 GHz Itanium-2 processor machine. The profile feedback uses Intel icc profile guided optimization. In our experiments, number of maximum clusters is set to 3 ($MaxK = 3$).

5.2 Experimental Results

Accuracy of Merged Profiles. Figure 5 shows that persisted profiles converge to the baseline profile at different convergence rates based on their program behavior using SPEC CPU2000 (INT, FP) benchmark and a sampling rate of one sample every 100M instructions. In SPEC CPU2000 INT benchmarks shown in Figure 5(a), most of benchmarks converge quickly above a similarity metric of 0.9 (i.e. more than 90% similar to the baseline profile) after the initial five runs except for five benchmarks (*gcc, vortex, perlbmk, crafty, eon*). Since those four benchmarks have relatively complex control flows and large instruction footprints, they need a higher sampling rate to achieve a targeted accuracy with only a limited number of runs. In the SPEC CPU2000 FP benchmarks shown in Figure 5(b), most benchmarks converge quickly above 0.9 similarity to the baseline after initial five runs except three benchmarks (*lucas, applu, fma3d*).

Entropy-Based Profile Characterization. Table 1 shows the entropy of SPEC CPU2000 INT benchmarks. Interestingly, we can observe that the entropy of those benchmarks are clustered in three ranges. Two programs (*vpr, mcf*) show a low entropy ($0 \leq E < 5$). Four programs (*gcc, crafty, perlbmk, vortex*) show a high entropy ($E \geq 8$). The rest of programs have a medium entropy ($5 \leq E < 8$). The four programs that show high entropy exactly match the



(a) INT (b) FP

Fig. 5. Convergence of merged profiles of SPEC CPU2000 benchmarks

Table 1. Entropy of SPEC CPU2000 INT benchmarks

benchmark	entropy	benchmark	entropy	benchmark	entropy	benchmark	entropy
gzip	5.67	vpr	4.87	gcc	10.12	mcf	4.45
crafty	9.52	parser	7.79	eon	7.95	perlbmk	8.46
gap	7.62	vortex	8.02	bzip2	7.50	twolf	7.24

programs, shown in Figure 5(a), that need higher sampling rates to achieve a targeted accuracy.

Table 2 shows the entropy of the SPEC CPU2000 FP benchmarks. Two programs (*swim, art*) show a low entropy ($0 \leq E < 5$). Three programs (*lucas, applu, fma3d*) show a high entropy ($E \geq 8$). Three programs that show high entropy also exactly match the programs, shown in Figure 5(b), that need higher sampling rate. The results strongly suggest that entropy is a good metric to select the sampling rate for SPEC CPU2000 benchmarks (INT, FP).

Table 2. Entropy of SPEC CPU2000 FP benchmarks

benchmark	entropy	benchmark	entropy	benchmark	entropy	benchmark	entropy
wupwise	6.43	swim	4.86	mgrid	5.09	applu	8.26
mesa	7.85	galgel	5.86	art	4.01	equake	5.98
facerec	6.44	ammp	6.95	lucas	9.26	fma3d	9.16
sixtrack	5.19	apsi	7.91				

We could start sampling with high frequency for all programs to obtain more accurate profiles in general. However, it would require unnecessary high overhead. Based on our entropy based characterization and observation on convergence of merged profiles, only seven programs among 26 SPEC CPU2000 programs need a sampling rate higher than one sample per 100M instructions. Hence, it is more cost-effective to start with low sampling rate and adjust the sampling rate according to the profile entropy collected at runtime.

Adaptive Profiler. Figure 6 shows the results of using entropy in an adaptive profiler for the SPECjbb ver. 1.01 benchmarks written for Microsoft .NET platform. After the first run of the program, the profiler increases the sampling rate from one sample per 100 M instruction to one sample per 10M instructions according to the entropy measured. In practice, we may not have the baseline profile to compute the similarity metric (S). We could use the delta (Δ) of similarity (S) between current cumulative profile and previous cumulative profile. If the ΔS is small enough (for example $\Delta S = 0.005$), we consider that convergence curve of merged profile has been flattened. Depending on the number of runs, the profiler can stop or continue to collect profiles. In Figure 6, the ΔS is less than a given threshold ($\Delta S < 0.005$) at the 7^{th} run. Since we expect 6 to 8 runs in this experiments, the profiler decides to stop profile collection.

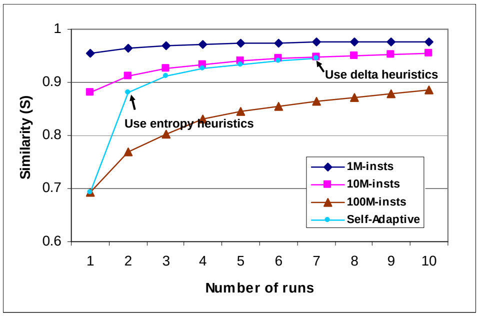


Fig. 6. Accuracy of entropy-based adaptive profiler on SPECJBB ver. 1.01

When we compare the profiles generated from our adaptive profiler with those from a sampling rate of one sample per 1M instructions, our profile is quite accurate (S=0.945) (94.5% similar to the baseline) with only 8.7% of samples taken. Our profile is only 3% less accurate compared to the profile generated from the sampling rate of one sample per 1M instructions. Since an edge profile could be deduced from this frequency profile as explained earlier, 3% difference in accuracy will not lead to any significant difference in the accuracy of the deduced edge profile.

Entropy-Based Profile Classification. We found that there are three types of program behavior. In the type I, the program behavior does not change much with different input sets. Hence, the entropy of their profiles from different inputs is pretty similar to each other. *vortex* program is like that. Their sampled profiles are classified and merged into one baseline profile. This is the most simplest case.

Table 3 shows performance improvement from PGO on *vortex* with multiple input sets. Each column of the table uses different input set. For convenience, it is numbered according to input sets, for example the one is for lendian1 input. Each row presents the performance improvement from the binary generated using feedback profiles. The baseline binary to compute performance improvement

Table 3. Performance improvement (%) from PGO on *vortex* with multiple input sets

	1:endian1	2:endian2	3:endian3	average
feedback:(1)	27.64	30.84	27.83	28.77
feedback:(2)	28.03	31.26	26.92	28.74
feedback:(3)	28.02	30.02	31.26	29.77
feedback:(self)	27.64	31.26	31.26	30.06
feedback:(classified)	27.30	31.39	26.25	28.31
Entropy	7.80	8.19	7.77	

is the binary generated without using PGO. For example, feedback(1) is the binary generated using the profile generated from lendian1 input. feedback(self) is the binary that is generated from the same input with which the profile generated. The feedback(self) is used to show the full potential of PGO. In *vortex*, feedback(classified) uses one profile merged from all profiles.

In the type II, program behavior changes significantly due to the change of input data set. Hence, the sampled profiles from each different input are dissimilar. *vpr* is like that. Since the entropy of two sampled profiles are in different ranges, they are classified into two different profiles. Since the similarity (S) of the profiles is very low ($S < 0.4$), the profile manager combines them into one profile when used for PGO. Combining disjoint profiles is generally beneficial since it would increase the code coverage.

Table 4 shows the performance improvement from PGO on *vpr* with different input sets. The feedback(1) experiences 3.0% slowdown compared to the baseline binary when input set 2 is used. The feedback(2) also loses 6.76% performance when input set 1 is used. It shows that PGO could degrade performance if the profile used in feedback is not generated from a representative input set. The feedback(classified) uses a merged profile from the two sampled profiles. Interestingly, it performs 2.26% better than the feedback(self) binary. It might be because merged profile provides increased code coverage that gives slightly better analysis results for compiler optimizations. This may be due to some heuristics used in compiler optimizations that are sensitive to path frequency distribution.

In the type III, their profiles could be classified into several groups of similar profiles. *gzip, perlbmk, gcc* and *bzip2* are in this camp.

Table 4. Performance improvement (%) from PGO on *vpr* with multiple input sets

	1:place	2:route	average
feedback:(1)	4.72	-3.00	0.86
feedback:(2)	-6.76	8.59	0.92
feedback:(self)	4.72	8.59	6.66
feedback:(classified)	8.96	8.88	8.92
Entropy	6.83	4.82	

Table 5. Performance improvement (%) from PGO on *gzip* with multiple input sets

	1:source	2:log	3:graphic	4:random	5:program	average
feedback:(1)	4.52	4.17	7.87	5.75	4.04	5.27
feedback:(2)	5.05	4.77	9.73	11.24	5.01	7.16
feedback:(3)	-6.97	-8.02	6.29	10.35	-8.89	-1.45
feedback:(4)	-0.28	-4.86	7.24	14.49	1.65	3.65
feedback:(5)	5.62	4.68	13.40	12.56	5.06	8.26
feedback:(self)	4.52	4.77	6.29	14.49	5.06	7.03
feedback:(classified)	6.38	4.95	6.29	12.48	5.06	7.03
feedback:(1,2,4)	6.38	4.95	12.32	12.48	6.38	8.50
Entropy	5.71	5.87	6.49	5.92	4.98	

Table 5 shows the performance improvement from PGO on *gzip* with different input sets. Three profiles (1:source, 2:log, 4: random) are merged into one profile. The profile (3:graphic) and the profile (5:program) are classified into separate profiles. The profiles from three text inputs are classified into the same group. Table 5 indicates that entropy-based classification works quite well. The feedback(1,2,4) performs 1.47% better than that from feedback(self).

From the above results, we could see that entropy is a good metric to classify sampled profiles for PGO. The binaries generated from classified profiles always perform similar to or better than that from feedback(self) binaries. It should be noted that in our experiments, the feedback(classified) binaries never caused any slowdown compared to the performance of the binaries generated without PGO for any input sets. In contrast, for example, feedback(3) in *gzip* shows slowdown in performance compared to the performance of the binary generated without PGO for three inputs (*1:source, 2:log, 5:program*).

6 Related Work

Savari and Young [10] introduced an approach to analyze, compare and combine profiles that use information theories. In our work, we used information entropy to adaptively select sampling rate, characterize profiles and classify them instead of combining them.

Kistler and Franz [11] presented a frequency profile (edges and paths) as a vector so as to compare the similarity among profiles. Their proposed similarity metric is based on geometric angle and vector distance between two vectors. Their goal was to determine when a program has changed significantly enough to trigger re-optimization. In our work, we use a Manhattan distance to compute a similarity metric between two profiles. Our metric is linear in contrast to Kistler's similarity comparison function.

Sun et al. [12] show that an information entropy based on performance events, such as L2 misses, could be a good metric to track changes in program phase behavior. Our paper uses an information entropy based on frequency profiles to determine an appropriate sampling rate for efficient adaptive profiling.

7 Summary

This paper shows that highly accurate profiles can be obtained by merging a number of profiles collected over repeated executions with relatively low sampling frequency. It also show that simple characterization of the profile with *information entropy* can effectively guide the sampling rate for a profiler. Using SPECjbb2000 benchmark, our adaptive profiler obtains very accurate profile (94.5% similar to the baseline profile) with only 8.7% of samples in a sampling rate of one sample per 1M instructions. Furthermore, we show that information entropy could be used to classify different profiles obtained from different input sets. The profile entropy-based approach provides a good foundation for continuous profiling management and effective PGO in a pre-JIT compilation environment.

References

1. Burke, M., Choi, J.D., Fink, S., Grove, D., Hind, M., Sarkar, V., Serrano, M., Sreedhar, V., Srinivasan, H., Whaley, J.: The jalapeno dynamic optimizing compiler for java. In: Proc. ACM 1999 Java Grande Conference, pp. 129–141. ACM Press, New York (1999)
2. Arnold, M., Fink, S., Grove, D., Hind, M., Sweeney, P.F.: Adaptive optimization in the jalapeno jvm. In: 15th Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 47–65 (2000)
3. Bosworth, G.: Prejit in the clr. In: 2nd Workshop on Managed Runtime Environments (MRE‘04) (2004)
4. Vaswani, K., Srikant, Y.: Dynamic recompilation and profile-guided optimisations for a.net jit compiler. IEE Proc.-Softw. 150(5), 296–302 (2003)
5. Luk, C., et al.: Ispike: A post-link optimizer for intel itanium 2 architecture. In: Proceedings of 2nd International Symposium on Code Generation and Optimization (CGO)
6. Choi, Y., Knies, A., Vedaraman, G., Williamson, J.: Design and experience: Using the intel itanium 2 processor performance monitoring unit to implement feedback optimization. In: EPIC2 Workshop (2002)
7. Lu, J., Chen, H., Fu, R., Hsu, W.C., Othmer, B., Yew, P.C.: The performance of runtime data cache prefetching in a dynamic optimization system. In: Proceedings of the 36th Annual International Symposium on Microarchitecture (December 2003)
8. Cover, T., Thomas, J.: Elements of Information Theory. John Wiley and Sons, Chichester (1991)
9. Zhang, X., Wang, Z., Gloy, N., Chen, J., Smith, M.: System support for automatic profiling and optimization. In: Proc. 16th ACM Symp. Operating Systems Principles, pp. 15–26 (1997)
10. Savari, S., Young, C.: Comparing and combining profiles. Journal of Instruction Level Parallelism (2004)
11. Kistler, T., Franz, M.: Computing the similarity of profiling data. In: Proc. Workshop on Feedback-Directed Optimization (1998)
12. Sun, M., Daly, J., Wang, H., Shen, J.: Entropy-based characterization of program phase behaviors. In: the 7th Workshop on Computer Architecture Evaluation using Commercial Workloads (February 2004)

Laplace Transformation on the FT64 Stream Processor[⋆]

Yu Deng, Xuejun Yang, Xiaobo Yan, and Kun Zeng

PDL, School of Computer, National University of Defense Technology,
Changsha, Hunan province, China
yudeng@nudt.edu.cn

Abstract. The stream architecture is one of the emerging architectures that address the memory-wall problem of modern processors. While it is successful in the domain of multimedia, its efficiency to scientific applications is increasingly concerned. This paper implements a stream program for Laplace transformation and evaluates its performance on the FT64 stream processor, which is the first implementation of a 64-bit stream processor for scientific computing. The stream program is optimized against the memory hierarchy of FT64 to minimize the expensive off-chip memory transfers. The preliminary results show that FT64 is more efficient than the traditional cache-based processor (Itanium2) for Laplace transformation.

1 Introduction

The increasing gap between the processor and memory is a well-known problem in the modern processor design. The stream processors [1] [2] [3] [4] [5] are presented to address the processor-memory gap through streaming technology. The stream processors have demonstrated significant performance advantages in the domains such as signal processing [6], multimedia and graphics [7]. Yet, it has not been sufficiently validated whether stream processor is efficient for scientific computing.

FT64 [5] is the first implementation of a 64-bit stream processor for scientific computing. FT64's instruction set architecture is reduced and optimized for scientific computing. FT64's processing unit consists of four ALU clusters, and each of them contains four floating-point multiply-accumulate (FMAC) units. FT64's peak performance reaches 16GFLOPS.

This paper implements a stream program for Laplace transformation and evaluates its performance on the FT64 stream processor. The stream program for Laplace transformation is constructed according to the memory hierarchies of FT64, so that the data reuse indicated by input dependence can be exploited in the on-chip memory hierarchies to minimize the expensive off-chip memory transfers. The evaluation results show that the data locality of the stream program for Laplace transformation can be well exploited by FT64 and the average

[⋆] This work is supported by NSFC (60621003 and 60633050).

© Springer-Verlag Berlin Heidelberg 2007

speedup over the traditional cache-based processor (Itanium2) is more than 2. Besides, it demonstrates a good scalability on FT64.

The rest of this paper is organized as follows. Section 2 overviews the architecture of FT64. Section 3 discusses the details of the implementation of the stream program for Laplace transformation. In Section 4, we present the preliminary experimental results over FT64 and Itanium2. Section 5 reviews the related work. In Section 6, we conclude the paper and discuss some future work.

2 The FT64 Architecture

FT64 employs stream programming model [8]. In this model, an application is represented by a set of computation kernels which consume and produce data streams. Each data stream is a sequence of data records of the same type, which can be classified into two kinds: basic streams and derived streams. Basic stream defines a new sequence of data records while derived stream refers to all or part of an existing basic stream. Each kernel is a program which performs the same set of operation on each input stream element and produces one or more output streams. Stream applications consist of stream-level programs and kernel-level programs. A stream-level program specifies the order in which kernels execute and organize data into sequential streams that are passed from one kernel to the next. A kernel-level program is structured as a loop that processes element(s) from each input stream and generates output(s) for each output stream.

FT64 is mainly composed of a stream controller (SC), a stream register file (SRF), a micro controller (UC), four ALU clusters, a stream memory controller (SMC), a DDR memory controller (DDRMC), a host interface (HI) and a network interface (NI), as illustrated in Figure 1.

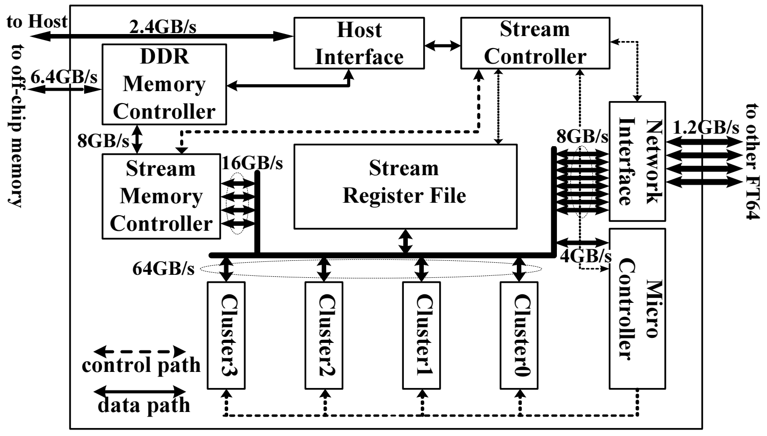


Fig. 1. Block diagram of FT64

FT64 runs as a coprocessor to a host through the HI. The host executes stream-level programs, loads the streams and kernel-level programs to FT64's off-chip memory, and sends stream-level instructions to the SC. The SC executes stream-level instructions. First, it loads a kernel-level program from the off-chip memory to UC's instruction memory through the SRF. Then, the SC loads streams from the off-chip memory to the SRF. When the kernel-level program and streams are ready, the SC will start the execution of the kernel-level program in the UC's instruction memory, and the UC controls the four clusters to process data in a SIMD fashion. When the computation is finished, the SC will transfer the output stream stored in SRF to the off-chip memory or to other FT64's SRF through the NI directly.

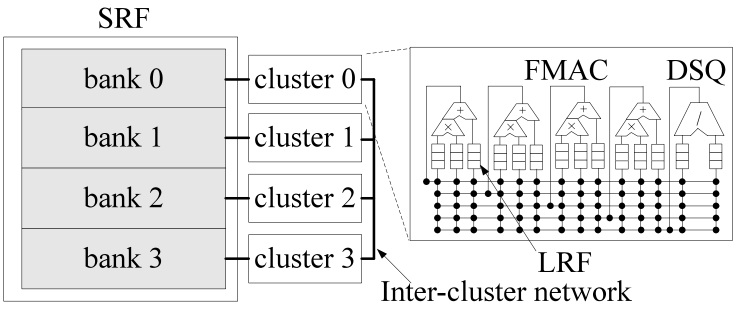


Fig. 2. The microarchitecture of ALU clusters

The microarchitecture of the ALU clusters is shown in Figure 2. Each cluster is composed of four floating-point multiply-accumulate units (FMACs), a divide/square root unit (DSQ) and their local register files (LRFs). The four clusters can communicate with each other through an inter-cluster network. The SRF is a software-controlled memory hierarchy which is used to buffer the input/output and intermediate streams during execution. The SRF is divided into four banks which correspond to the four clusters respectively. Each cluster can only access its own SRF bank.

3 A Stream Program Implementation for Laplace Transformation

Laplace transformation is very common in scientific applications, such as partial differential equation solver and digital signal processing. The core of Laplace transformation is a simple two-dimensional central differencing scheme. Let s be the source matrix for the Laplace transformation to be played on, d be the resulting matrix, the transformation can be formulated as a two-loop-nests in Figure 3. Each element in d is the result of subtracting 4 times of the corresponding element in s from the sum of the four elements around it. Five computations

```
do j = 2, n-1
   do i = 2, n-1
      d(i, j) = s(i-1, j) + s(i, j-1) + s(i+1, j) + s(i, j+1) - 4*s(i, j)
   enddo
enddo
```

Fig. 3. The core of Laplace transformation

(three adds, one subtract and one multiply) are needed for one result and the computation-to-memory ratio is $O(1)$.

The inner and the outer loop nest of Laplace transformation both carry input dependences. It is essential to exploit the data reuse indicated by the loop-carried input dependences in the on-chip memory hierarchies for good performance. In the traditional distributed parallel computers, Laplace transformation is done by two steps. First, each column of s is distributed to each processor, where the sum of the two rows in stride of two is calculated, resulting an intermediate matrix. Second, each processor communicates with its neighbors to calculate the sum of the two columns in stride of two, and yields each column of the resulting matrix with the intermediate matrix.

The most important thing to implement Laplace transformation in a stream processor is the data arrangement, i.e. how data streams are organized from the original arrays. The simplest way is to take each array reference in the loop nests in Figure 3 as one stream. But this will incur much overlap between streams and increase the off-chip memory transfers.

Figure 4 gives the data arrangement layout in our implementation. As shown in Figure 4(a), each column of the source matrix is taken as one data stream. When one stream is loaded into the SRF, it is distributed across the four banks. For example, the first element of $s1$ (corresponds to the first column of s), s_{11}, is allocated to bank0, the second element s_{21} is allocated to bank1, and so on. The whole stream program for Laplace transformation consists of a series of kernels, each of which takes three streams as input streams to produce one

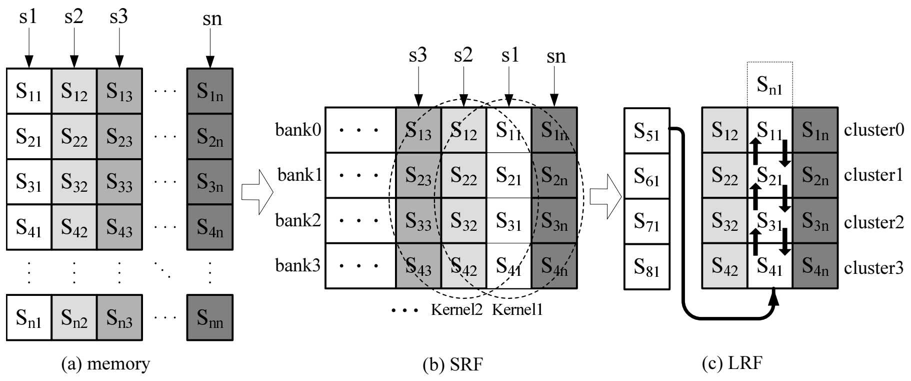


Fig. 4. The data layout of the stream program for Laplace transformation

output stream, as shown in Figure 4(b). The first kernel (kernel 1 in Figure 4(b)) needs to load three streams to the SRF and the successive kernels only need to load one stream to the SRF, for the other two streams have been loaded to the SRF by their predecessors.

The kernel is structured as a loop. During each loop iteration, each cluster reads one element from each input stream and communicates with its neighbors to obtain the other two elements, as shown in Figure 4(c). The black arrows in Figure 4(c) indicate the direction of communication. For example, cluster1 reads s_{2n}, s_{21} and s_{22} from sn, $s1$, $s2$ respectively to its LRF, then obtain s_{11} and s_{31} from cluster0 and cluster2 respectively. Actually every cluster needs to read two elements from the second stream, for cluster3 needs the next round of element from cluster0.

The borders are considered separately. The elements in the last row of s which are needed to calculate the first row of d are passed to the kernels as scalar parameters, so do the elements in the first row of s.

In this implementation, the data reuse indicted by the outer loop-carried input dependence is exploited in the SRF and the data reuse indicated by the inner loop-carried input dependence is exploited in the LRF through inter-cluster communication.

4 Performance Evaluation

To evaluate the performance of the stream program for Laplace transformation, we perform experiments on the FT64 development board. As shown in Figure 5, the FT64 development board contains one host processor and eight FT64 stream processors. In this preliminary work, only one FT64 is used.

To compare the performance on FT64 with that on the traditional cache-based processor, we execute the Laplace benchmark from NCSABench [9] on an Itanium2 based server. The Laplace benchmark is compiled by Intel's compiler *ifort* (version 9.0) with the optimization option of $-O3$.

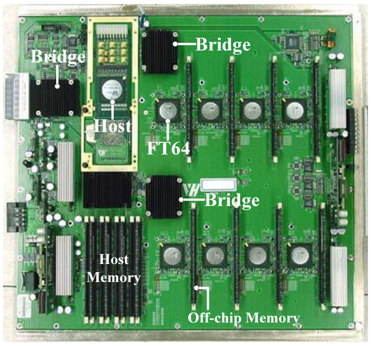


Fig. 5. The FT64 development board

Table 1. Architectural overview of test platforms

	FT64	Itanium2
Clock	500MHz	1.6GHz
#Core	4	1
Register File	19KB	2.4KB
Local Store	256KB	-
Caches	-	L1: 16KB; L2: 256KB; L3: 6MB
Off-chip BW	6.4GB/s	6.4GB/s

Table 1 gives the architectural parameters of the two test platforms.

The execution time is obtained by inserting the clock-fetch assembly instructions. If the data size of the program is small, we eliminate the extra overheads (such as system calls) by means of executing it multiple times and calculating the average time consumption. As I/O overheads are hidden in our experiments, the CPU time is nearly equal to the wall-clock time.

The number of operations per second (GOPS) can be used to evaluate the performance of the stream processor. There are two kinds of operations: computational and non-computational operations. The computational operations include adds, subtracts and multiplies, etc. The non-computational operations include communications and conditional selections, etc. Although the non-computational operations do not contribute to the result directly and inflate the total number of operations performed, they are necessary to implement the data reuse and reduce the overall execution time.

Figure 6 illustrates the number of computational and non-computational GOPS as the data size ($N * N$) increases. The GOPS increases before the data size reaches $256 * 256$, and changes little after that. This is because when the

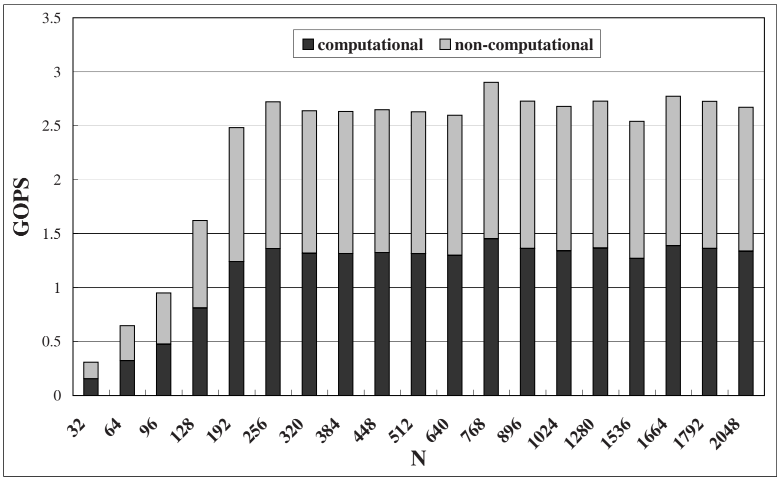


Fig. 6. GOPS of computational and non-computational operations

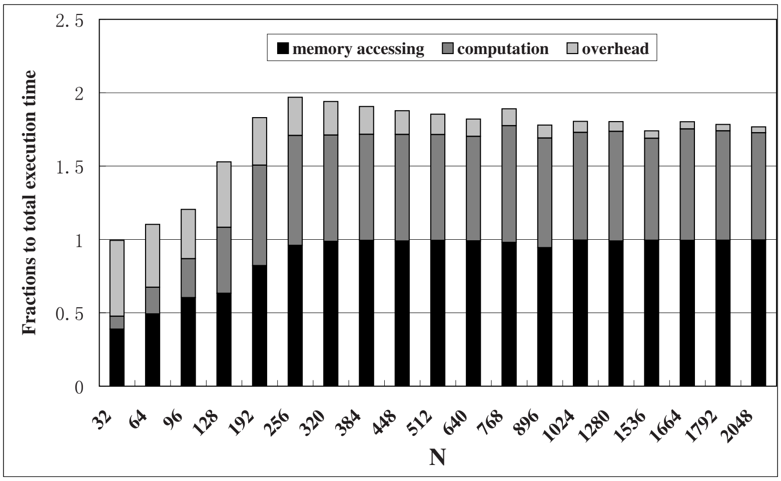


Fig. 7. Fractions of computation time, memory access time and overhead to the total execution time

data size is small, the memory access and overhead take a great fraction of the execution time. The computational operations take only half of the total GOPS. The maximal computational GOPS is 1.45, approximately 9% of the peak performance of the FT64 processor.

The execution time of a stream program on the stream processor is determined by the following factors: computation time, memory access time, overheads and the overlapping between them. The overheads include the cost to maintain the SRF allocating information, read/write the control registers and initialize memory load/store. Figure 7 gives the fractions of computation time, memory access

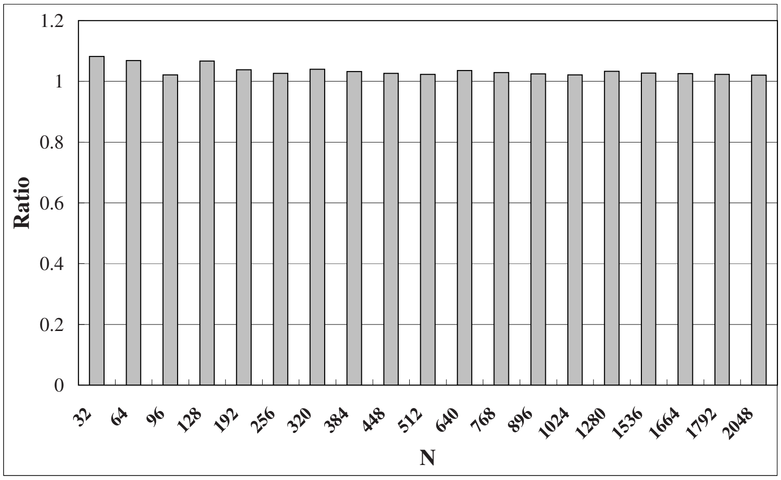


Fig. 8. The ratios of actual to the theoretically minimal memory transfers

time and overheads in the total execution time as the data size increases. The totals exceed 100% due to operation overlapping. As can be seen in Figure 7, the overheads occupy a considerable percentage of the total execution time when the data size is below 256∗256. As the data size increases, the overheads decrease and are totally hidden in the memory access time. The overlapping degree between computation and memory access is low when the data size is below $256 * 256$. This is because the execution time of a kernel is too short to hide the overhead to start a memory operation when the data size is below $256 * 256$. After that, the memory access time occupies nearly 100% of the total execution time and thus the others are hidden in it.

Figure 8 illustrates the ratios of the actual memory transfers to the theoretically minimal ones of varying the data size. The actual memory transfers are very close to the theoretical minimum (only $2\% - 8\%$ more), which means that our stream program for Laplace transformation has eliminated most extra memory transfers.

As the stream-level program for Laplace transformation is structured as a loop of kernels, the execution time will be affected by the loop unrolling times. As shown in Figure 9, the execution time at the data size of $512 * 512$ decreases gradually as the loop unrolling times increases. This is because more data reuse can be captured in one loop iteration as the loop unrolling times increases. To achieve the best performance, complete loop unrolling is used in experiments.

LRF (or SRF) -to-memory throughput ratio is the ratio of the data throughput in the LRF (or SRF) to that in the off-chip memory. Figure 10 shows the LRF/SRF-to-memory throughput ratios of varying data size. These metrics represent the LRF and SRF locality exploited by the stream processors. It can be observed that the LRF- and SRF-to-memory throughput ratios become stable when the data size is larger than $192 * 192$. The average LRF- and SRF-to-

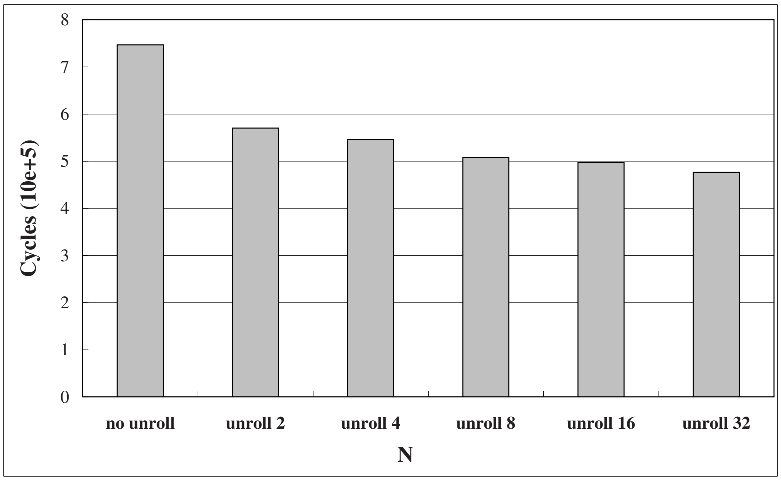


Fig. 9. Effect of loop unrolling at $512 * 512$

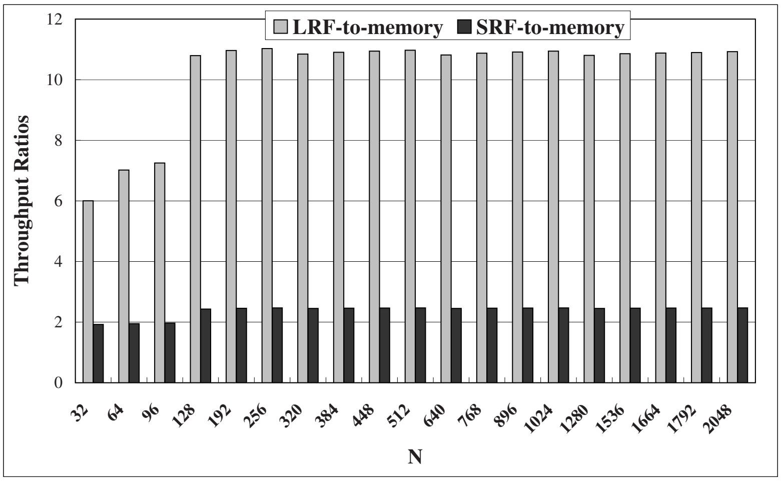


Fig. 10. LRF- and SRF-to-memory throughput ratios

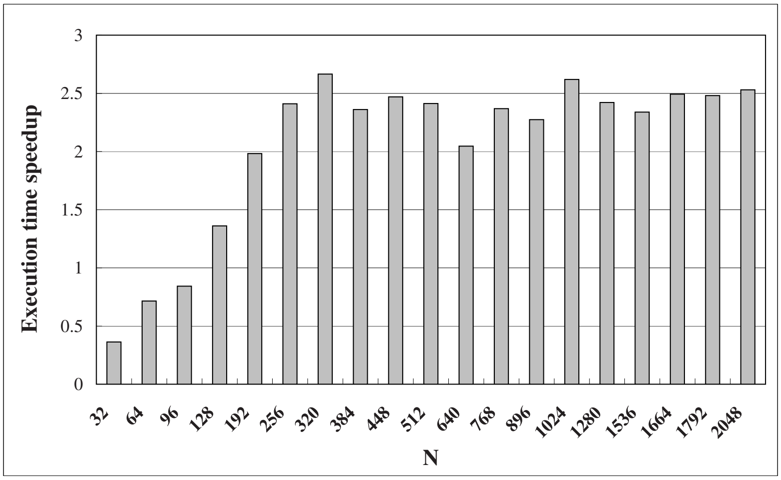


Fig. 11. Speedup of FT64 over Itanium2

memory throughput ratio is about 11 : 2.5 : 1, which is far from the ratio provided by the hardware (85 : 10 : 1). This implies that the most pressure of the memory hierarchy still lay on the off-chip memory.

Figure 11 gives the respective execution time speedup attained by FT64 over Itanium2 processor of varying the data size. When the data size is below 128∗128, the performance of FT64 is poorer than Itanium2, due to the overheads incurred by short streams. After that, the speedup increases as the data size increases and remains 2 to 2.6 after 192∗192. This demonstrates the good scalability of Laplace transformation on the FT64 processor.

5 Related Work

Though media applications are becoming the dominate consumer of stream processors [3] [8] [7] [10], there is an important effort to research whether scientific applications are suited for stream processors. Some linear algebra equation solvers, dense matrix applications and some mathematic algorithms such as transitive closure have been implemented on Imagine and Merrimac [2] [11] [12] [13]. The previous works on applying the stream processors to scientific applications have some shortages. Imagine is 32-bit stream processor which is designed for media applications and its support to scientific applications is insufficient. Merrimac has not yet been taped out and only simulator can be used. FT64 is the first implementation of a 64-bit stream processor for scientific computing and the work in this paper is the original evaluation on the real chip.

6 Conclusion and Future Work

In this paper, we implement the stream program for Laplace transformation and evaluate its performance on the FT64 stream processor. The stream program for Laplace transformation is constructed according to the memory hierarchies of FT64, so that the data reuse indicated by input dependence can be exploited in the on-chip memory hierarchies of FT64 to minimize the expensive off-chip memory transfers. The evaluation results show that FT64 is more efficient than the traditional cache-based processor (Itanium2) for Laplace transformation.

There are a number of interesting but challenging research directions in this area. We are considering extending the stream program for Laplace transformation to a parallel computer system based on multiple FT64s. More scientific applications are going to be evaluated on FT64. Besides, we will take account of developing an automatic compile tools to generate efficient stream programs from the existing scientific programs.

References

1. Kapasi, U., Dally, W.J., Rixner, S., Owens, J.D., Khailany, B.: The Imagine Stream Processor. In: ICCD'02: Proceedings of 20th IEEE International Conference on Computer Design, pp. 282–288. IEEE Computer Society Press, Los Alamitos (2002)
2. Dally, W.J., Hanrahan, P., Erez, M., Knight, T.J., et al.: Merrimac: Supercomputing with Streams. In: SC'03: Proceedings of Supercomputing Conference 2003, pp. 35–42 (2003)
3. Rixner, S.: Stream Processor Architecture. Kluwer Academic Publishers Group, Dordrecht (2002)
4. Taylor, M., Kim, J., Miller, J., Wentzlaff, D., et al.: The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs. IEEE Micro 22(2), 25–35 (2002)

5. Yang, X., Yan, X., Xing, Z., Deng, Y., Jiang, J., Zhang, Y.: A 64-bit Stream Processor Architecture for Scientific Applications. In: ISCA'07: Proceedings of the 34st Annual International Symposium on Computer Architecture (2007)
6. Gordon, M.I., Thies, W., Karczmarek, M., Lin, J., et al.: A Stream Compiler for Communication-exposed Architectures. In: ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems, pp. 291–303. ACM Press, New York (2002)
7. Owens, J., Kapasi, U., Mattson, P., Towles, B., Serebrin, B., Rixner, S., Dally, W.: Media Processing Applications on the Imagine Stream Processor. In: ICCD'02: Proceedings of 20th IEEE International Conference on Computer Design, pp. 295–302. IEEE Computer Society Press, Los Alamitos (2002)
8. Mattson, P.: A Programming System for the Imagine Media Processor. PhD thesis, Stanford University (2002)
9. NCSABench, `http://www.ncsa.uiuc.edu/UserInfo/Perf/NCSAbench/`
10. Ahn, J.H., Dally, W.J., Khailany, B., Kapasi, U.J., Das, A.: Evaluating the Imagine Stream Architecture. In: ISCA'04: Proceedings of the 31st Annual International Symposium on Computer Architecture, pp. 14–25 (2004)
11. Griem, G., Oliker, L.: Transitive Closure on the Imagine Stream Processor. In: the 5th Workshop on Media and Streaming Processors (2003)
12. Erez, M., Ahn, J.H., Garg, A., Dally, W.J., Darve, E.: Analysis and Performance Results of a Molecular Modeling Application on Merrimac. In: SC'04: Proceedings of Supercomputing Conference 2004, pp. 263–272 (2004)
13. Fatica, M., Jameson, A., Aloso, J.J.: StreamFlo: an Euler Solver for Streaming Architectures. In: 42nd AIAA conference, Reno, Nevada, USA (2004)

Towards Data Tiling for Whole Programs in Scratchpad Memory Allocation

Lian Li[1,2], Hui Wu[3], Hui Feng[1], and Jingling Xue[1,2]

[1] Programming Languages and Compilers Group
[2] National ICT Australia
[3] School of Computer Science and Engineering, University of New South Wales, Sydney, NSW 2052, Australia

Abstract. Data tiling is an array layout transformation technique that partitions an array into smaller subarray blocks. It was originally proposed to improve the cache performance of regular loops. Recently, researchers have applied this technique to scratchpad memory (SPM) allocation. Arrays whose sizes exceed a given SPM size can be tiled or divided into smaller subarray blocks or tiles and the program performance can be significantly improved by placing the smaller subarray tiles in SPM. Existing data tiling techniques are applicable to regularly-accessed arrays in individual loop nests. In embedded applications, arrays are often accessed in multiple loop nests via possibly aliased pointers. Tiling arrays in a loop nest alone will often affect the tiling and allocation decisions for arrays accessed in other loop nests. Moreover, tiling arrays accessed via aliased pointers is difficult since their access patterns are unknown at compile time. This paper presents a new data tiling approach to address these practical issues. We perform alias profiling to detect the most likely memory access patterns and use an ILP solver to select the best tiling schemes for all loop nests in the program as a whole. We have integrated data tiling in an existing SPM allocation framework. Our preliminary experimental results show that our approach can improve significantly the performance of a set of programs selected from the Mediabench suite.

1 Introduction

The effectiveness of memory hierarchy is critical to the performance of a computer system. Traditionally microprocessors use cache to overcome the ever-widening gap between the processor speed and memory speed. However, cache introduces two major problems. First, it consumes a significant amount of processor power due to its complex tag-decoding logic. Second, it makes it very difficult to compute the worst-case execution time (WCET) of a program. In real-time systems, the schedulability analysis is based on the WCETs of all tasks. Therefore, many real-time systems do not use any cache. To overcome these two major problems, scratchpad memory (SPM) has been introduced. SPM consists of on-chip SRAM only. Therefore, it is much more energy efficient than cache [1]. Since SPM is managed by the compiler, it provides better time predictability. Given these advantages, SPM is widely used in embedded systems. In some high-end

© Springer-Verlag Berlin Heidelberg 2007

embedded processors such as ARM10E, ColdFire MCF5 and Analog Devices ADSP-TS201S, a portion of the on-chip SRAM is used as an SPM. In some low-end embedded processors such as RM7TDMI and TI TMS370CX7X, SPM has been used as an alternative to cache.

Effective utilisation of SPM is critical for an SPM-based system. Research on SPM management has been focused on data allocation. The objective is to place most frequently used data in SPM so that the average execution time or the total energy consumption of a program is minimised [8,7,10,11,5]. Most existing SPM allocation schemes place data objects in SPM only if they can be stored entirely in SPM. Therefore, SPM cannot be used by the frequently accessed arrays that are larger than SPM, resulting in lower SPM utilisation and slower program execution. Since many applications have large arrays that are frequently accessed, new approaches are needed to partition them into smaller subarray blocks so that the smaller subarrays can be placed in SPM.

Data tiling or data shackling [6] is an array layout transformation technique that partitions a large array into smaller subarray blocks. It was originally proposed to improve the cache performance of regular loop kernels. Data tiling is typically applied together with loop tiling [12]. Loop tiling partitions the iteration space of a loop nest into iteration space tiles so that different subarray blocks will be accessed in different iteration space tiles. Loop tiling may change the order of array accesses in a loop nest. A legal tiling must preserve all dependences in the program. As a result, data tiling is usually restricted to arrays with regular access patterns. In the past, different data tiling techniques [2,6,4] have been proposed. Furthermore, Kandemir et al. [5] have applied data tiling to SPM allocation. Like other data tiling approaches, their approach targets regularly accessed arrays in loop kernels. It tiles arrays accessed in a particular loop nest and copies the small data tiles to SPM during program execution. The objective is to maximise the utilisation of SPM while minimising the incurred copy cost for a particular loop nest. Although the proposed technique can provide near-optimal results for certain loop nests, it has two major deficiencies:

1. It considers individual single loop nests in isolation. In a program with multiple loop nests, tiling arrays in one loop nest alone may often affect the tiling and allocation decisions for arrays accessed in others. In other words, the best tiling scheme chosen for a loop nest may not be desirable when all the loop nests in the program are considered as a whole. To maximise the overall SPM utilisation while minimising the copy cost for a program, we need to consider all arrays accessed in different loop nests in the program and select the best tiling schemes for all loop nests in concert.
2. It does not handle arrays that are accessed via aliased pointers. In many applications, arrays are often accessed via aliased pointers. Applying data tiling to those arrays can sometimes be difficult partly because their memory access patterns are uncertain at compile time and partly because code rewriting for a tiling transformation can be complex.

In this paper, we present some preliminary results on applying a new approach to applying data tiling to a whole program in SPM allocation. Unlike previous work that is focused on a single loop nest, our approach considers all loop nests in a program simultaneously. We formulate the tiling problem for a program as an ILP (Integer Linear Programming) problem and use an ILP solver to find the optimal tiling schemes for all loop nests simultaneously as a global optimisation problem. We use an existing SPM allocation algorithm [8] to place all arrays, including the partitioned subarrays, in SPM. Furthermore, we use pointer analysis and profiling information to detect the most likely memory access patterns for pointer accesses and tile memory objects accordingly. As a result, some runtime checks are inserted to preserve all data dependences in the original program for a tiling transformation. In comparison with existing work on data tiling [2,6,5,4], we are presently able to tile some loop nests with pointer aliases. However, more research is needed to enlarge the scope of programs that can be handled. Our preliminary results obtained for a set of programs selected from the Mediabench suite are very encouraging. Our approach can improve significantly the performance of these programs in an SPM-based system.

The rest of the paper is organised as follows. Section 2 addresses some major challenges in data tiling. Our approach is then introduced in Section 3. We evaluate the effectiveness and efficiency of this approach in Section 4. Section 5 reviews some related work and Section 6 gives some future work.

2 The Challenges

We use an example to illustrate the two challenges we address in this work when applying data tiling to whole programs in SPM allocation. One is concerned with selecting the best tile sizes for arrays accessed in multiple loop nests simultaneously. The other lies in how to deal with the arrays that may be accessed via aliased pointers when these arrays are tiled.

This example, shown in Fig. 1(a), is abstracted from the benchmark unepic in Mediabench. There are two single loops executed in two separate functions. In the loop contained in *unquantize_pyr*, the global array *lo_filter* is accessed. In the loop contained in *unquantize_image*, some arrays may be accessed via two pointers *q_im* and *res*; these can be two different arrays or a common array, which may also be the global array *lo_filter*. It is clear that optimising either loop by data tiling in isolation may not be optimal for both loops.

Let us now consider some complications that arise when tiling loops with aliased pointer accesses. In the loop contained in function *unquantize_image*, memory objects are accessed via two pointers *q_im* and *res*. Although it appears that two different arrays are accessed via two pointers *q_im* and *res*, it is possible that only one array (which could be *lo_filter*) is accessed by both pointers at different offsets. Therefore, it is not trivial to tile the arrays that the two pointers point to while also preserving all data dependences in the original program. In addition, code rewriting for a given tiling transformation can be quite complex.

Fig. 1(b) illustrates one possible solution when the program is optimally tiled. The loop in *unquantize_pyr* is not tiled. Note that the global array *lo_filter* can be regarded as being tiled with a tile size equal to its original array size. The

```
int $lo\_filter[\,]$;
void unquantize_pyr(BinIndexType* $q\_im$, int* $res$, int $im\_size$) {
    for (...) {
        $\ldots = lo\_filter[\ldots]$;
        unquantize_image($q\_im$, $res$, $im\_size$);
    }
}
void unquantize_image(BinIndexType* $q\_im$, int* $res$, int $im\_size$) {
    for ($i = 0;\ i < im\_size;\ i++$) {
        $res[i] = q\_im[i] \ldots$
        ...
    }
}
```

(a)

```
int $lo\_filter[\,]$;
void unquantize_pyr(BinIndexType* $q\_im$, int* $res$, int $im\_size$) {
    for (...) {
        $\ldots = lo\_filter[\ldots]$
        unquantize_image($q\_im$, $res$, $im\_size$);
    }
}
void unquantize_image(BinIndexType* $q\_im$, int* $res$, int $im\_size$){
    // tile size is $S_a$ for both $res$ and $q\_im$
    if ($res$ and $q\_im$ point to the same array) {
        for ($i = 0;\ i < im\_size;\ i++$) {
            $res[i] = q\_im[i] \ldots$
            ...
        }
    }
    else {
        for ($it = 0;\ it < im\_size;\ it = it + S_a$) {
            **read_tile** $q\_im[it : it + S_a - 1] \rightarrow q\_im'[0 : S_a - 1]$
            for ($i' = 0;\ i' < S_a;\ i'++$) {
                $res'[i'] = q\_im'[i'] \ldots$
                ...
            }
            **write_tile** $res'[0 : S_a - 1] \rightarrow res[it : it + S_a - 1]$
        }
    }
}
```

(b)

Fig. 1. (a) A code snippet abstracted from the benchmark unepic in Mediabench. For simplicity, we assume that *im_size* is a multiple of S_a. (b) The modified code after tiling *q_im* and *res* with a tile size of S_a (the tiled code is boxed).

loop in *unquantize_image* is tiled only when two pointers *q_im* and *res* are not aliases. A runtime test will be performed to determine whether the original loop or the tiled version will be executed. To avoid unnecessary runtime overhead, such code will be generated only if the two pointers frequently point to different memory objects based on profiling information. In the tiled code, the arrays pointed to by *q_im* and *res* are tiled with a common size of S_a. Two new arrays, called *tile arrays* and denoted q_im' and res', have been introduced to store all data elements accessed in the partitioned subarray tiles for *q_im* and *res*, respectively. At the beginning of loop i', a subarray of *q_im* is copied to q_im'. At its end, res' is copied to a subarray of *res*.

This example shows that allocation of tile arrays such as q_im' and res' in SPM may affect the allocation of other arrays in the program. In addition, selecting a large tile size for a tile array may decrease the chances for the global array lo_filter to be placed in SPM. Therefore, we need to consider the SPM size, the cost and benefit of tiling certain arrays, and the access frequencies of all other arrays in the program together to select the best tile sizes for all arrays.

3 Our Approach

In embedded applications (e.g., programs in C/C++), array accesses may take different forms. An array may be accessed by using its name and an index or via aliased pointers. So we need to identify all arrays of different forms in a program.

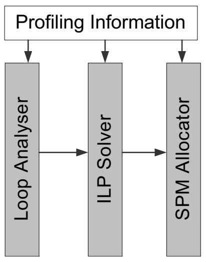


Fig. 2. A framework for applying data tiling to SPM allocation

As illustrated in Fig. 2, our approach consists of three components: Loop Analyser, ILP Solver and SPM Allocator. First, the loop analyser analyses each loop nest based on pointer analysis and profiling information. It identifies all the arrays that are either implicitly or explicitly accessed. Next, the ILP solver formulates the data tiling problem for a program as an ILP problem and uses an ILP solver to find which arrays should be SPM-resident and the best tile sizes for all SPM-resident tile arrays (such as q_im' and res' in Fig. 1(b)). Lastly, the SPM allocator places in SPM all the SPM-resident arrays suggested by the ILP solver by using an existing SPM allocation framework [8].

In our current implementation, we assume that the live ranges of all arrays (including tile arrays) satisfy the so-called containment relationship as defined in [8]. That is, two live ranges must be such that either they are disjoint or one contains the other. In this case, all SPM-resident arrays determined by the ILP solver can always be placed in SPM by the SPM allocator. If some live ranges do not satisfy this containment relationship, there are two options. We can apply the approaches reported in [8,7] to obtain a possibly sub-optimal solution, in which case, some arrays are spilled to off-chip memory even if they are suggested to be placed in SPM by the ILP solver. Alternatively, the ILP solver and SPM allocator can be merged and solved optimally based on ILP after some additional constraints on the placements of SPM-resident arrays are added.

3.1 Loop Analyser

Loop tiling concentrates on iteration space tiling [12]. Data tiling focuses on tiling the data space of an array. In general, data tiling is done together with loop tiling [2,6,4,5] so that when each iteration space tile is executed, different subarray tiles of an array can be accessed for improving data locality in a program.

Our loop analyser analyses each loop nest L to identify all the arrays that are either explicitly or implicitly accessed and a subset of these arrays that are considered to be tiled at L. For each array P that may be tiled at L, we introduce an array T_{L_P}, called *tile array*, to store the current values of a subarray tile of P. In theory, there can be many possible choices for $S_{T_{L_P}}$, known as the *tile size* of T_{L_P}. However, this may cause an ILP solver to spend too much time on finding a feasible solution. Therefore, for each tile array T_{L_P}, we pre-define a set of values to confine our search for the best tile size for T_{L_P}.

In this work, all arrays accessed in a particular loop nest will be tiled by rectangular tiles with possibly different tile sizes. The tile size used for tiling the loop nest is assumed to be deduced from the tile sizes for all tile arrays. How to tile a given loop nest this way is likely dependent on the computation and memory access characteristics of the loop nest.

In Fig. 1(b), q_im' and res' are tile arrays introduced to tile q_im and res, respectively. The tile sizes for both tile arrays happen to be identical and are denoted S_a, which is also used to tile the enclosing loop in *unquantize_image* into a two-dimensional loop nest in the tiled code.

3.2 ILP Solver

We use an ILP formulation to model the data tiling problem for a whole program in SPM allocation. Let $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$ be a subset of these arrays identified as tile arrays by the loop analyser. Let $\mathcal{T} = \{T_1, T_2, \ldots, T_m\}$ be the set of tile arrays introduced by the loop analyser to tile some arrays in $\mathcal{P}$. So $\mathcal{U} = \mathcal{P} \bigcup \mathcal{T}$ represents the set of all array candidates to be considered in SPM allocation.

Each tile array T_i may be aliased with some or all the arrays in $\mathcal{P}$. Let Q_{T_i,P_j}, where $1 \leqslant j \leqslant n$, be the probability that each array access via T_i turns out to be an array access to P_j. This information can be obtained via profiling.

For each SPM candidate $U_i \in \mathcal{U}$, we define a binary variable X_{U_i} to denote whether U_i is placed in SPM or not:

$$X_{U_i} = \begin{cases} 1 & U_i \text{ is placed in SPM} \\ 0 & U_i \text{ is } \textit{not} \text{ placed in SPM} \end{cases} \tag{1}$$

In practical applications, the tile sizes for all tile arrays in a particular loop nest L can be different but are related. Let all their tile sizes be specified by r_L independent integer variables represented by an r_L-dimensional vector $(S_{L,1}, S_{L,2}, \ldots, S_{L,r_L})$, whose value will be searched from a pre-defined set:

$$(S_{L,1}, S_{L,2}, \ldots, S_{L,r_L}) \in TileSizeSet_L \tag{2}$$

The tile size for tiling L is determined from $(S_{L,1}, S_{L,2}, \ldots, S_{L,r_L})$ and the loop bounds of L. If both are given, then the total number of copy operations for a copy statement (like **read_tile** or **write_tile** in Fig. 1(b)) will be known.

We perform a cost-benefit analysis to decide whether an array should be placed in SPM or not. We distinguish two kinds of arrays: those in $\mathcal{P}$ and those in $\mathcal{T}$. The benefit of placing an array $P_i \in \mathcal{P}$ in SPM can be modelled as:

$$B_{P_i} = \text{Number of accesses to } P_i \text{ in the program} \times \beta \tag{3}$$

where β is the benefit of turning a single memory access to the off-chip memory into one to the SPM (in cycles). For a tile array $T_i \in \mathcal{T}$, however, there will be no benefit to place T_i in SPM for those accesses to T_i that turn out to be the accesses to an array $P_j \in \mathcal{P}$ such that P_j has already been placed in SPM (since T_i is aliased to P_j). Therefore, the benefit of placing T_i in SPM is:

$$B_{T_i} = \text{Number of accesses to } T_i \text{ in the program} \times \beta \times \left(1 - \sum_{P_j \in \mathcal{P}} (Q_{T_i,P_j} \times X_{P_j})\right) \tag{4}$$

The cost of placing an array $P_i \in \mathcal{P}$ in SPM, denoted C_{P_i}, is simply set to be 0 since P_i will be stored in SPM throughout its lifetime:

$$C_{P_i} = 0 \tag{5}$$

Thus, no array copying between SPM and off-chip memory is needed.

The cost of placing a tile array $T_i \in \mathcal{T}$ in SPM is the cost incurred in all copy operations inserted in the tiled code for copying all subarrays of all aliased array that have been tiled into T_i between SPM and off-chip memory (cf. Fig. 1(b)). When copying a subarray tile to SPM, we assume that the underlying architecture supports DMA transfer so that a continuous memory block can be copied to SPM by one DMA operation. The DMA copy cost of transferring a memory block of n bytes size is $C_s + C_t \times n$, where C_s is the DMA start up cost and C_t is the DMA transfer cost per byte. Therefore, the cost of placing a one-dimensional tile array T_i in a loop nest L with a tile size of $Size_{T_i}(S_L)$, denoted C_{T_i}, in SPM is approximated by:

$$C_{T_i} = copy\_count(L, S_L) \times (C_s + C_t \times Size_{T_i}(S_L)) \tag{6}$$

Here, $S_L \in TileSizeSet_L$ and $copy\_count(L, S_L)$ represents the number of DMA operations for T_i between SPM and off-chip memory. In this one-dimensional case, $Size_{T_i}(S_L)$ is a particular component of S_L. In general, $Size_{T_i}(S_L)$ is determined by several components in S_L depending on the dimensionality of T_i. The cost formulas for higher-dimensional tile arrays can be developed similarly.

Our objective function is to maximise the difference between the sum of all benefits and the sum of all costs involved in tiling a given program:

$$\sum_{P_i \in \mathcal{P}} (B_{P_i} \times X_{P_i}) + \sum_{T_i \in \mathcal{T}} (B_{T_i} \times X_{T_i}) - \sum_{T_i \in \mathcal{T}} (C_{T_i} \times X_{T_i}) \tag{7}$$

As in any linear optimisation problem, a set of constraints is also defined. We require that, at any program point during program execution, the sum of sizes of all SPM-resident arrays be no larger than the given SPM size. Such information can be approximated by examining the interference graph formed by all SPM candidates. For every maximal clique $\mathcal{C}$ in the graph, we have the following constraint:

$$\sum_{U_i \in \mathcal{U} \text{ is an array node in } \mathcal{C}} (M_{U_i} \times X_{U_i}) \leqslant SPM\_size \tag{8}$$

where M_{U_i}, which represents the size of array U_i, is further refined below.

The above formulation cannot be fed into an ILP solver directly. However, this can be overcome by transforming all non-linear constraints into linear ones. Let us illustrate this process for a loop nest L by assuming that all arrays accessed in L are one-dimensional. First of all, (2) and (6) can be linearised by introducing $|TileSizeSet_L|$ binary variables. Let the binary variable $G_{L,j}$ be 1 if all tile arrays in L are tiled by tile sizes using the values in $S_{L,j} \in TileSizeSet_L$ and 0 otherwise. Thus, $\sum_{S_{L,j} \in TileSizeSet_L} G_{L,j} = 1$. We can then replace C_{T_i} in (7) by $\sum_{S_{L,j} \in TileSizeSet_L} (G_{L,j} \times copy\_count(L, S_{L,j}) \times (C_s + C_t \times Size_{T_i}(S_{L,j})))$. Since $S_{L,j}$ is now a constant vector, $Size_{T_i}(S_{L,j})$ is constant. In addition, $copy\_count(L, S_{L,j})$, which depends on only the loop bounds of L, is also a constant (or a parameterised constant). Next, M_{U_i} in (8) is defined as follows. If $U_i \in \mathcal{P}$, then M_{U_i} is the array size of U_i. If $U_i \in \mathcal{T}$, then M_{U_i} accessed in a loop nest L is replaced by $\sum_{S_{L,j} \in TileSizeSet_L} (G_{L,j} \times Size_{U_i}(S_{L,j}))$. Finally, the resulting formulation will still consist of products of binary variables. They can also be linearised by introducing extra binary variables. Such transformation techniques are standard and thus omitted. A commercial ILP Solver, CPLEX, is used to solve the ILP problem for tiling a given program.

3.3 SPM Allocator

After the ILP solver has determined which arrays should be SPM-resident and the tile sizes for those SPM-resident tile arrays, we will now determine the SPM addresses for the SPM-resident arrays. This will be realised using an SPM allocation algorithm proposed by Lian et al. [8]. The SPM allocation algorithm formulates the SPM allocation problem as an *interval colouring problem* [3].

Given that for every two SPM-resident arrays, their live ranges are either disjoint or containing-related, all SPM-resident arrays can be placed in SPM.

4 Experimental Results

4.1 Benchmarks

We have used six benchmarks from Mediabench as shown in Table 1. For each benchmark, Column 3 gives the number of arrays in each benchmark. Column 4 shows the total size of all arrays in each benchmark. Arrays that are not accessed in a benchmark are not counted. In the two benchmarks, epic and unepic, two very large arrays (larger than 64K bytes) that are accessed infrequently cannot be tiled. Therefore, they are excluded in SPM allocation. In Table 1, only the original arrays in a program are considered when the data set size for the program is calculated; the tile arrays introduced by data tiling are excluded. Benchmarks with different data set sizes are evaluated with different SPM sizes.

Table 1. Benchmarks from MediaBench

Benchmark	#Lines	#Arrays	Data Set of Arrays (Bytes)	SPM Sizes (Bytes)
rawcaudio	741	5	2.9K	{512, 1024, 2048, 4096}
rawdaudio	741	5	2.9K	{512, 1024, 2048, 4096}
epic	3524	4	344	{256, 512, 1024, 2048}
unepic	3524	4	344	{256, 512, 1024, 2048}
mpeg2encode	8304	62	9.2K	{1024, 2048, 4096, 8192}
mpeg2decode	9832	76	21.8K	{1024, 2048, 4096, 8192}

All programs are compiled into assembly programs for the Alpha architecture using the SPM allocation framework described in [8]. These assembly programs are then translated into binaries on a DEC Alpha 20264 architecture. The profiling information for MediaBench is obtained using *the second data set* provided by the MediaBench web site. These benchmarks are evaluated using the data sets that come with their source files.

4.2 Performance Evaluation

We have modified SimpleScalar in order to carry out the performance evaluations for this work. As in [7,8], our target architecture has only on-chip SPM and off-chip memory. There are four parameters involved in the execution model. The cost of transferring n bytes between SPM and off-chip memory is approximated by $C_s + C_t \times n$ in cycles, where C_s is the startup cost and C_t is the cost per byte transfer. Two other parameters are M_{spm} and M_{mem}, which represent the

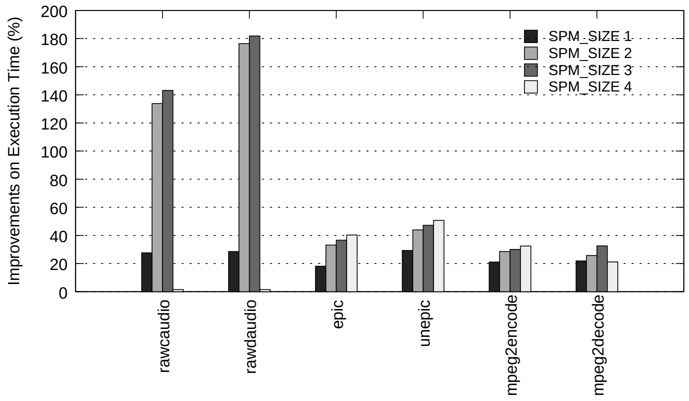


Fig. 3. Performance improvements when data tiling is applied

number of cycles required for one memory access to the SPM and the off-chip memory, respectively. In all our experiments, the values of the four parameters are set to be $C_s = 100, C_t = 1, M_{mem} = 100$ and $M_{spm} = 1$.

We have evaluated the effectiveness of our data tiling approach by comparing it with the case when data tiling is not used. All six benchmarks were evaluated with four different SPM sizes given in Table 1, denoted as SPM_SIZE1, SPM_SIZE2, SPM_SIZE3 and SPM_SIZE4 from the smallest to the largest size.

As shown in Fig. 3, all benchmarks exhibit significant performance improvements. For rawcaudio and rawdaudio, speedups of more than 130% are observed when the SPM size is set to 1024 or 2048 bytes. However, there are no performance improvements for these two benchmarks when the given SPM size is 4K bytes. The reason can be explained using the SPM hit rates as shown in Fig. 4.

Fig. 4 illustrates the advantages of applying data tiling in terms of SPM hit rate improvements. As a general trend, there are significantly more access hits in SPM when data tiling is applied except for rawcaudio and rawdaudio when the SPM size is set to 4K bytes. In these two exceptional cases, all arrays can be placed in SPM without resorting to tiling. So data tiling is not helpful in these

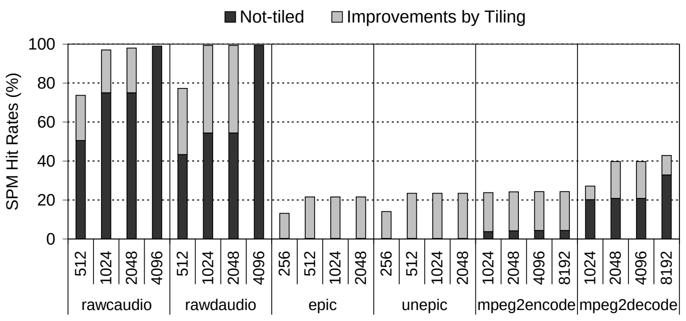


Fig. 4. SPM hit rate improvements when data tiling is applied

two cases. The other four benchmarks enjoy significant speedups and SPM hit rate increases for all SPM configurations.

5 Related Work

Existing SPM allocation approaches are either static or dynamic, depending on whether an array can be copied into and out of SPM during program execution or not. A large number of early approaches are static. A dynamic approach can often outperform an optimal static one. Some dynamic approaches exist [10,11,7,8]. In [11], Verma et al. use an ILP formulation to find which memory object should be placed in SPM. In [10], Udayakumaran and Barua present a set of heuristics and apply them to a set of benchmarks. Lian et al. [7,8] have proposed two approaches. The first approach [7] transforms the SPM allocation problem into a well-understood register allocation problem and the second approach [8] converts the SPM allocation problem into an interval colouring problem. The data tiling techniques proposed in this paper can be applied in the above SPM allocation schemes to further improve SPM utilisation.

Data tiling [6] is an array layout transformation technique that partitions the data space of an array into smaller tiles. The technique was originally proposed to improve cache performance because the partitioned data tiles are more likely to fit in cache blocks. Various techniques on how to effectively tile array accesses in different loops kernels have been proposed [2,4,9].

Recently, researchers have applied data tiling to SPM allocation. Kandemir et al. [5] exploited this technique in SPM allocation. Their approach partitions an array into tiles, then copies the tiles to SPM during program execution. Loop and data transformations are mentioned in their paper to improve the data reuse of each tile so that the copy cost can be minimised. Chunhui et al. [13] also proposed an approach that combines loop tiling and data tiling for SPM-based systems. Both approaches target on individual loop nests only and do not handle arrays accessed via aliased pointers. Our approach is the first one that handles multiple loop nests consisting of arrays that may be accessed via aliased pointers.

6 Conclusion

Data tiling was originally proposed to improve the cache performance of regular loops. Recently, researchers have applied this technique to scratchpad memory allocation. However, previous data tiling approaches are focused on a single loop nest only. A good tiling scheme for a particular loop nest may be a bad one for other loop nests due to the live range interferences of arrays. In addition, previous works cannot handle arrays accessed via aliased pointers. In this paper, we have proposed a new data tiling approach. Our approach formulates the data tiling problem for a whole program as an ILP problem and uses an ILP solver to find the optimal tiling schemes for all loop nests in the program simultaneously. Compared to previous approaches, our approach has two unique features. First, it tiles all loop nests in concert. Second, it can handle arrays accessed via aliased pointers. Validation using benchmark programs has confirmed the effectiveness of our approach. One future work is to generalise our approach by considering non-rectangular tiles. Another is to generalise our approach to tile loop nests whose arrays may be accessed by more complex alias relations.

References

1. Banakar, R., Steinke, S., Lee, B.-S., Balakrishnan, M., Marwedel, P.: Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In: CODES'02: Proceedings of the 10th International Symposium on Hardware/Software Codesign, pp. 73–78. ACM Press, New York (2002)
2. Chatterjee, S., Jain, V.V., Lebeck, A.R., Mundhra, S., Tethodi, M.T.: Nonlinear array layout for hierarchical memory systems. In: ACM International Conference on Supercomputing (ICS'99), Rhodes, Greece, June 1999, pp. 444–453. ACM Press, New York (1999)
3. Golumbic, M.C.: Algorithmic graph theory and perfect graphs. Annals of Discrete Mathematics (2004)
4. Huang, Q., Xue, J., Vera, X.: Code tiling for improving the cache performance of pde solvers. In: ICPP'03: Proceedings of the International Conference on Parallel Processing (2003)
5. Kandemir, M., Ramanujam, J., Irwin, J., Vijaykrishnan, N., Kadayif, I., Parikh, A.: Dynamic management of scratch-pad memory space. In: DAC'01: Proceedings of the 38th Conference on Design Automation, pp. 690–695. ACM Press, New York (2001)
6. Kodukula, I., Ahmed, N., Pingali, K.: Data-centric multi-level blocking. ACM SIGPLAN Notice 32(5), 346–357 (1997)
7. Li, L., Gao, L., Xue, J.: Memory coloring: a compiler approach for scratchpad memory management. In: PACT'05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques, pp. 329–338. IEEE Computer Society, Los Alamitos (2005)
8. Li, L., Nguyen, Q.H., Xue, J.: Scratchpad allocation for data aggregates in superperfect graphs. In: LCTES '07: Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools, pp. 207–216. ACM Press, New York (2007)
9. Strout, M.M., Carter, L., Ferrante, J.: Compile-time composition of run-time data and iteration reorderings. In: PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, pp. 91–102. ACM Press, New York (2003)
10. Udayakumaran, S., Barua, R.: Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In: CASES'03: Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems, pp. 276–286. ACM Press, New York (2003)
11. Verma, M., Wehmeyer, L., Marwedel, P.: Dynamic overlay of scratchpad memory for energy minimization. In: CODES+ISSS'04: Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, pp. 104–109. ACM Press, New York (2004)
12. Xue, J.: Loop tiling for parallelism. Kluwer Academic Publishers, Boston (2000)
13. Zhang, C., Kurdahi, F.: On combining iteration space tiling with data space tiling for scratch-pad memory systems. In: ASP-DAC '05: Proceedings of the 2005 conference on Asia South Pacific design automation, pp. 973–976. ACM Press, New York (2005)

Evolution of NAND Flash Memory Interface[*]

Sang Lyul Min, Eyee Hyun Nam, and Young Hee Lee

School of Computer Science and Engineering, Seoul National University, Shinlim-dong, Kwanak-gu, Seoul 151-744, Republic of Korea
Tel.: +82-2-880-7047; Fax: +82-2-880-7589
symin@archi.snu.ac.kr

Abstract. In this paper, we describe the basics of NAND flash memory and describe the evolution of its interface to facilitate easy integration, to provide high bandwidth, to offer disk-like interface, and/or to guarantee interoperability.

Keywords: NAND Flash Memory, Block Device Interface, FTL.

1 Introduction

This paper describes the basics of NAND flash memory and explains the evolution of its interface. The paper is organized as follows. Section 2 explains a high-level interface of NAND flash memory along with its peculiar features. Then in Section 3, we describe the evolution of NAND flash memory interface to facilitate easy integration, to provide high bandwidth, to offer disk-like interface, and/or to guarantee interoperability. Finally, we offer conclusions in Section 4.

2 NAND Flash Memory

Fig. 1 gives a high-level interface of a NAND flash memory, the type of flash memory used for mass storage. The NAND flash memory consists of a set of blocks that in turn consist of a set of pages where each page has the data part that stores the user data and the spare part that stores meta-data associated with user data such as ECC. Although different sizes may be used, currently the most popular block size is 128 Kbytes consisting of 64 pages of 2 Kbytes [1]. There are three basic operations to NAND flash memory: read page, program page, and erase block. The read page operation, given the chip number, the block number, and the page number returns the contents of the addressed page, which takes about 20 us excluding the data transfer time. Likewise, the program page operation writes the supplied contents to the target page and takes about 200 us, again excluding the data transfer time. Unlike a write operation to other types of storage medium, the program operation can change the stored bits from 1 to 0 only. Therefore, the write operation is implemented by selectively changing bits from 1 to 0 to match the supplied contents assuming all bits

[*] This work was supported by the IT R&D program of MIC/IITA. [2006-S-040-01, Development of Flash Memory-based Embedded Multimedia Software].

© Springer-Verlag Berlin Heidelberg 2007

in the target page are 1's before the program operation. In flash memory, the only way to change a bit in a page from 0 to 1 is to erase the block that contains the page. The erase operation sets all bits in the block to 1 and it takes about 2 ms.

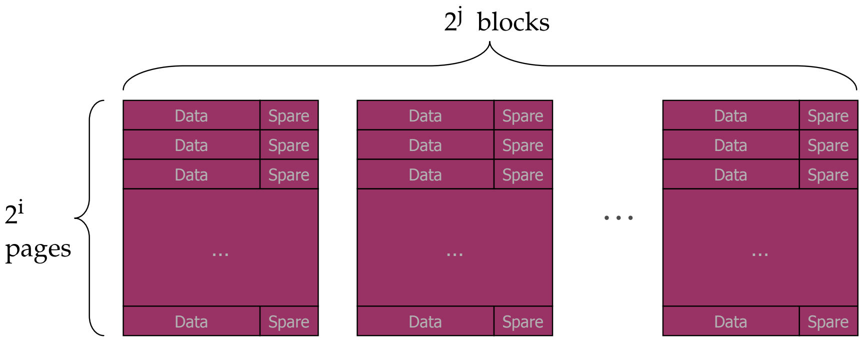


Fig. 1. NAND flash memory chip

One notable characteristic of NAND flash memory is that it is allowed to have a limited number of bad blocks at the manufacturing time to improve the yield. Moreover, additional blocks become bad at run time. Another notable characteristic is that NAND flash memory has a limit to the number of times a block can be erased and re-written, typically in the range of 10,000 to 100,000 times.

3 NAND Flash Memory Interfaces

3.1 Conventional NAND Flash Interface

Fig. 2 shows the interface for conventional NAND flash memory [1]. The chip enable signal (CE) enables the chip to respond to other signals. The I/O signals (I/O_7~I/O_0) carry not only data but also commands and addresses, which helps reduce the pin count. To distinguish among data, address, and command, the CLE and ALE signals are used. For example, when CLE is asserted, it means that a command is currently on the bus. The two strobe signals, WE and RE signals, are for clocking data in and out of NAND flash memory, respectively, in a byte-serial fashion.

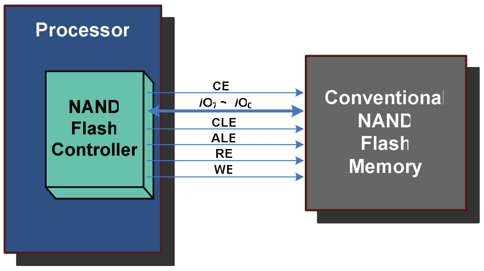


Fig. 2. Conventional NAND flash interface

3.2 OneNAND Interface

One of the problems of the conventional NAND flash interface is that it requires a dedicated interface of its own. This results in either (1) an increased pin count of the processor when the processor has a dedicated interface to the NAND Flash memory or (2) a degraded performance when the processor uses GP I/O (General Purpose Input/Output) pins to access the NAND flash memory.

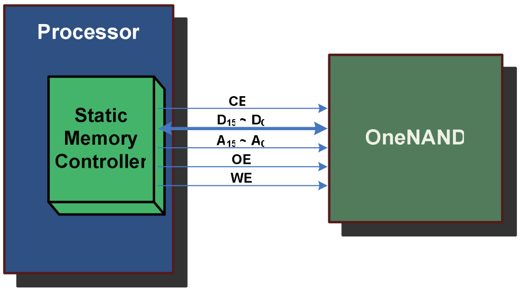


Fig. 3. OneNAND Interface

To address the problem above, OneNAND flash memory was introduced in 2005 that uses a traditional static memory interface as shown in Fig. 3 [2]. OneNAND uses memory-mapped I/O to access command registers such as for flash commands and addresses and also for status registers such as for program/erase status. In addition, OneNAND has on-chip SRAM for booting purposes and also for buffering data in and out of flash memory. These features together with the high density and high program/erase speeds of NAND flash memory make OneNAND an attractive storage choice for mobile handsets and PDAs.

3.3 Hyper Link NAND Flash Interface

Both the conventional NAND flash interface and the OneNAND interface use a shared bus topology to connect multiple chips. This shared bus topology, although

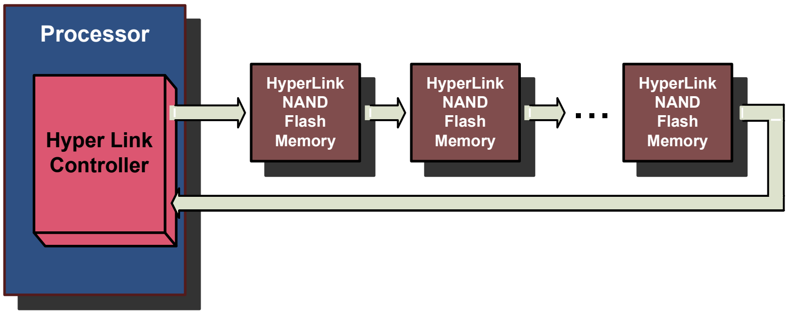

Processor
Hyper Link Controller
HyperLink NAND Flash Memory
HyperLink NAND Flash Memory
. . .
HyperLink NAND Flash Memory

Fig. 4. Hyper Link NAND Flash Interface

simple, has an upper limit on the bandwidth. In addition, the shared bus limits the number of chips that can be connected to the bus due to signal integrity problems. Because of this scalability problem, shared bus-based NAND flash interface cannot meet the demanding requirements of high capacity and high performance mass storage applications such as solid state disks (SSDs).

To address the scalability problem, the Hyper Link NAND flash interface has been proposed that connects multiple chips in a daisy-chain fashion using multiple point-to-point serial links (cf. Fig. 4) [3]. Each point-to-point link carries packets containing commands, addresses, and data at speeds up to 800 MB/s.

3.4 Block-Oriented NAND Flash Interface

As explained in Section 2, NAND flash memory provides operations that are not compatible with the interface provided by hard disk drives known as the block device interface that involves an overwrite semantics in the case of write operation. The block device interface has been a de facto standard for accessing storage devices mainly because of the dominance of hard disk drives as storage devices since their introduction in 1956 by IBM.

To bridge the gap between the operations provided by NAND flash memory and those required by the block device interface, a software module called the Flash Translation Layer (FTL) is commonly used. The main functions of the FTL are (1) dynamic remapping between sectors (units of 512 bytes visible at the block device interface) and physical blocks/pages in NAND flash memory, (2) wear-leveling that evens out the erase counts of physical blocks, and (3) bad block management.

By incorporating a processing element in the same package as NAND flash memory and implementing the FTL using the processing element, the block interface can be provided. This is the approach taken by MoviNAND by Samsung [4] and iNAND by SanDisk [5], and it greatly simplifies the integration of NAND flash memory into the system and improves the storage system performance by offloading the FTL from the host processor.

3.5 ONFI (Open NAND Flash Interface)

One serious problem of the NAND flash industry has been the lack of standards for multi-sourced NAND flash memory chips. Although suppliers of NAND flash memory use similar command sets and electrical specifications, there are subtle differences that prohibit NAND flash memory chips from different manufacturers to be “true” drop-in replacements of each [6].

The lack of standardization of NAND flash memory is being addressed by the Open NAND Flash Interface (ONFI) organization whose goal is to define a uniform NAND flash component interface. ONFI has published standards that define a common command set, timing parameters, and pin-outs for ONFI-compliant NAND flash memory chips. One notable feature of the ONFI standard is a Read Parameter Table that self-describes the functionalities of the NAND flash device, a feature very similar to the “identify drive” feature in hard disk drives.

4 Conclusion

In this paper, we have explained the basics of NAND flash memory and described the evolution of its interface. The evolution has been directed to facilitate easy integration, to provide high bandwidth, to offer disk-like interface, and/or to guarantee interoperability. Although there is an on-going standardization effort by the ONFI organization, it is expected that multiple heterogeneous interfaces will co-exist for the time being, and an interim solution such as the one proposed in [7] that provides a consistent software/hardware interface to NAND flash memory out of heterogeneous interfaces seems to be appropriate.

References

1. Samsung Electronics, Co.: 2G x 8 Bit / 4G x 8 Bit / 8G x 8 Bit NAND Flash Memory Data Sheets (2007), http://www.samsung.com/Product/Semiconductor/NANDFlash/
2. Kim, B., Cho, S., Choi, Y., Choi, Y.: OneNAND^{TM}: A High Performance and Low Power Memory Solution for Code and Data Storage. In: Proceedings of the 20th Non-Volatile Semiconductor Workshop (2004)
3. MOSAID Technologies Incorporated, Unleasing the Next Generation Flash Memory Architecture: HyperLink NAND (HLNAND^{TM}) Flash (2007)
4. http://www.samsung.com/Products/Semiconductor/moviNAND/index.htm
5. http://www.sandisk.com/OEM/ProductCatalog(1295)-SanDisk_iNAND.aspx
6. Kamat, A.: Simplifying Flash Controller Design. ONFI Organization (2007)
7. Lee, Y.H.: Design and Implementation of a Flash Memory Controller for Heterogeneous NAND Flash Memories. Seoul National University, MS Thesis in preparation (2007)

FCC-SDP: A Fast Close-Coupled Shared Data Pool for Multi-core DSPs*

Dong Wang, Xiaowen Chen, Shuming Chen, Xing Fang, and Shuwei Sun

School of Computer, National University of Defense Technology
nudtjum@163.com

Abstract. Multi-core Digital Signal Processors (DSP) have significant requirements on data storage and memory performance for high performance embedded applications. Scratch-pad memories (SPM) are low capacity high-speed on-chip memories mapped with global addresses, which are preferred by embedded applications than traditional caches due to their better real-time characterization. We construct a new Fast Close-Coupled Shared Data Pool (FCC-SDP) for our multi-core DSP project based on SPMs. FCC-SDP is organized as multi-bank parallel structure with double-bank interleaving access modes, and provides a fast transmission path for fine-grain shared data among DSP cores. We build the behavior simulator of FCC-SDP and make design realization. Simulation experiments with several typical benchmarks show that FCC-SDP can well capture the fine-grain shared data in multi-core applications, and can achieve average speedup ratio of 1.1 and 1.14 compared with traditional shared L2 caches and DMA transmission modes respectively.

1 Introduction

Multi-core Digital Signal Processors (DSP), such as OMAP ®, are recently emerging multi-processor SoCs for high performance embedded applications. Multi-core DSPs need much higher memory bandwidth and more flexible memory structure to meet the requirement of parallel computations than traditional single-core DSPs. It is critical to ensure most data requests fulfilled by on-chip memories, as the latency and power consumption of accessing off-chip memories is becoming intolerable.

Scratch-Pad Memories (SPMs) are low capacity on-chip memories mapped with global addresses and can be accessed directly by LOAD/STORE instructions. SPMs have special advantages in area and power consumption than caches due to their simpler control logics and better data access definition, and can ensure single cycle (or certain cycles) access delay [3, 4, 5, 6, 7]. These special features make SPMs more suitable to embedded real-time applications than traditional caches. In term of the features of SPMs, we propose a new on-chip shared close-coupled SPM structure, FCC-SDP, for our heterogeneous multi-core DSP project. FCC-SDP provides a fast transmission path for fine-grain shared data among DSP cores, and would benefit MPSoCs by its low access delay and high transmission efficiency.

* Funded by the project of National Science Foundation of China (No. 60473079).

© Springer-Verlag Berlin Heidelberg 2007

2 Related Works

Data optimization by on-chip scratch-pad memories has been a research topic since the last decade. Banakar et al. [8] computed the area and energy consumption for different size of SPMs and caches. Their results showed scratch-pad memory with an average energy reduction of 40%, and average area-time reduction of 46% against caches. Issenin et al. [9] presented a novel multiprocessor data reuse analysis technique to arrange the most common used data in low capacity on-chip SPMs, and replace the global accesses with the local ones for the reduction of power. Mathew and Davis [10] proposed a low energy and high performance SPM system for VLIW processors. Their system made use of array variable rotation technique replacing register renaming. Kandemir and Suhendra et al. [1, 7] proposed a multi-processor model including a Virtual Shared Scratch-Pad Memory (VS-SPM).

Most of these works have exploited the storage allocation and data management problems of SPMs. The corresponding research work in multi-core processors is rarely found. However, different SPMs organizations would have distinct effect on the performance of MPSoCs. It is necessary to exploit SPM structures based on the applications requirement and communication features of multi-core processors.

3 Architecture Prototype of the Multi-core DSP

QDSP is our heterogeneous multi-core DSP prototype containing five processor cores, which are four 32b floating-point DSP cores and one 32b RISC core, as shown in Fig. 1. The RISC core functions as the user interface and tasks manager, and the four DSP cores, which are VLIW architectures, are used for background data processing. Each DSP core has its private two levels of caches and external memory interface (EMCI). FCC-SDP is on the same hierarchy with L1 caches, and forms two parallel

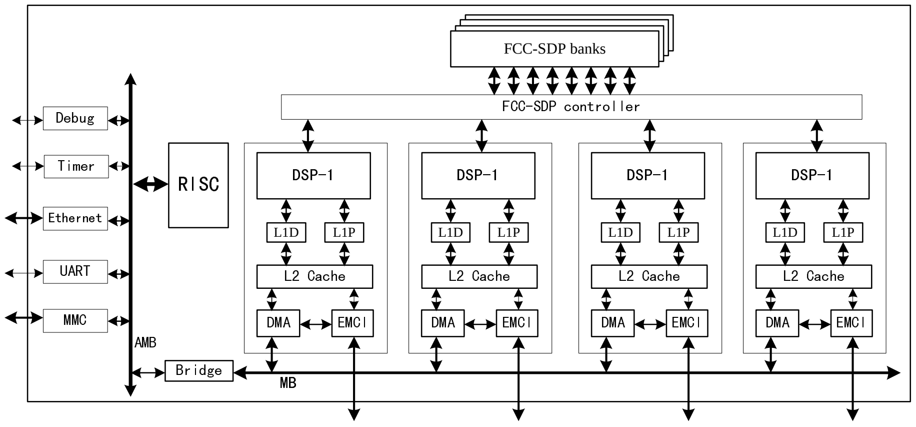


Fig. 1. The heterogeneous multi-core DSP architecture: QDSP

data paths together with the caches. Some fine-grain shared data, such as global variables and coefficient matrixes, can be transferred by FCC-SDP, and other data and commands could be transferred by some on-chip inter-links.

4 FCC-SDP Architecture

FCC-SDP is organized by four lanes, and each lane connects with a DSP core, as shown in Fig. 2. Each lane is composed of two same sized memory banks SiA and SiB (i=1, 2, 3, 4), a bank controller, and Read/Write queues. The four lanes are interconnected by the on-chip communication links. All the eight banks are addressed sequentially to form the whole memory space of FCC-SDP. A group of control registers, Con_Reg, connects to the four DSP cores through the common configure bus Conf_Bus, used for the synchronization/exclusion operation.

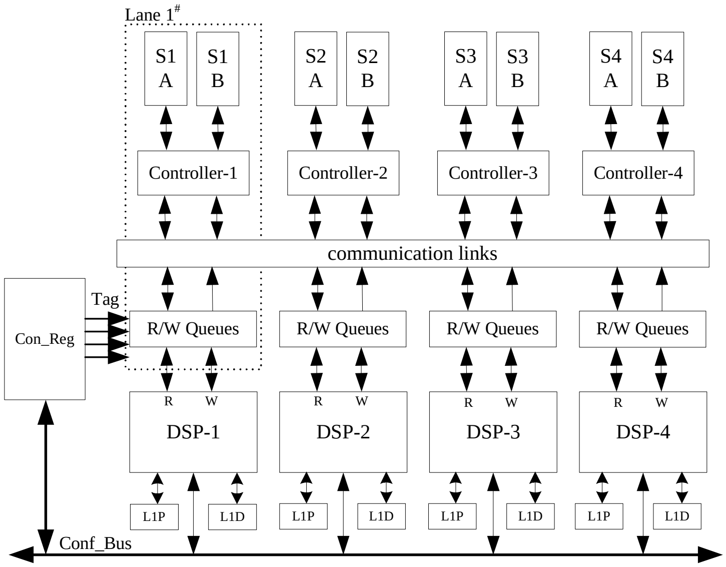


Fig. 2. The architecture of FCC-SDP

4.1 The R/W Queues with Bypass Logics

We configure a read buffer with depth of three entries and a write buffer with depth of one entry for each lane. These buffers can hold the blocked read or write requests due to bank conflicts and failed synch, and alleviate the effect of access latency on the pipeline of DSP cores. If a read request meets a failed synchronization, it and its successive two requests in cycle T +1 and T+2 will be held in the read buffer. If DSP cores can't receive the requested data of cycle T until cycle T+3, they won't issue new instructions and wait. Write buffers use an enable signal **WrtEN** to stall DSP pipelines under failed synchs. However, under successful synchs, our bypass logics will forward R/W requests to proper banks of FCC-SDP directly. The bypass logic improves the access efficiency of FCC-SDP, as shown in Fig. 3.

4.2 Dual-Mode Operations and Double-Bank Interleaving Access

As a shared close-coupled on-chip memory, FCC-SDP provides a fast transmission path for fine-grain shared data between two DSP cores. To improve the parallelism of FCC-SDP, we provide the dual-mode operation: private mode and shared mode.

- In private mode, each DSP core only can read or write the two bank SiA and SiB in its corresponding lane;
- In shared mode, each DSP can read, but can't write the banks in another three lanes. Any DSP must exchange its data with other DSPs through its own lane.

Although this shared mode limits the spaces of DSP cores write requests, it removes the write conflicts, simplifies the control logic for maintaining the consistency, and reduces the energy consumption.

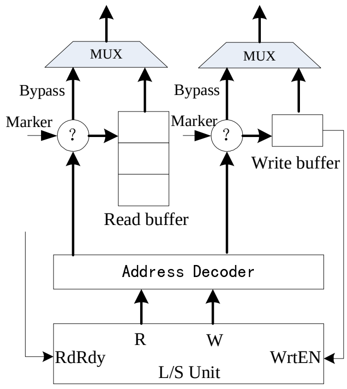


Fig. 3. The read/write buffers with bypass logics

To improve the parallelism of FCC-SDP transmission, we provide a double-bank interleaving access mode. **DSP-i** writes the first slice of shared data into its bank **SiA**, then releases bank **SiA** and writes the second slice into its bank **SiB**; at the same time, the consumer **DSP-j** (j≠i) begin reading the first slice from bank **SiA**. When both the write and read operation are completed, the two DSP cores switch their target banks: **DSP-i** releases bank **SiB** and writes the third slice to **SiA**, and **DSP-j** reads the second slice from **SiB**, and so on. When the execution time of each DSP core is balanced, the data transmission among DSP cores can be pipelined.

4.3 RC Model and Fast Synchronization Based on Marker Lights

In the Release-Consistency (RC) model [2], a synchronization operations include acquiring and releasing. We use the reduced RC model to realize a fast synchronization protocol for FCC-SDP. We configure three marker light registers for each memory bank as hard locks for synch operations. The marker light registers are mapped with global addresses and connect to the four DSP cores by the common configuration bus **Conf_Bus**. Each marker light has two states, light and dark, representing the bank data ready state and exhausted state respectively. The detailed synchronization procedure is as follows:

- After a producer DSP stores its shared data into bank **SiA** (or **SiB**), it turns on the marker lights of bank **SiA** (or **SiB**) by a STORE instruction. Before the marker light gets light, the corresponding bank is forbidden reading;
- After a consumer DSP reads off all shared data in bank **SiA** (or **SiB**), it turn off its corresponding marker lights also by a STORE instruction;
- After all consumers turn off their corresponding marker lights, the bank is released; otherwise the bank is forbidden writing;
- When a read (or write) request meets an unexpected marker light state, it will be appended to the tail of the read (or write) buffers automatically.

Since the path of shared data transmission is separated from the path of synch operations, we have to insert some independent instructions or NOPs as delay slots between the last read request for shared data and the darkening operation on the marker light, to avoid potential RC model violation and wrong synchronization. Fig. 4 gives an example of shared data transmission.

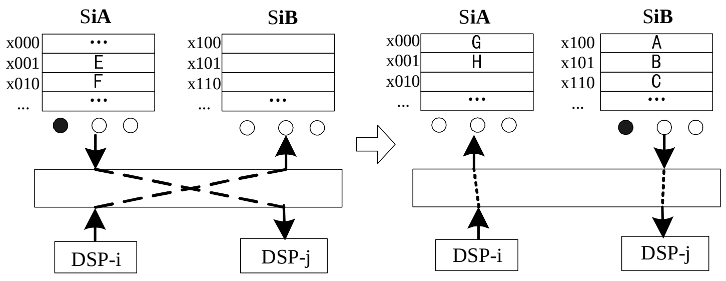


Fig. 4. An example of shared data transmission between two DSP cores

The automatic hardware synchronization is transparent to users. Besides the kind of synch mode, we provide another synch solution based on software query: inserting a routine before each lightening or darkening operation to poll the state of marker lights, and then decide the program branches. The poll routine is realized by read operations on the configuration bus, which can be paralleled with other write operations due to the separate read and write buses.

5 Selection of Single Bank Capacity

The single bank capacity ***C*** of FCC-SDP is a design parameter needing tradeoff, because it has different transmission process and synch overhead with different bank capacity. To be convenient for analyzing, let's suppose all data written by the producer core is read off by the consumer core; both the read and the write requests are issued continuously; the turnaround time of accessing different banks is omitted. Then, we establish an analysis model for the data transmission between two cores. Table 1 gives the parameters.

Table 1. The parameters for the analysis model

Parameter	Unit	Notation
W	Bytes	the width of memory bank
L	Entries	the depth of memory bank
C	Bytes	memory bank capacity, $C=W\cdot L$
V	Bytes	the volume of shared data between two cores
s	Cycles	the time consumption of one synch operation
D_r/w	Cycles	the access delay of FCC-SDP read/write
T	Cycles	the total time consumption for exchanging data *V*

(i) If $V \le C$, there is only one synch operation before finishing the transmission:

$$\begin{aligned} T = T_{write} + T_{read} &= (\frac{V}{W} + D\_w + s) + (\frac{V}{W} + D\_r + s) \\ &= \frac{2V}{\mathrm{W}} + 2s + D\_w + D\_r \end{aligned} \tag{1}$$

(ii) If $V > C$, the shared data should be partitioned as multi-transmission by the single bank capacity *C*. We define $T_{\mathrm{interleave}}$ is the time interval between two successive bank switch operations:

$$T_{\mathrm{int}erleave} = MAX\{L + D\_w + s, L + D\_r + s\} \tag{2}$$

The total transmission time *T* can be expressed as follows:

$$T = \begin{cases} (L + D\_w + s) + T_{\mathrm{int}erleave} \times \dfrac{V}{WL} \quad , & if \left\lfloor \dfrac{V}{WL} \right\rfloor = \dfrac{V}{WL} \\ (L + D\_w + s) + T_{\mathrm{int}erleave} \times \left\lfloor \dfrac{V}{WL} \right\rfloor + (\dfrac{V}{W} - \left\lfloor \dfrac{V}{WL} \right\rfloor \cdot L) + D\_r + s \quad , & if \left\lfloor \dfrac{V}{WL} \right\rfloor \ne \dfrac{V}{WL} \end{cases} \tag{3}$$

According to our design result, *D_w*=1, *D_r*=2, ***s*** =2 and *W*=4B, we get $T_{\mathrm{interleave}}=4$, and express *T* as:

$$T = \begin{cases} L + \dfrac{V}{L} + (\dfrac{V}{4} + 3) \quad , & if \left\lfloor \dfrac{V}{WL} \right\rfloor = \dfrac{V}{WL} \qquad (4.1) \\ L + 4\left\lfloor \dfrac{V}{4L} \right\rfloor + (\dfrac{V}{4} + 7) , & if \left\lfloor \dfrac{V}{WL} \right\rfloor \ne \dfrac{V}{WL} \qquad (4.2) \end{cases}$$

If the volume V can be divided exactly by C , the condition of (4.1) is fulfilled, we can get the minimum value of (4.1) as follows:

$$L_{optimal} = \sqrt{V}\ ,\ C_{optimal} = WL_{optimal} = 4\sqrt{V}\ ,\ T_{\min} = 2\sqrt{V} + \frac{V}{4} + 3 \qquad (5)$$

In fact, it is impossible to decide whether V could be divided exactly by C or not, because C is one of the unknown target parameters. However, it should be noted that the difference between formula (4.1) and (4.2) is negligible. So it is adaptable to use formula (4.1) and (5) to get the best value of $L_{optimal}$, $C_{optimal}$, and then $T_{\min}$..

6 Design Results and Performance Analysis

We make hardware design implementation for FCC-SDP based on the SMIC 0.13um CMOS technology, and use eight 256*32b single-port SRAM modules as the memory banks. FCC-SDP frequency reaches 350MHz, peak bandwidth is 43.75Gbps, read delay without blocking is 2 cycles, and synch delay is 2 cycles.

6.1 Performance Comparison and Analysis

We build the C simulator, QDSP-Sim for the heterogeneous multi-core DSP. QDSP-Sim is a cycle-accurate behavioral simulator, and can execute some programs that are partitioned and assigned manually. Based on the framework of QDSP-Sim, we construct three multi-core DSP models including different on-chip memory structures, FCC-SDP, shared L2 caches and private L2 caches, respectively. Their memory hierarchies and configurations are shown in Table 2.

Table 2. The memory hierarchies and configuration of three simulators

Hierarchy	**FCCSDP-Sim**	**SharedL2-Sim**	**PrivateL2-Sim**
SPM	FCC-SDP	Non	Non
L1 Cache	Private 4KB L1D and 4KB L1P, one cycle of hit delay		
L2 Cache	Private 64KB, 4-way set association, 5 cycles of hit access delay	Shared 256KB, cache consistency bases on directories, 4 parallel banks, 6 cycles of hit delay, 5 cycles of synch delay	Private 64KB, 4-way set association, 5 cycles of hit access delay, 20 cycles of DMA initialization time, the burst length is 4 words.
Off-chip	Private 256MB memories, 30 cycles of access delay		

We select four typical DSP benchmarks for our simulation experiments. Table 3 presents the detailed information and mapping methods of the four programs. The original data is stored in the off-chip memories for above three simulations. The intermediate results (i.e. shared data) are transferred respectively as follows:

- FCCSDP-Sim: by L/S instructions through FCC-SDP directly;
- SharedL2-Sim: by L/S instructions through the shared L2 cache;
- PrivateL2-Sim: by background DMA transmission through on-chip buses.

Table 3. Four benchmarks and their mapping used in the comparasion experiments

Benchmark	Notation	Shared	Mapping
FFT	Fast Fourier Transform for 1024 points	128B	non-interlace transform, pipelined process
JPEG-E	JPEG encoder, photo resolution 1024*768, Y:U:V=4:2:0	6KB	Partition photo by 8*8 macro blocks, pipelined encoding with the unit of 64 blocks
MP3-D	MP3 decoder, code rate 128Kbps, 40fps	16.2KB	Pipelined decoding with unit of 40 frames
H264-E	H.264 encoder, imagine resolution 352*288 (CIF, 4:2:0), 25fps	37.125KB	1/4 frame (99 macros, 16*16 pixels per macro) per DSP core, pipelined encoding

We compare the programs execution time of the three models. Fig. 5 shows the experiments results normalized with the time of shared L2 cache (SharedL2-Sim).

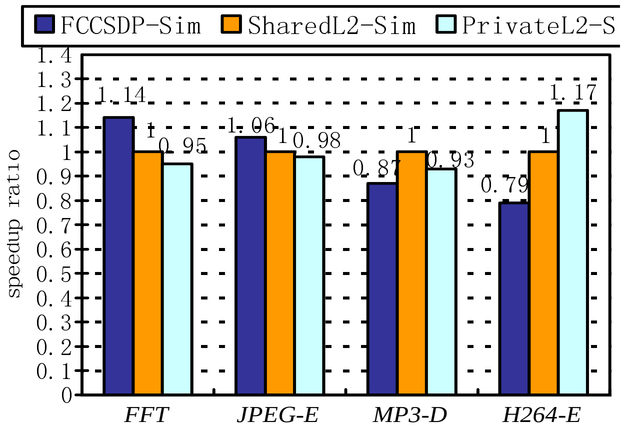


Fig. 5. Execution time comparison of three memory structures

The comparison results show that:

- For transmitting fine grain shared data between DSP cores, it has higher performance through FSS-SDP than through shared L2 cache and DMA transmission. The average speedup ratio of FCC-SDP in our results is 1.1 and 1.14, which benefits from the low access delay, low synch overhead and the Double-Bank Interleaving Access mode.

- For mass shared data transmission, the performance of FCC-SDP is lower than that of shared L2 caches and DMA transmission. The reason is that FCC-SDP transmission divides the shared data into too many little blocks, which increases the proportion of synch operations time consumption.
- As the overhead to maintain cache consistency increasing, the DMA background transmission exhibits better performance than shared L2 caches.

6.2 Experiments on Scalability

We have eliminated the access conflicts of FCC-SDP by replicated banks. In theory, the FCC-SDP structure eliminated access conflicts can scale well with the increase of DSP cores. However our experiments show different results. We define the shared access bandwidth *B* as the maximum bandwidth of FCC-SDP when all DSP cores access FCC-SDP simultaneously, as follows:

$$B = \frac{F}{D} \cdot W \cdot N \tag{6}$$

F is the maximum operation frequency of FCC-SDP, *D* is the access delay (if using pipelined access modes, *D* should be replaced by average access delay *D'*, *D'*<*D*), *W* is the width of access bus (here, *W*=32b), and *N* is the number of DSP cores. We make design implements for *N*=1, 2, ... , 8 respectively, and get the result of *B* according formula (6). Fig. 6 presents the experiments result of *B* vs. *N*.

As the increase of *N*, the overhead of interconnection links, control logics and crossbar between DSP and FCC-SDP will increase by order of O(N^2). The frequency of FCC-SDP becomes slower and the access delay gets longer, which cause the degrading of *B*. For the number of DSP cores more than eight, it is recommended that using four cores as a super-node to scale the multi-core DSP, which could employ FCC-SDP as a fast intra-super-node data-path to ensure the maximum fine-grain shared data transmission bandwidth, and employ NoCs (networks-on-chip) or other interconnection links as inter-super-node data-paths that have lower requirements in terms of communication delay and transmission bandwidth.

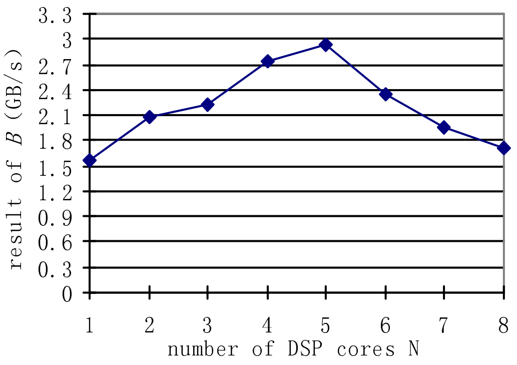


Fig. 6. The shared access bandwidth *B* vs. the number of DSP cores *N*

7 Conclusion

As yet, the “Memory Wall” problem in processors still is the bottleneck of system performance, and the problem is aggravated in multi-core processors. Scratch-pad memories can meet the need of embedded applications to some extend with its flexible configurations, fast access operations, convenient management and simple control logic. In development of embedded multi-core processors, it is necessary to exploit on-chip memory architectures, increase the effective bandwidth and improve the MPSoC performance.

References

1. Kandemir, M., Ramanujam, J., Choudhary, A.: Exploiting Shared Scratch Pad Memory Space in Embedded Multiprocessor Systems. In: Proceedings of the 39th Design Automation Conference (DAC 2002), New Orleans, USA, June 10-14, 2002 (2002)
2. Hu, W.: Shared Memory Architecture. Higher Education Press, Beijing, China (2001)
3. Verma, M., Petzold, K., Wehmeyer, L., et al.: Scratchpad Sharing Strategies for Multiprocess Embedded Systems: A First Approach. In: Proceeding of IEEE 2005 3rd Workshop on Embedded Systems for Real-Time Multimedia (ESTIMEDIA 2005), New York, USA, September 2005, IEEE Computer Society Press, Los Alamitos (2005)
4. Panda, P.R., Dutt, N.D., Nicolau, A.: Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications. In: Proceeding of 1997 European Design and Test Conference (ED&TC’97), Paris, France (1997)
5. Kandemir, M., Ramanujam, J., et al.: Dynamic Management of Scratch-Pad Memory Space. In: Proceeding of the 38th Design Automation Conference (DAC’01), June 18-22, 2001, Las Vegas, USA (2001)
6. Udayakumaran, S., Barua, R.: Compiler-Decided Dynamic Memory Allocation for ScratchPad Based Embedded Systems. In: Proceeding of the International Conference on Compilers, Architecture and Synthesis for Embedded System (CASES’03), October 30-November 2, 2003, San Jose, California, USA (2003)
7. Suhendra, V., Raghavan, C., Mitra, T.: Integrated Scratchpad Memory Optimization and Task Scheduling for MPSoC Architectures. In: Proceeding of International Conference on Complilers, Architecture and Synthesis for Embedded System (CASES’06), October 23–25, 2006, Seoul, Korea (2006)
8. Banakar, R., Steinke, S., Lee, B., et al.: Scratchpad Memory: A Design Alternative for Cache On chip memory in Embedded Systems. In: Proceeding of 10th International Symposium on Hardware/Software Co-design (CODES’02), May 6-8, 2002, Estes Park, Colorado (2002)
9. Issenin, I., Brockmeyer, E., Durinck, B., Dutt, N.: Multiprocessor System-on-Chip Data Reuse Analysis for Exploring Customized Memory Hierarchies. In: Proceeding of 43rd Design Automation Conference (DAC’06), July 24–28, 2006, San Francisco, California, USA (2006)
10. Mathew, B., Davis, A.: An Energy Efficient High Performance Scratch-pad Memory System. In: Proceeding of 2004 International Design Automation Conference (DAC’04), June 7-11, 2004, San Diego, USA (2004)

Exploiting Single-Usage for Effective Memory Management[⋆]

Thomas Piquet, Olivier Rochecouste, and André Seznec

IRISA, Campus Beaulieu, 35042 Rennes Cedex, France

Abstract. Efficient memory management is crucial when designing high performance processors. Upon a miss, the conventional operation mode of a cache hierarchy is to retrieve the missing block from lower levels and to store it into all hierarchy levels. It is however difficult to assert that storing the block into intermediate levels will be really useful. In particular, this is unnecessary if a cache block is accessed only once before getting evicted - i.e. a single-usage block. This paper is typically concerned with reducing the number of single-usage blocks. Our observations reveal that single-usage blocks are significant at runtime and especially in the lowest cache level. We show that using an address-based prediction mechanism is sufficient to identify this phenomenon. Two schemes are examined to remove pollution caused by single-usage blocks: a bypass scheme and a cache replacement policy. Our results show that leveraging single-usage pollution is beneficial to memory-intensive applications running on superscalar and multi-core architectures.

1 Introduction

Processor performance is strongly dependent on the memory hierarchy management. Access time to off-chip memory now represents several hundreds of cycles. In order to hide such huge latencies, modern processors feature a complete memory hierarchy composed of multiple cache levels with variable latencies. In addition, hardware prefetch mechanisms [1] are also often used to minimize the impact of main memory access time.

On a cache miss, the conventional memory hierarchy propagates the missing block from the lowest level in the memory hierarchy to the highest level, each cache level getting a copy of the block. When this strategy is used, the cache hierarchy acts as a set of more and more efficient filters that retains different memory accesses. This strategy is in general quite efficient, since in case of a subsequent miss on the same block in a lower memory hierarchy level, the block remains accessible.

However, this strategy does not take into account that blocks have very disparate usages across applications. In particular, in some cases, a block stored in

[⋆] This work was partially supported by an Intel research grant, an Intel research equipment donation and by the European Commission in the context of the SARC integrated project #27648 (FP6).

© Springer-Verlag Berlin Heidelberg 2007

the cache after a miss may not be accessed again before it is evicted. We call such a block, a *single-usage* block or a SU-block. Storing a SU-block in the cache may cause severe performance degradation as it could evict another block that could potentially be more useful. We refer to this phenomenon - storing SU-blocks in a cache - as *single-usage pollution* or SU-pollution.

Our first contribution in this paper is the characterization and analysis of the SU-pollution phenomenon. For a 2-level cache hierarchy, we show that most applications only exhibit a limited amount of SU-pollution in the L1 data cache, while some applications exhibit a high SU-pollution rate in lowest cache level. Our analysis also reveals that the single-usage property of a block is closely related to the memory instruction that triggers the L2 miss on this block.

Our second contribution is the proposal of a hardware mechanism for predicting single-usage pollution. Two schemes are presented to exploit SU-pollution: (1) a bypass scheme that prevents SU-blocks from entering the cache and (2) a SU-based cache replacement policy. Experiments show that our proposal is beneficial to both superscalar and multi-core architectures where the memory subsystem is a bottleneck.

The remainder of this paper is organized as follows. Section 2 quantifies single-usage pollution. In Section 3, we propose a single-usage prediction scheme and two techniques to exploit single-usage blocks. Section 4 presents our experimental results. Section 5 discusses the related work. Section 6 concludes this study.

2 Characterizing Single-Usage Pollution

We quantified the number of SU-blocks that are accessed at runtime for a subset of the SPEC2000 applications. Our data has been collected on a 4-way superscalar architecture featuring a 2-level cache hierarchy (32KB 4-way L1 data cache and 512KB 4-way L2 cache). A complete description of our baseline configuration is available in Section 4.1.

2.1 Quantifying SU-Pollution Within L1 and L2 Caches

We measured the number of dynamic accesses to SU-blocks within both the L1 and L2 caches. For a given cache level, *a cache block is defined as single-usage if it is accessed only once before getting evicted from this level.* Figure 1 reports the fraction of memory accesses that are single-usage at execution. We notice that SU-pollution is quite negligible in the highest cache level as only 6%, on average, of the dynamically accessed blocks are single-usage. The high usage behavior of cache blocks, mainly stems from the fact that data contained in L1 data cache exhibits high spatial and temporal localities. In our context, this means that attempting to reduce SU-pollution in the L1 cache would only have a small impact on overall performance. In contrast, the SU-pollution amount is much more significant in the lowest cache level. On average, 33% of memory accesses in the L2 cache are single-usage. We can even observe for some applications (*wupwise, swim, mgrid, applu, art, ammp*), mostly based on memory-intensive scientific

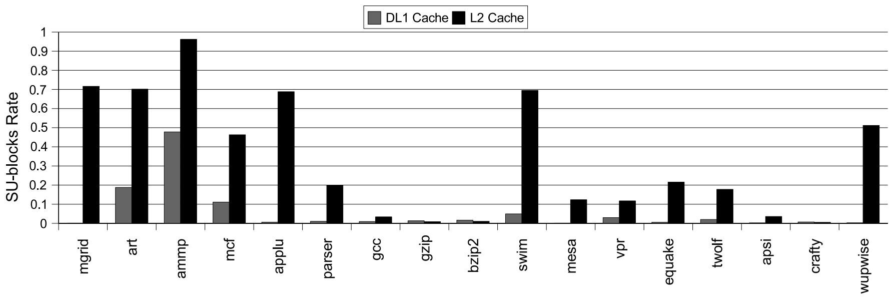


Fig. 1. Single-usage pollution within L1 and L2 caches

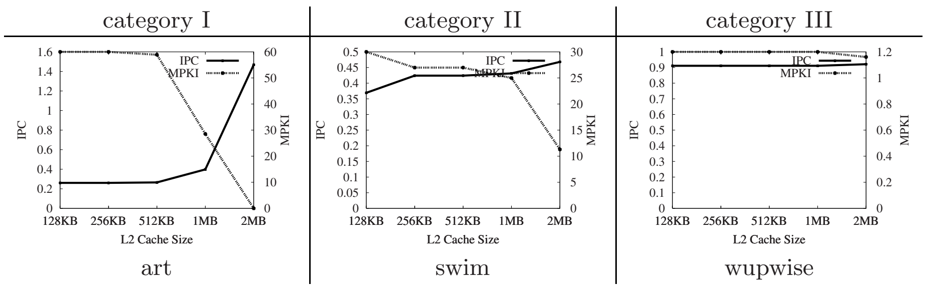


Fig. 2. Applications sensitivity to varying L2 cache size in terms of IPC and MPKI

kernels, that SU-blocks are quite prominent at runtime. Other applications such as *gzip, gcc, crafty, bzip2* depict a small SU-pollution level and are unlikely to benefit from our scheme.

2.2 Categorizing SPEC2000 Applications

Albeit some applications exhibit a high SU-pollution rate, minimizing this quantity would not necessarily translate into speed improvement, especially if the program performance is not dependent on the memory hierarchy behavior. We studied the applications sensitivity by varying the L2 cache size (from 128KB up to 2MB), using IPC and MPKI (miss per kilo-instruction) as metrics. Due to space constrains, we only report results for a few programs.

Figure 2 classifies the SPEC2000 applications into three distinct categories. The first category encompasses benchmarks in which performance and miss rate are very sensitive to increasing the L2 cache size. In our context, these applications are very likely to benefit from a scheme that reduces SU-pollution. For *art*, increasing L2 cache size from 512KB to 1MB has a significant impact on IPC and MPKI. In addition, since *art* exhibits a considerable SU-pollution amount, minimizing this quantity would also allow a substantial increase in the available cache space; and hence, in a potential performance improvement. The second category comprises memory-intensive workloads that do not benefit from resizing the L2 cache from a performance viewpoint. Nonetheless, we observe that doubling the L2 cache size does still lead to a noticeable reduction in terms of miss rate. This would therefore allow to reduce the main memory bandwidth usage, which could be used to trigger prefetch requests instead. The last category gathers applications in which performance and miss-rate are not dependent on cache size. For these applications, attempting to reduce SU-pollution will have only a marginal impact on performance, or worse, in case of a SU misprediction, this could even result a performance loss.

3 Predicting and Exploiting Single-Usage Blocks

Minimizing the pollution due to SU-blocks can help improve the whole memory hierarchy behavior as well as the processor performance. In this section, we first show that the single-usage property of a dynamic L2 cache block is closely related to the instruction that triggers the L2 cache miss. This observation makes a PC based block-usage prediction mechanism viable. We also describe how a stride prefetcher could be adapted at a minimum extra hardware cost for predicting cache blocks usage. To decrease SU-pollution, we propose two distinct schemes: (1) a bypass solution and (2) a SU-based cache replacement policy.

3.1 Single-Usage Property is Associated with the Instruction

In order to speculate over the cache block usage property, one can consider the cache block address itself and quantify its usage over time. Johnson et al. used this approach in [2]. However, we show below that a cache block usage property is tied to the program instruction that triggers the memory access.

Quantifying Single-Usage I-Sequences. We refer to the sequence of L2 cache accesses initiated by a single program instruction as *an I-sequence*. Let us also define a single-usage I-sequence as a I-sequence for which the amount of SU-blocks exceeds 95%. Note that the SU I-sequences property depends on the memory hierarchy configuration. On average, we found that over 90% of SU-blocks are referenced by SU I-sequences across SPEC2000 benchmarks. Hence, a mechanism capable of detecing SU I-sequences at runtime could help decrease SU-pollution.

3.2 Hardware Support to Predict Block Usage

The block-usage (BU) predictor (see Figure 3), mainly consists of a block-usage prediction table and on two extra tags associated with each L2 cache line. Each entry in the block-usage prediction table consists of two fields: 1) the instruction address (IA) and 2) a saturated single-usage detection (SUD) counter. Two tags attached to a L2 cache line are the SU tag, a single bit that records whether or not the block has been re-accessed after being stored into the L2 cache, and the instruction address tag (or IA tag) that records the address of the instruction

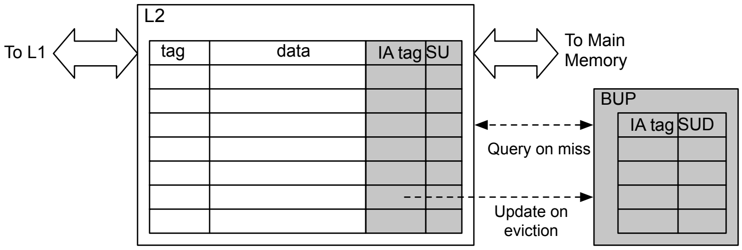


Fig. 3. Block-usage predictor (BUP)

generating the miss. The predictor operates in two main phases : 1) query and 2) update.

Query. On a miss on the L2 cache, a query is sent to the BU predictor. The address of the instruction that incurred the miss is used as an index. If an entry matches, the predictor delivers a SU or non-SU verdict depending on SUD value. A SU verdict is only delivered upon a saturated SUD counter state, else a non-SU verdict is returned.

Update. The BU table is updated whenever a block is evicted from L2 cache. The IA tag of the evicted block is used to get the corresponding I-sequence in the BU table. The SUD counter associated with the BU table entry is updated according to the SU tag of the block. If the tag indicates that the block is single usage, the counter is incremented, otherwise the counter is reset to zero.

The BU predictor can suffer from two misprediction types : 1) the block is predicted as non-SU, but is single usage and 2) the block is predicted as single usage but might have been accessed several times if it was stored into L2 cache - this is referred to as a SU misprediction. Due to the potential performance loss induced by a SU misprediction, our BU predictor favors accuracy over coverage. This is done through delivering SU verdicts only upon a saturated SUD counter state and by resetting counters on non-SU updates.

Adapting a Stride Prefetcher for Block-Usage Prediction. One can easily extend a stride prefetcher [1] for block-usage prediction as this mechanism also uses the instruction address to initiate prefetch requests. This would enable to mitigate the hardware overhead due to the BU predictor. To do so, each entry in the stride prefetcher table has to be augmented with a SUD counter. The main overhead induced by the BU predictor is the additional tags on the L2 cache. In our experiments, each L2 cache line is augmented with a 1-bit SU tag and a 13-bit partial IA tag (9-bit are actually used to index the 512-entry stride prefetcher table, the purpose of the remaining 4-bit is to reduce aliasing). For the 128-byte cache blocks considered in the paper, these extra tags account for 2% of the cache storage budget.

3.3 Reducing SU-Pollution

We propose two distinct techniques to exploit SU-pollution reduction in the L2 cache: (1) a bypass scheme and (2) a SU-based cache replacement policy.

Bypass Scheme. In order to prevent SU-pollution, we suggest to directly forward missing SU-blocks - identified by means of the BU predictor - from memory to the L1 data cache; thus bypassing the L2 cache to enable more useful data to remain cached. Note that non-SU blocks are still processed in a standard way. Performing bypassing could however have a detrimental effect on the predictor accuracy. Once an I-sequence is marked as single-usage, its corresponding SUD counter could remain saturated forever. To overcome this issue, we propose to re-inject a SU-block into the L2 cache once in a while. This allows to update the SUD counter. If the I-sequence behavior changes, the SUD counter will be reset. The decision of re-injecting a SU-block is taken using a low probability.

SU-Based Replacement Policy. Another way to reduce SU-pollution is to use the BU predictor for a cache replacement purpose. Let us suppose a set-associative cache featuring a LRU replacement policy. Our proposal is to augment the LRU algorithm with block usage information. Upon selecting a block for eviction, our technique favors the replacement of least-recently-used blocks marked as single-usage instead of solely using the recency information. If there are no SU-block in the current set, the LRU block is selected. The architectural support needed for this scheme consists of extending each L2 cache block with a single bit that reflects whether or not this block is single-usage. This prediction bit is updated each time a cache block is loaded from memory. As for the bypass scheme, however, SUD counters could remain saturated. To avoid this scenario, we take an arbitrary decision using a low probability to decide if we should select the LRU-block as a victim instead of the SU-block.

4 Evaluation

This sections evaluates the performance of the BU predictor in terms of accuracy and coverage. It also examines the impact on performance, miss-rate and memory traffic induced by the SU-based cache replacement policy and the bypass scheme.

4.1 Experimental Setup

Our experiments were performed on SESC, an execution-driven simulator developed by [3]. Our baseline processor is a 4-way out-of-order superscalar architecture. Table 1 summarizes the configuration we used as a reference. Our memory subsystem models a 512-entry stride prefetcher [1] that is coupled with a 32-entry prefetch buffer [4, 5] to filter pollution related to aggressive prefetching.

Benchmarks. We evaluate our proposal on a subset of SPEC2000 benchmarks that run on SESC: *wupwise, swim, mgrid, applu, mesa, art, equake, ammp, apsi,*

Table 1. Simulated machine parameters

Parameter	Configuration
Decode / Issue / width	4
Retire width	5
ROB size	36 Issue + 32 entries
LSQ size	20 Issue + 32 entries
Branch predictor	O-GEHL [6], 64-Kbit, 6-cycle mispred. penalty
L1 inst.	64kB, direct-map, 128B/block, LRU, 1-cycle
L1 data	32kB, 4-way, 128B/block, LRU, 1-cycle
L2 unified	512kB, 4-way, 128B/block, LRU, 11-cycle
Main Memory latency	500-cycle

Table 2. BU predictor coverage and accuracy (512-entry table + 3-bit SUD counters)

	coverage	accuracy	SU-rate	#cache accesses (*M)		coverage	accuracy	SU-rate	#cache accesses (*M)
mgrid	96.4%	99.53%	71.64%	4.96	swim	87.74%	99.64%	69.52%	31.3
art	83.72%	99.8%	70.18%	69.91	mesa	88.23%	99.98%	12.35%	1.54
ammp	99.63%	99.98%	96.23%	86.76	vpr	1.39%	80.2%	11.77%	19.45
mcf	71.71%	98.16%	46.31%	88.41	equake	59.16%	99.77%	21.58%	4.57
applu	98.01%	99.84%	68.84%	6.34	twolf	0.81%	74.08%	17.79%	25.43
parser	4.81%	81.65%	19.87%	9.45	apsi	67.19%	98.06%	3.6%	2.58
gcc	49.04%	97.92%	3.42%	13.47	crafty	0.02%	91.67%	0.54%	11.78
gzip	67.05%	97.2%	0.87%	7.85	wupwise	97.65%	99.74%	51.19%	2.22
bzip2	41.61%	89.26%	1.05%	9.75					

gzip, vpr, gcc, mcf, crafty, parser, bzip2, twolf. All applications were compiled for the MIPS ISA with the `-O3` optimization flag enabled. We used the reference data as an input. The first billion instructions were skipped and the next billion instructions were simulated.

4.2 Block-Usage Predictor Accuracy and Coverage

We define the BU predictor coverage as the fraction of the number of SU-blocks that are correctly predicted. The BU predictor accuracy as the fraction of SU verdicts that are correct. Table 2 reports the coverage and accuracy of a 512-entry BU predictor table featuring 3-bit SUD counters. Overall, the BU predictor provides high accuracy on most benchmarks. The rationale is that we deliver SU verdicts exclusively on a counter saturated state while resetting the count value on non-SU verdicts. Although this slightly impairs the predictor coverage, we observe that we still identify a large fraction of SU-blocks for memory-intensive applications. For other applications such as *crafty, vpr, twolf*, our predictor is quite inefficient. This is due to the fact that these applications only exhibit a small SU-pollution rate as mentioned in Section 2.1.

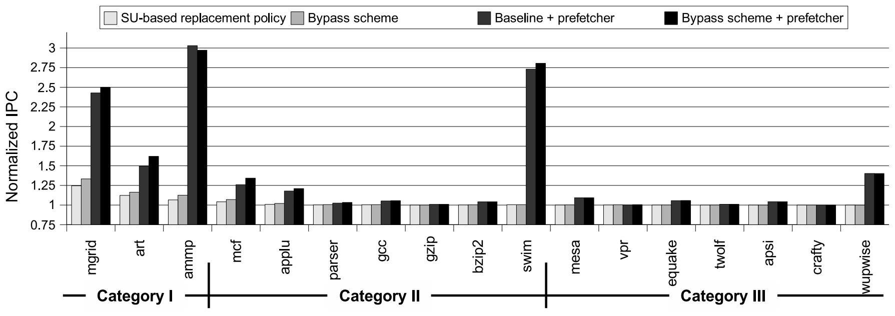


Fig. 4. IPC normalized to baseline for our different schemes

Varying BU Predictor Parameters. We varied the main BU predictor parameters (number of predictor entries and SUD counter width) to study their influence on coverage and accuracy. Increasing the number of entries from 128 to 1024 has a positive, but small impact on coverage. Increasing the SUD counter width decreases the coverage of the BU predictor but increases the SU-verdict accuracy. Our results indicate that a suitable trade-off between the predictor efficiency and its hardware complexity is to use a 512-entry predictor table comprised of 3-bit saturating counters.

4.3 Impact on Performance, Miss-Rate and Memory Traffic

Figure 4, Figure 5 and Figure 6 compare our cache management policies using three metrics, namely the IPC, the L2 cache miss rate (in MPKI) and the bus traffic (number of accesses) between L2 cache and main memory. These results are normalized to our baseline architecture described in Table 1. The first bar corresponds to our SU-based replacement policy described in Section 3.3. The second bar represents the bypass scheme (see Section 3.3). The following bar is the baseline architecture enhanced with a stride prefetcher. The last bar corresponds to the stride prefetching scheme adapted for block-usage prediction.

Figure 4 points out that our proposal performs well with workloads from the first category (see Section 2.2) whereas applications from other categories show little or no performance gains. *mgrid* performs well on both schemes by achieving a speed-up close to 30% along with a noticeable decrease in the L2 cache miss-rate. This is consistent with our analysis as we observed that *mgrid* performance is very sensitive to adapting the L2 cache size - especially from 512KB to 1MB.

When a performance gain is observed, the bypass scheme usually performs better than the SU-based cache replacement policy. This is somewhat coherent as bypassing SU-blocks allows existing multi-usage data to remain cached in L2. In contrast, with the SU-based replacement policy, a few SU-blocks can still reside in L2 cache; hence the lower performance gain.

For most applications, using a stride prefetcher for block-usage prediction is beneficial to performance and miss rate. Due to a reduced memory traffic - see

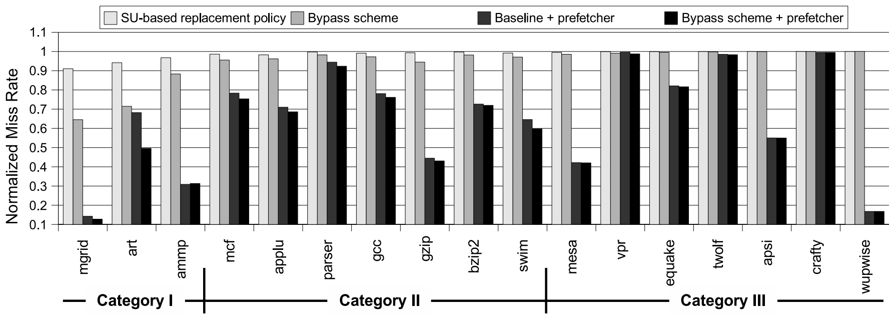


Fig. 5. Normalized L2 cache miss-rate

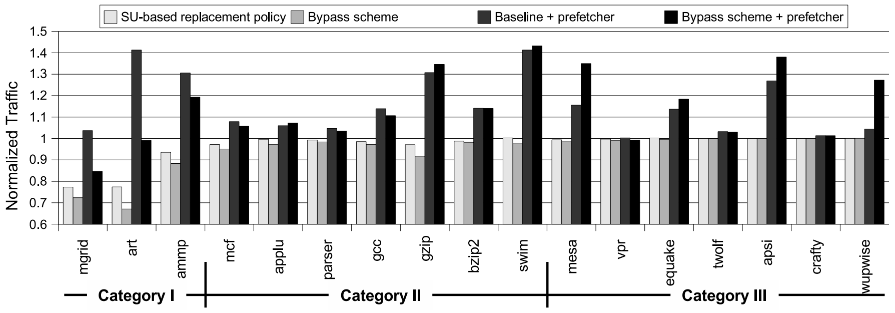


Fig. 6. Normalized memory traffic between L2 and main memory

Figure 6 - the number of prefetching opportunities is accordingly increased; hence an extra performance gain as compared to a basic bypass scheme. On some applications such as *gzip*, no performance gain is observed as this program has a small miss rate. Note that the performance improvement obtained with the prefetcher-based BU predictor is not as significant as that of a basic BU predictor scheme - e.g. see *mgrid*. This stems from the fact that the usage of prefetching overrides part of the benefits achieved by our scheme.

Our proposal slightly degrades performance of *ammp* when used with a stride prefetcher - from a 0.0732 IPC to 0.0718. This is due to the management of the memory bus. Prefetches are initiated only when the bus is free. Bypassing the L2 cache on write-backs of SU-block tends to create a burst of traffic that could prevent prefetches. When data are first stored in the L2 cache, the write-back traffic is smoothened, creating new opportunities for triggering prefetch requests.

4.4 Exploiting SU-Pollution Reduction in Multi-core Systems

Managing the memory hierarchy is a crucial issue in multi-core architectures where processing cores often share the lowest cache level. We examined the potential gains that could be achieved by our bypass scheme in this context. To do

Table 3. Weighted IPC, L2 cache miss-rate and SU-pollution for a dual-core system

	2-core - 1MB L2			2-core - 2MB L2			2-core - 1MB L2 + bypass		
	W_{IPC}	Miss Rate	SU rate	W_{IPC}	Miss Rate	SU rate	W_{IPC}	Miss Rate	SU rate
wupwise	1.00	0.50	98.52	1.00	0.50	97.20	1.00	0.50	4.14
mgrid	0.87	3.48	44.99	1.00	2.96	35.12	1.00	2.98	3.51
applu	1.00	4.47	90.90	1.00	4.47	90.83	1.00	4.47	3.05
mcf	0.98	51.39	68.42	1.08	45.95	57.51	1.21	43.17	18.78
mgrid	0.71	4.54	72.59	1.00	2.97	35.38	0.94	3.53	4.59
ammp	0.57	9.34	4.37	1.00	0.19	0.07	0.93	1.14	0.17
art	0.94	35.23	44.62	2.73	0.26	0.11	1.09	23.81	12.35
gzip	0.79	1.68	11.08	0.99	0.21	1.65	0.81	1.55	3.93

so, we modeled with SESC a dual-core system that features private 32KB 4-way L1 data caches and a shared 4-way 1MB L2 cache. A 1k-entry BU predictor table is considered. We mixed together applications from distinct categories (see Section 2.2) to study the performance impact on our scheme. As a performance guide, we use the weighted speed-up metric [7, 8]. For each benchmark, we simulated 250M instructions. If a benchmark completes 250M instructions before the other, we keep on executing the finished benchmark till the second benchmark finishes its processing.

Results. Table 3 reports the weighted IPC, the L2 cache miss-rate and the associated SU-pollution rate for different memory configurations of the baseline dual-core system. For each programs mix, we report the contribution of individual programs for the considered metrics. For instance, running *wupwise-mgrid* on the baseline CMP shows that *wupwise* does not suffer from cache sharing ($W_{IPC} = 1$) while *mgrid* ($W_{IPC} < 1$) does. Table 3 shows that executing our mixed applications with a larger L2 cache often improves the considered metrics. Overall, our bypass scheme applied to a multi-core architecture provides noticeable performance gains. It does even outperform a CMP system featuring a twice as large L2 cache when executing *applu-mcf*. While *applu* by itself does not benefit from reducing SU-pollution, it does however makes room for *mcf*, thus allowing substantial performance gain on this latter application. The same phenomenon occurs with *mgrid/ammp*. Reducing *mgrid* SU-pollution essentially reduces *ammp* miss rate.

5 Related Work

Tyson et al. [9] observed that, on many applications only a few load instructions are responsible for the majority of data cache misses. They proposed a scheme to decide whether or not a load instruction should allocate data in the L1 cache. The authors suggest using PC-indexed counters that are incremented on a miss and decremented on a hit. In practice, a load instruction is classified as a "to be bypassed" if in general it is the first to touch a memory block and if further instances of the same load do not touch back the same block in the near future. This scheme is able to capture instructions exhibiting no spatial locality on the L1 cache, such as loads exhibiting a stride longer than a cache block. However it is not able to capture future reuse of the cache block by other loads or writes. This may sometimes lead to dramatically poor behavior, particularly on optimized code. For example, on a streaming application, unrolling a loop with an unrolling factor larger then the cache line size may push the hardware to classify the first access to each data in a cache block as a "bypass access".

Dybdahl et al. studied block bypassing in the last cache level in [10]. They extended the dynamic scheme for the L1 cache proposed by Tyson [9] to the lowest cache level. They noticed that this extension sometimes leads to a severe performance loss. They proposed a new hardware scheme to address this issue. The result is mitigated: performance losses are reduced on some applications, performance benefits are also reduced on other applications. The hardware cost of their scheme is relatively high, since each block in the last-level cache is augmented with voluminous information (shadow address tag, instruction address, status). Moreover the management algorithm is quite complex.

Chi et al. [11] proposed a software scheme to address single-usage cache pollution. The compiler determines for each memory reference its *cachability*. Since an architecture with a single cache level is considered, on a reference marked as *not cachable*, the data is not stored in the L1 cache. The main limitation of this software solution is that it does not take into account the spatial locality within a cache block.

Rivers and Davidson [12] proposed a hardware mechanism to capture the temporality of a data block. Data blocks are classified as temporal or non temporal (NT). A block is classified as NT if none of its words is re-referenced before its eviction. A NT bit is added to each block in the first and second levels of the cache. The main memory does not have the NT bit, therefore once a block is evicted from the L2 cache, the information is lost. In contrast to this proposal, we associate the single-usage property to memory access instructions rather than to cache blocks, and we address the L2 cache.

Wong and Baer [13] described a cache replacement policy enhanced with temporal locality information to guide block replacement. Instead of systematically evicting LRU blocks, their scheme favors replacing non-temporal blocks instead. The temporal information is obtained through profiling or by means of a hardware predictor.

6 Conclusion

This paper proposes to exploit reduction in single-usage cache pollution for a better memory hierarchy management. We observed that the single-usage property of a cache block is very tied to the load/store instruction that causes a cache miss (on this block). Hence, we suggest using a PC-based hardware predictor to uncover SU-blocks at runtime. Our experiments show that our predictor provides high coverage and accuracy on most programs. We evaluate two schemes to reduce SU-pollution: (1) a bypass technique and (2) a SU-based cache replacement policy. Our results point out that using either technique is beneficial to a superscalar architecture. Extra gain is further observed when adapting a stride prefetcher for block-usage prediction - while mitigating the storage overhead due to our predictor. Our proposal is also evaluated in a multi-core environment using multi-programmed workloads. Exploring the benefits on multi-threaded programs is part of our future work.

References

[1] Fu, J.W.C., Patel, J.H., Janssens, B.L.: Stride directed prefetching in scalar processors. In: Proceedings of the 25th annual international symposium on Microarchitecture (1992)

[2] Johnson, T.L., Connors, D.A., Merten, M.C., mei W. Hwu, W.: Run-time cache bypassing. IEEE Trans. Comput. 48(12) (1999)

[3] Renau, J., Fraguela, B., Tuck, J., Liu, W., Prvulovic, M., Ceze, L., Sarangi, S., Sack, P., Strauss, K., Montesinos, P.: SESC simulator (2005), `http://sesc.sourceforge.net`

[4] Jouppi, N.P.: Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In: Proceedings of the 17th annual international symposium on Computer Architecture (1990)

[5] Chen, W.Y., Mahlke, S.A., Chang, P.P., mei W. Hwu, W.: Data access microarchitectures for superscalar processors with compiler-assisted data prefetching. In: Proceedings of the 24th annual international symposium on Microarchitecture (1991)

[6] Seznec, A.: Analysis of the o-geometric history length branch predictor. In: Proceedings of the 32nd Annual International Symposium on Computer Architecture (2005)

[7] Snavely, A., Tullsen, D.M., Voelker, G.: Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In: Proceedings of the 2002 international conference on Measurement and modeling of computer systems (2002)

[8] Hsu, L.R., Reinhardt, S.K., Iyer, R., Makineni, S.: Communist, utilitarian, and capitalist cache policies on cmps: caches as a shared resource. In: Proceedings of the 15th international conference on Parallel architectures and compilation techniques (2006)

[9] Tyson, G., Farrens, M., Matthews, J., Pleszkun, A.R.: A modified approach to data cache management. In: Proceedings of the 28th annual international symposium on Microarchitecture (1995)

[10] Dybdahl, H., Stenström, P.: Enhancing last-level cache performance by block bypassing and early miss determination. In: Asia-Pacific Computer Systems Architecture Conference (2006)

[11] Chi, C. H., Dietz, H.: Improving cache performance by selective cache bypass. In: 22nd Hawaii International Conference on System Sciences (1989)

[12] Rivers, J., Davidson, E.: Reducing conflicts in direct-mapped caches with a temporality-based design. icpp 01 (1996)

[13] Wong, W.A., Baer, J.L.: Modified lru policies for improving second-level cache behavior. In: HPCA (2000)

An Alternative Organization of Defect Map for Defect-Resilient Embedded On-Chip Memories

Kang Yi[1], Shih-Yang Cheng[2], Young-Hwan Park[2], Fadi Kurdahi[2], and Ahmed Eltawil[2]

[1] School of Computer Sceince and Electronic Engineering, Handong Global University, Pohang, Korea
yk@handong.edu

[2] Department of EECS, University of California, Irvine, CA 92697-265
{shihyanc,younghwp,kurdahi,aeltawil}@uci.edu

Abstract. In this paper, we propose the low power and low area defect map organization for the defect-resilient embedded memory system for multimedia SOCs. Existing approach to build defect map of embedded memories is based on the CAM (Content Addressable Memory) organization. But, it consumes too much power and relatively large chip area. It may be serious problem in the near future for very deep submicron technologies. Therefore, we propose the SRAM-based defect map organization to reduce both the power consumption and chip area. We also develop new defect map access algorithm to minimize the number of defect map access operations to save power. Our estimation results show the new scheme based on SRAM defect map organization consumes only 1/4 times of power at BER=1.0% compared with the power overhead by the existing approach.

Keywords: Embedded memory Yield, Defect Map, Memory Error Resilient Design, Video error concealment.

1 Introduction

The memory hungry application is becoming the dominant portion of the SOC market because of highly increasing demands on multimedia applications. According to the 2001 International Technology Roadmap for Semiconductor, the embedded memories are going to occupy from 54% to 94% of silicon real estate by year 2014 [1,2] as shown in Figure 1. In addition, the ever-shrinking geometry of semiconductor devices is pushing the memory parts into a single chip integrated with core logic parts because of many practical benefits such as manufacturing cost reduction, performance enhancements, and much more power saving.

However, it is well known problem that the large embedded on-chip memory with very deep submicron technology will be suffering from high defect density resulting in low SOC chip yield and high production cost.

Even though there are several existing approaches for this memory recovering from defects problem, these approaches with redundancies require too much area overhead for the memory defect density in the near future. In [3] we can see the

© Springer-Verlag Berlin Heidelberg 2007

prediction of the defect densities that are 1-2 orders or magnitude higher than today's defect density. In [4] the existing approach with redundancy scheme to address 0.1% defect density shows a huge cost of 70% area overhead.

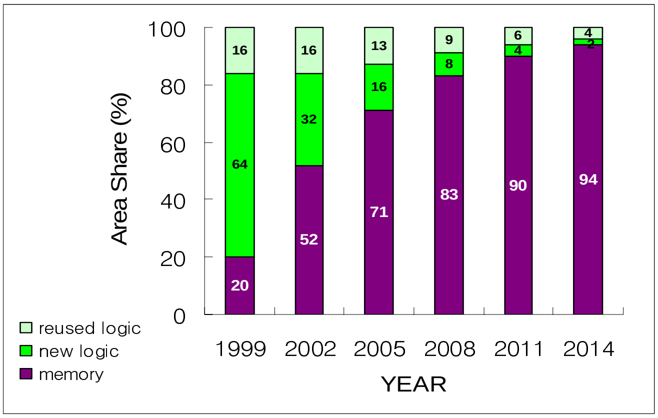


Fig. 1. Area of memory portion in SoC design trend

An innovative approach to this embedded memory data recovering problem with highly defect density for the near future technology was proposed by [5] and [6] focusing on the multimedia application. The new approach observes that multimedia applications have the information redundancy in themselves like spatial and/or temporal locality. The new approach is based on the defect map which records all the defect memory cell location and post processing to recover corrupted data from defective data location. But, the overhead of the new approach is defect map power consumption.

Therefore, in this paper we propose an alternative defect map organization that has the same functionality as previously proposed system while it saves much of the power consumption. The key idea is using of SRAM instead of CAM (Content Addressable Memory) and reduction defect map read operation frequency drastically with new memory lookup strategy.

2 Summary of the Previous Work

2.1 Overview of Filtering Scheme with Defect Map

By utilizing the redundant information in multimedia application data themselves, [6] developed an image data recovery method from corrupted memory targeting at H.264 decoder system. According to the previous work from [6] they achieved high visual quality as well as high PSNR values even with simple image filters if the defect pixel location is known by defect map. The works in [6] recovers corrupted moving pictures with memory defects up to 1.0% bit defect density which is quite higher density than necessary even in the foreseeable future technology. Figure 2 shows the system block diagram of the approach.

The defect map in this system is constructed by built-in-self-testing process at the time of system power up booting phase. Once the defect map is constructed, the defect map lookup is requested for every read operation for the pixel data stored in a frame buffers to see if the location is defect or not. Whenever an H.264 decoder tries to read the defect location, image filter is applied to get the properly estimated data for the pixel. In this filtering process, neighboring pixel data in the frame buffer (DPB memory) may be required, too. Figure 3 shows the visual quality of corrupted and recovered image by the filtering with defect map approach. The recovery result is almost very good.

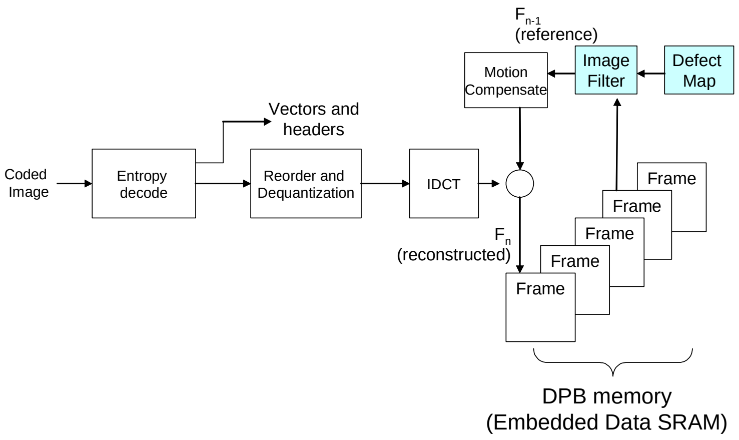
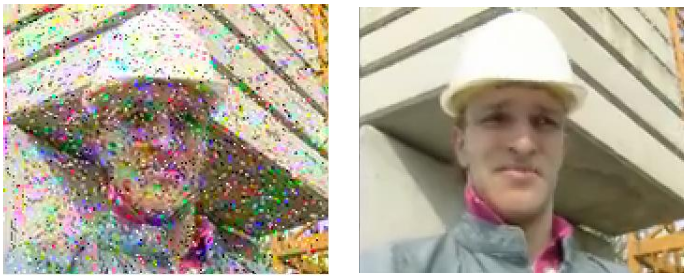


Fig. 2. The overall architecture for defect-resilient multimedia data memory with H.264 decoder system



(a) corrupted image by defects (b)recovered image with filter

Fig. 3. Comparison of Images before and after recovery by [6] at BER=1.0%

2.2 The Problem with Filtering Scheme with CAM-Based Defectmap

Figure 4 shows the CAM defect map organization. It emphasize that defect map reference is required for every pixel value reading. The problem with the defect map is that at higher defect rates the defect map power consumption increases and the defect map area overhead is no more negligible. Figure 5 shows this problem with power overhead of filtering scheme for different defect densities in memory bits. The problem stems from the fact defect map size is proportional to the pixel error rates. The relationship between pixel error rates (PER) and bit error rates (BER) are shown equation (1).

$$PER = 1 - (1 - BER)^{PIXEL\_DEPTH} \quad (1)$$

The PIXEL_DEPTH in equation (1) is the number of bits for each pixel which is usually 8. The power consumption of the defect map is proportional to the size of defect map because of CAM nature.

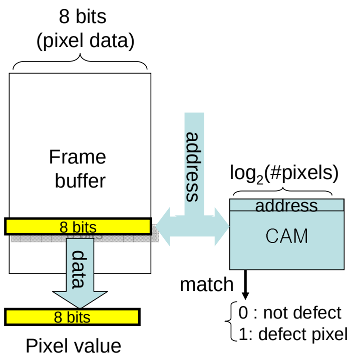


Fig. 4. CAM-based defect map organization

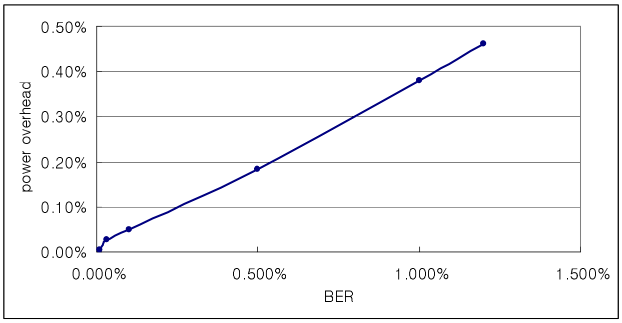


Fig. 5. Power overhead by filtering scheme with CAM-based defect map organization

This experimental power consumption data is based on the numbers in [7] and assumes the implementation with 90 nm technology and VGA sized image with 5 reference frames. In the Figure 6 we analyzed the source of power consumption. We observe that defect map power consumption accounts for more than 80% of the total filtering scheme power consumption at higher defect densities. Note that the BER=0.01% is high enough to meet the current technology requirements but, it is expected the BER may be higher than 0.1% under the coming very deep submicron technology. Therefore, in order to reduce the power overhead of filtering scheme for the foreseeable future technology we have to find a way to reduce the defect map power consumption. In this paper, we propose the use of SRAM for the defect map construction rather than CAM-based defect map organization.

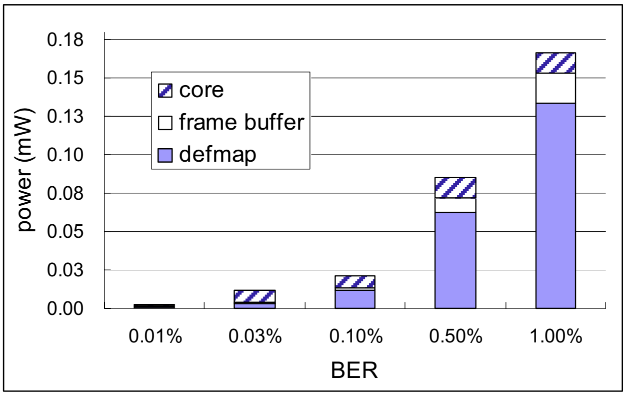


Fig. 6. Power consumption share by filtering scheme with CAM- based defect map organization

3 Alternative Defect Maps

3.1 The Idea of SRAM-Based Defect Map Organization

We compare the old defect map with our new SRAM-based one in Figure 7. In this new defect map scheme, we use SRAM instead of CAM to save power. We store the addresses of defective pixels in the SRAM. And, we look up the defect map for every pixel data reading operation to find if there is any entry in the defect map that has the same given address. But, the problem with the SRAM-based approach is that we need to search all the SRAM memory to find a specific SRAM address having specific frame buffer address. What makes it worse is that SRAM only performs the search operation one by one manner while CAM performs a parallel search with contents. To avoid the exhaustive search in the defect map we assume the following two things.

(1) The addresses of defect pixels stored in the SRAM are all sorted in an ascending order.
(2) The frame buffer read operation for pixel value acquisition is performed by an address issued in an ascending order.

The above two assumptions can be realized by (1) well organizing the memory address ordering and (2) by address sorting at the time of defect map construction which occurs only one time per system power up. In this scheme we also use special registers called “defectmap_pointer” register and “defect_pixel_addr” register. The defectmap_pointer register points to the defect map location that contains the address of the frame buffer where the next defective pixel is located. Defect_pixel_addr represent the defect map content currently pointed by defectmap_pointer. Under the two assumptions above, we develop an SRAM-based defect map look up algorithm as shown in Figure 8.

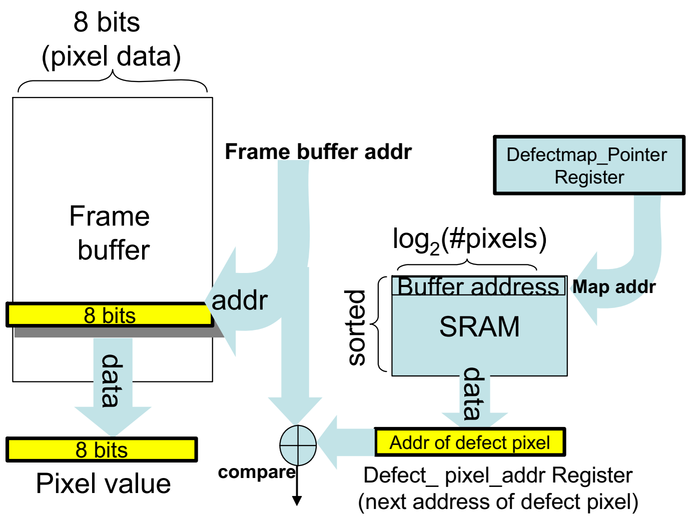


Fig. 7. Basic SRAM-based defect map organization

```
function DefMap_LookUp (frame_buffer_address : address)
  if(*defectmap_pointer == UNINITIALIZED) then
         defectmap_pointer = 0;
         defect_pixel_addr = *defectmap_pointer;
  end
  if (frame_buffer_address < defect_pixel_addr) then
         return DEFECT_NOT_FOUND;
  else if (frame_buffer_address == defect_pixel_addr) then
         defectmap_pointer = defectmap_pointer + 1 ;
         if (defectmap_pointer > MAX_ADDR) then
             defectmap_pointer = 0;
         end
         defect_pixel_addr = *defectmap_pointer ;
         return DEFECT_FOUND;
  end ;
  else begin
      return DEFECT_NOT_FOUND
  end
```

Fig. 8. SRAM-based Defect map lookup algorithm

The first line of the algorithm describes the defectmap pointer register and defect_pixel_addr initialization procedure. The following “if statement” determines whether the given frame buffer address is the defect location or not by comparing the given address with the defectmap output register. If both matches it means given address is defect location and increase the defect pointer and read out the content from the defect map which is the next defect pixel address in the frame buffer.

With this algorithm, we achieve the sequential read of defect map per frame. That means the core H.264 decoder reads defect map only as many time as the number of defective pixels in the frame buffer. Therefore, we can save the defectmap power consumption drastically with SRAM-based defect map organization due to (1) the less power consumption by SRAM circuit than CAM circuit per access and (2) the less number of defect map access by the new defect map access algorithm.

3.2 A Consideration for Image Filters: Enhanced SRAM-Based Organization

We still have the problem with the SRAM-based defect map organization proposed in previous subsection when we care about the image filters. Some filters developed in [6] require the surrounding pixel values neighboring the defect pixel. And, some of them also need to find whether each of neighboring pixels is in defect location or not. In order to provide this neighboring pixel defectiveness information we added more bits to each of the entries in the SRAM-based defect map. We show the modified organization in Figure 9.

The enhanced organization allocates more space of B bits slot per pixel value for the neighborhood pixel information of defectiveness. Each of bit in the extra space The B is from 0 to 8 where 0 means the image filter does not care about the defectiveness of neighboring pixels and 8 means the image filter needs every

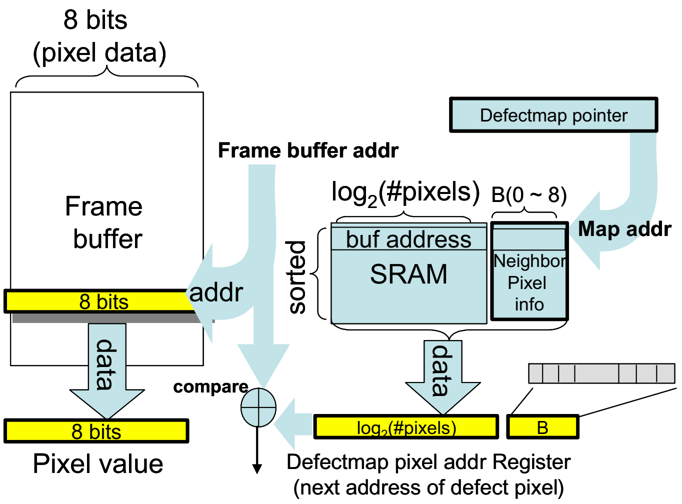


Fig. 9. Enhanced SRAM -based defect map organization

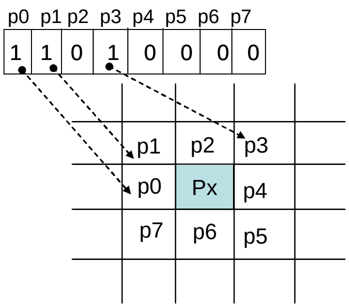


Fig. 10. Representation of defectiveness information of neighbor pixels around the defect pixel

neighboring pixel defectiveness information. Figure 10 demonstrates the meaning of each bit in the extra bits where B=8. If the left-most bit (1st bit), the 2nd, and the 4th bits in the extra field are '1's, the pixel p0, p1 and, p3 are defect pixels.

3.3 Simpler Defectmap Organization: Flag Bit Approach

Sometimes, simple strategy is the best strategy under a special condition. We propose another alternative defect map organization named "Flag" approach which can be effective when defect density is too high for separate defect map memory. Figure 11 shows the "Flag-based" defect map organization.

This approach adds one flag-bit for each pixel value component. Each of the flag-bit whose value is '1' means the corresponding entry of the frame buffer has one or more defect bits per entry of each frame buffer location. Its area overhead is 1/8 =12.5% regardless of defect density. But, it has the benefits over SRAM-based method : (1) very simple organization, (2) low power consumption comparable to enhanced SRAM-based approach, (3) low area overhead at very high defect density (BER >= 0.5%).

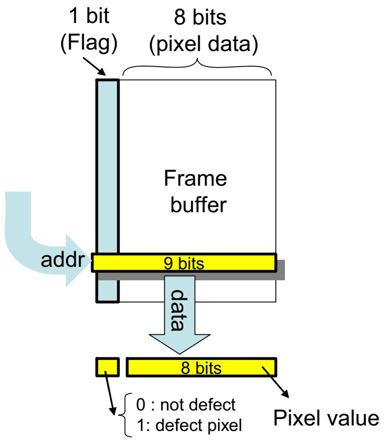


Fig. 11. Flag-based defect map organization

4 Quantitative Evaluation

4.1 Memory Area Overhead Comparison

The following equation (2) through (4) is for the defectmap area overhead for CAM-based defect map and equation (5) through (6) is for the enhanced SRAM-based defect map with B more extra bits per entry

$$Area_{FrameBuffer} = 8 \times Num\_Pixels \times Area_{SRAM\_CELL} \tag{2}$$

$$Area_{CAM} = \log_2(Num\_Pixels) \times Area_{CAM\_CELL} \times (Num\_Pixels \times PER). \tag{3}$$

$$Overhead_{CAM} = \frac{Area_{CAM}}{Area_{framebuffer}} = \log_2(Num\_Pixels)/8 \times (9/6) \times PER. \tag{4}$$

$$Area_{ESRAM\_MAP} = (\log_2(Num\_Pixels) + B) \times Area_{SRAM} \times (Num\_Pixels \times PER). \tag{5}$$

$$Overhead_{ESRAM\_MAP} = \frac{Area_{ESRAM\_MAP}}{Area_{Frambuffer}} = (\log_2(Num\_Pixels) + B)/8 \times PER. \tag{6}$$

In the equation PER is the value from equation (1). And we assume the ratio of CAM cell area over SRAM cell area 9/6 because CAM cell is composed of 9 TRs while SRAM is of 6 TRs. Based on the equations above we show the area overhead comparison graph in Figure 12 for CAM, Enhanced SRAM with B=0, Enhanced SRAM with B=8, and Flag approach. Below the high density defects (BER < 0.5%) SRAM-based defect map requires the least area overhead among different types. Flag-based defect map requires the least area overhead at higher defect densities.

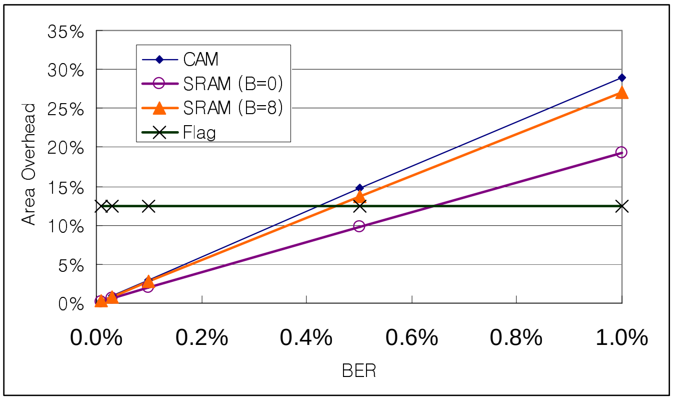


Fig. 12. Area overhead of different types of defect maps

4.2 Power Overhead Comparison

To estimate the power consumption with filtering scheme we scaled the published memory access frequency data from [8] to fit QCIF image with 30 fps and we used the H.264 decoder core power from [9] and estimated our filter core power consumption by the power estimation tool. With the new defect map organization, we can expect quite low power consumption. Figure 13 shows the power consumption by filtering scheme with the enhanced SRAM-based defect map (B=8) for different BERs. Compared with the graph in Figure 6, we see the reduction of the relative power of defect map as well as the reduction of the whole power consumption reduction by filtering scheme. The defect map consumes less than 13% of the total power used by filtering scheme with SRAM-based approach while more than 80% of total filtering power is used by defect map access at BER=1.0%. Figure 14 shows the power consumption by the filtering scheme with the Flag-based defect map at different BERs. We see that the defect map power consumption is constant regardless of BERs while frame buffer consumes the power proportional to the defect rates. The overall power consumption of Flag-based defect map is slightly larger than that of SRAM-based defect map.

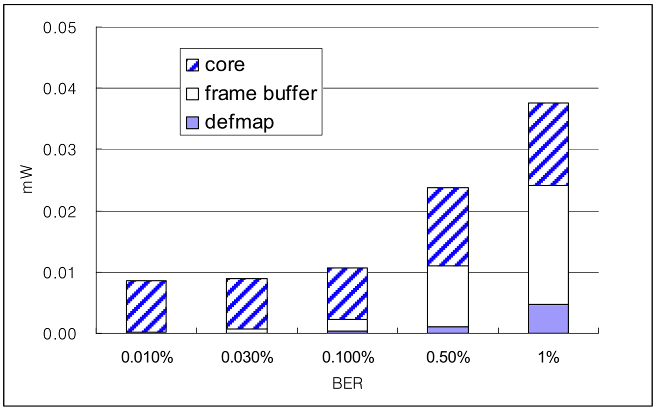


Fig. 13. The power consumption share by filtering scheme with SRAM-based defect map organization

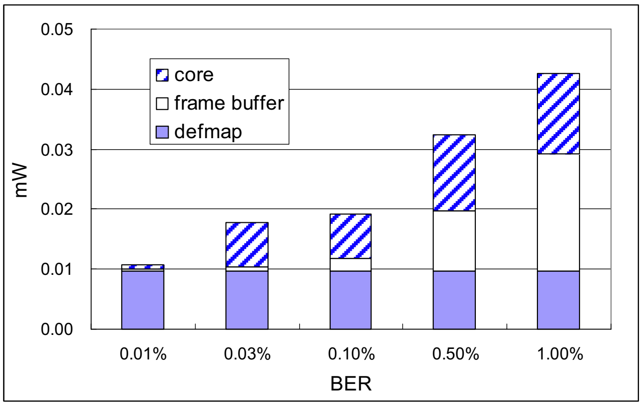


Fig. 14. The power consumption share by filtering scheme with Flag-based defect map organization

Now, in Figure 15 we compare the power consumption overhead by filtering scheme with different defect map organizations (CAM, SRAM, Flag) at different BERs. At BER=1.0% the power overhead by filtering with SRAM-based defect map and Flag-based defect map are less than 1/4 times of the power overhead by filtering with CAM-based defect map. Up to 0.1% BER the SRAM-based defect map is the best in terms of area and power overhead. At very high defect density beyond 0.5% BER and upto 1.0% BER, the SRAM-based defect map organization shows still slightly better performance than the Flag-based organization in terms of power overhead but, Flag-based approach is better than any other approaches considering the area overhead.

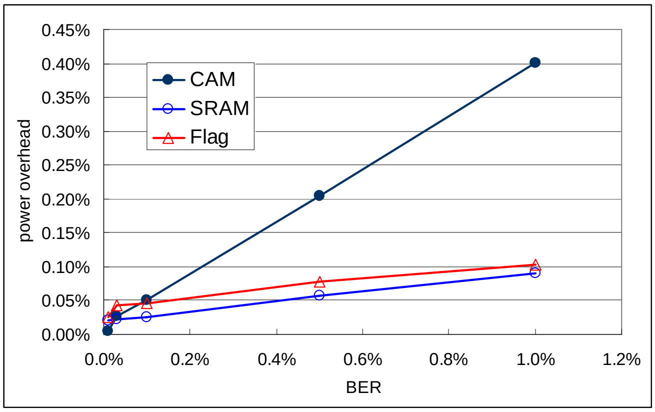


Fig. 15. Power overhead comparison by different defect map organizations at different BERs

5 Conclusion

We focus on the new memory defect recovery method for the multimedia application published in [6] that utilizes the defect map and simple image filters. We found that the defect map in [6] will suffer from the power consumption at higher defect density in the near future. In this paper we propose alternative defect map organizations that can save power consumption by 4 times less than the CAM-based defect map organization by using SRAM –based and Flag-based approach.

References

1. Shoukourian, S., Vardanian, V., Zorian, Y.: “SoC yield optimization via an embedded-memory test and repair infrastructure”. IEEE Design & Test of Computers 21(3), 200–207 (2004)
2. http://public.itrs.net
3. Gupta, T., Jayatissa, A.H.: Recent advances in nanotechnology: key issues & potential problem areas. In: Proceedings of IEEE Conference on Nanotechnology, vol. 2, pp. 469–472 (2000)

4. Agarwal, A., Paul, B.C., Mahmoodi, H., Datta, A., Roy, K.: "A process-tolerant cache architecture for improved yield in nanoscale technologies". IEEE Transactions on Very Large Scale Integration (VLSI) Systems 13(1), 27–38 (2005)
5. Kurdahi, F.J., Eltawil, A.M., Park, Y.-H., Kanj, R.N., Nassif, S.R.: "System-Level SRAM Yield Enhancement". In: Proceedings of the 7th International Symposium on Quality Electronic Design (ISQED 2006), pp. 179–184 (2006)
6. Yi, K., Jung, K.H., Cheng, S.-Y., Park, Y.-H., Kurdahi, F.J., Eltawil, A.: "Design and Analysis of Low Power Filter toward Defect-Resilient Embedded Memories for Multimedia SoC". In: Proceedings of Asia-Pacific Computer Systems and Architecture Conference, pp. 300–313 (September 2006)
7. Pagiamtzis, K., Sheikholeslami, A.: "Contents-Addressable Memory (CAM) Circuits and Architectures: A Tutorial and Survey". IEEE journal of Solid-State Circuits vol.41(3) (2006)
8. Horowitz, M., et al.: H.264/AVC Baseline Profile Decoder Complexity Analysis. In: IEEE Trans. on Circuits and Systems for Video Technology, vol. 13(7) (2003)
9. ARC, http://www.arc.com/subsystems/video/ARC_Video_pb.pdf

An Effective Design of Master-Slave Operating System Architecture for Multiprocessor Embedded Systems

Minyeol Seo, Ha Seok Kim, Ji Chan Maeng, Jimin Kim, and Minsoo Ryu

Computer Engineeirng, College of Information and Communications, Hanyang University, Korea
`{myseo,hskim,jcmaeng,jmkim}@rtcc.hanyang.ac.kr, msryu@hanyang.ac.kr`

Abstract. In this paper, we explore the problem of designing an effective master-slave operating system architecture for multiprocessors and describe current status of our prototype implementation, called APRIX (Asymmetric Parallel Real-tIme KernelS). This work has been largely motivated by the recent emergence of heterogeneous multiprocessors and the fact that the master-slave approach can be easily applied to heterogeneous multiprocessors while SMP (symmetric multiprocessing) approaches are restricted to homogeneous multiprocessors with UMA (Uniform Memory Access). The purpose of this paper is to identify and discuss design issues that have significant impact on the functionality and performance of the master-slave approach. Specifically, our study will investigate three major issues: structural design of a master-slave operating system based on our experience with a prototype development of APRIX, functional design of remote invocation mechanism that is required for executing kernel mode operations on a remote procesor, and performance improvement via application-specific kernel configuration. We finally describe our initial implementation of APRIX and preliminary experiment results.

Keywords: Master-slave, multiprocessor, design issues, remote invocation, kernel configuration.

1 Introduction

As demand grows for high performance and reliability, multiprocessor systems are becoming more widespread ranging from small-scale embedded devices to large-scale supercomputers. From an operating system's perspective, there exist two major classes of operating system architectures for multiprocessor systems. Symmetric multiprocessing (SMP) kernels are the most widely used operating system architecture. All processors run a single copy of SMP kernel that exists on shared memory. Since all processors share the code and data of the SMP kernel, synchronization is a key issue for the SMP architecture. Early implementations of SMP kernels relied on coarse-grained locking to simplify the synchronization problem, but almost all modern SMP kernels are now based on fine-grained locking to achieve the maximum parallelism [2, 6, 9].

© Springer-Verlag Berlin Heidelberg 2007

The other class of operating system architectures is the master-slave architecture [10], which commonly runs on top of heterogeneous multiprocessors with distributed memory. One processor is designated as the master that is responsible for handling all system calls and interrupts while other processors are designated as slaves that execute application tasks only in user mode. This organization greatly simplifies the problem of synchronization since only the master can access the kernel code and data. However, as the number of slaves increases, the master will become a critical performance bottleneck.

In this paper, we explore the problem of designing an effective master-slave operating system architecture for multiprocessors, and also describe the current status of our prototype implementation, called APRIX (Asymmetric Parallel Real-tIme KernelS). This work has been largely motivated by the recent emergence of heterogeneous multiprocessors such as Philips Nexperia, TI OMAP, ST Nomadic, Qualcomm MSM, and STI Cell [14]. Processor heterogeneity may offer many opportunities to better reduce the development cost and resource consumption while meeting application-specific performance requirements [22]. As a result, heterogeneous multiprocessors are expected to become widespread in future embedded systems. One advantage of the master-slave approach is that it can be easily applied to heterogeneous multiprocessors while the SMP approach is restricted to homogeneous multiprocessors with UMA (Uniform Memory Access). Additionally, the master-slave approach allows a more straightforward way of implementation than the SMP approach due to the simplified synchronization problem. In spite of these, however, it seems that the inherent performance problem of the master-slave architecture has rendered it less well studied than the SMP approach.

The purpose of this paper is to identify and discuss design issues that have significant impact on the functionality and performance of the master-slave approach. Specifically, our study will investigate three major issues that are closely related to the architecture and performance of the master-slave approach. The first issue is the structural design of a master-slave operating system. We will discuss how a master-slave operating system can be constructed from a legacy uniprocessor operating system. This discussion will be based on our experience with a prototype development of APRIX. The second issue is the functional design of remote invocation mechanism. As mentioned above, all the kernel mode operations are executed only in the master. When a task on some slave requests a kernel mode operation like a system call, this request must be serviced by the remote master. We will show that a naive approach to remote invocation may raise serious performance problems such as indefinite task blocking and priority inversion, and also show that a priority-based approach can overcome these problems. The last issue is closely related with the inherent architectural problem of the master-slave approach. When application tasks involve a significant portion of kernel mode operations, the master may become a critical bottleneck. The problem will be made worse as the number of processors increases. To tackle this problem, we suggest a component-based kernel architecture and application-specific kernel configuration. With a priori knowledge about the application tasks' behavior, each kernel can be configured such that all the required implementations are included within the kernel itself. This will effectively reduce the number of interactions between the master and slaves, and thereby achieving much higher performance.

We finally present the initial implementation of APRIX and preliminary experiment results. Unfortunately, the current implementation offers a limited set of features we describe above, and our experiments thus do not provide comprehensive results. We believe that a thorough evaluation of our approach can be reported with a full APRIX implementation in the near future.

1.1 Related Work

Traditional SMP kernels have evolved from monolithic uniprocessor kernels [4, 5, 6, 7, 8]. Since several processors can execute simultaneously in the kernel and may access the same kernel data structures, sophisticated synchronization mechanisms were required in adapting uniprocessor kernels to SMP versions.

Early uniprocessor kernels like UNIX operating systems prevented more than one kernel thread or interrupt handler from executing at the same time in the kernel [4]. For instance, any thread running in the kernel can continue its execution without being preempted until it completes its kernel mode operations or voluntarily blocks for some resources. However, it is still possible for the thread to be interrupted from hardware. The interrupt handler may manipulate the same data structures with which the current thread was working. A conventional approach to this problem has been masking interrupts whenever the thread executes a critical region of code.

However, the above protection schemes are not enough in multiprocessor environments since several processors can simultaneously execute in the kernel. This requires that the kernel be partitioned into critical regions and at most one thread of control can execute in a critical region at a time. In the most extreme case, the entire kernel can be considered as a single big critical region. This allows only one processor to be active in the kernel while other processors are prevented from entering the kernel [2, 3, 1]. This, called the *giant lock* approach, may greatly simplify the implementation of SMP kernel, but may suffer from contention on the giant kernel lock.

On the other hand, fine-grained locking can mitigate the contention problem by partitioning the kernel into a number of smaller critical regions [6, 9]. This scheme requires more careful design and is more prone to errors such as deadlocks. But it is able to produce much higher performance gain than the coarse-grained locking scheme. Consequently, many modern operating systems rely on fine-grained locking. Examples based on fine-grained locking include the FreeBSD [2], AIX [6], and Solaris [9].

In the master-slave approach, one processor is designated as the master that can execute in kernel mode while other processors are designated as slaves that execute only in user mode. In [10], Goble *et al.* implemented a master-slave system on a dual processor VAX 11/780, where the master is responsible for handling all system calls and interrupts. Slaves execute processes in user mode and send a request to the master when a process makes a system call. Recent work on the master-slave approach has been reported in [11]. Kagstrom *et al.* attempted to provide multiprocessor support without modifying the original uniprocessor kernel. Their idea was to create and run two threads for each application, a bootstrap thread and an application thread. The application thread runs the application on the slave and the bootstrap thread runs on the master awaiting a request for system call. On receiving a request, the bootstrap thread then calls the requested system call on behalf of the application thread.

There exist another interesting approaches called asymmetric kernels [12, 13], where each kernel provides different OS services. For example, one kernel may provide network services while another provides file system services. The AsyMOS (Asymmetric Multiprocessor Operating System) in [12] assumes SMP hardware, but assigns different functions to different processors. It divides processors into two functional groups, device processors and application processors. Each device processor is dedicated to a specific device such as network card or disk controller while the application processor runs a native kernel. The kernel that runs on the device processor, called light weight device kernel (LDK), includes and executes all device-specific code, thus producing increased performance for I/O-intensive applications.

The remainder of this paper is organized as follows. Section 2 describes the structural design of a master-slave operating system. Section 3 presents two techniques for performance optimization in the master-slave approach. Section 4 describes a prototype implementation of APRIX and experiment results. Section 5 concludes this paper with future work.

2 Structural Design of Master and Slave Kernels

The master-slave approach can be applied to a wide range of multiprocessor architectures while the SMP approach is restricted to homogeneous multiprocessors with shared memory. Note that the seminal work by Goble *et al.* [10] was performed on homogeneous processors. In this paper, however, the primary target of our master-slave operating system is heterogeneous multiprocessors such as the Cell processor [23].

We begin with a simple model of master-slave that is similar to that of [10]. The master kernel has two basic functions: assigning application tasks to slaves and providing kernel services to slaves. Every slave, on the other hand, has a simple function of executing application tasks in user mode. Note that the master kernel itself is also able to run application tasks.

Based on the above master-slave model, we assume two-level priority-based task scheduling. The master has a global scheduler that selects the highest priority task from a single global run queue and assigns it to a slave processor. Each slave has a local scheduler that selects the highest-priority task from a local run queue. The local run queue stores runnable tasks that have been assigned by the global scheduler. In this paper, we will focus on architectural and performance issues rather than algorithmic issues about multiprocessor. For a general review of multiprocessor scheduling for embedded real-time systems, see [24].

2.1 Organizing Master and Slave Kernel Structures

We now describe how to organize master and slave kernels from an existing uniprocessor kernel. Recall that the main functions of master kernel include assigning application tasks onto slaves and servicing kernel mode operations requested by slaves. Therefore, it is very straightforward to organize the master kernel's structure. It can be done by incorporating the original uniprocessor kernel with additional components to support the above two functions. The additional components are a *global task scheduler* and a *kernel service handle*, which are responsible for task assignment and kernel services, respectively. Fig. 1 shows a typical structure of embedded operating system for uniprocessor and Fig. 2 (A) shows the master kernel's structure, where shaded boxes represent added components and white boxes represent native components. In fact, Fig. 1 has been taken from our research kernel QURIX intended for uniprocessors and Fig. 2 has been taken from our initial version of APRIX.

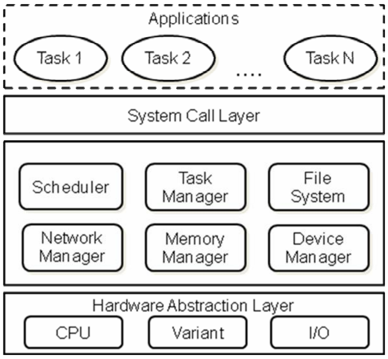
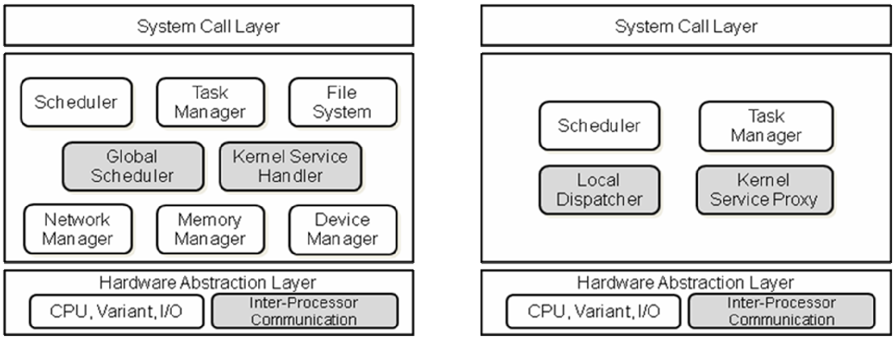


Fig. 1. A common structure of embedded operating system for uniprocessor



(A) Master kernel's architecture (B) Slave kernel's architecture

Fig. 2. Structures of master and slave kernels for multiprocessors

The slave kernel requires a minimal set of functions. First, it should be able to create, schedule, and execute application tasks assigned by the master kernel. Second, it should be able to support remote invocation of kernel mode operations. As a result, many of the original components can be removed from the uniprocessor kernel while the scheduler and task manager remain. The slave kernel also requires two additional components, a *local dispatcher* and a *kernel service proxy* that are responsible for handling master's command for task assignment and managing remote invocation, respectively. Fig 2 (B) shows the resulting structure of slave kernel.

2.2 Interactions Between Master and Slave Kernels

There are two types of interactions between master and slaves. The first type of interaction is associated with task assignment and scheduling. When the global scheduler in the master decides to assign a task to a slave, it initiates communication by sending a message to the slave. On receiving the message, the slave's local dispatcher interprets the message and immediately requests the thread manager to create a new task. The local dispatcher has another function. It should inform the master when some scheduling event occurs. For instance, if a task completes, the local dispatcher informs the master of this event.

The second type of interaction is carried out for remote invocation of kernel mode operations. When a task on slave invokes a kernel mode operation such as a system call, the kernel service proxy in the slave initiates communication by sending a message to the master. The kernel service handler in the master then invokes the requested operation and returns results to the kernel service proxy.

In order to enable the above interactions between master and slaves, both kernels should incorporate a common component, called *inter-processor communication* component. Since the implementation of this component mostly depends upon the underlying hardware architecture and communication mechanisms, it is desirable to place the component at the lowest layer of kernel structure such as HAL (hardware abstraction layer). The inter-processor communication component should be implemented to provide a minimal set of message passing interfaces including **send** and **receive**. These interfaces may be implemented by using an interrupt mechanism when inter-processor interrupt is supported by the hardware or a polling mechanism when global shared memory is available.

2.3 Remote Invocation Mechanism

Remote invocation is not new. It has been well studied in the fields of middleware and distributed operating systems. However, we need to revisit this issue since it has a significant impact on the performance in the master-slave architecture.

Remote invocation is supported by the kernel service proxy in slave and the kernel service handler in master. The slave-side proxy is merely a procedure that looks like the requested kernel operation. Its main function is to convert the operation name and parameters into a message and send the message to the master by using the **send** operation provided by the inter-processor communication component. The master-side kernel service handler receives the message, and parses the received message to extract the operation information. It then invokes the requested operation, packs the result into a message, and sends it to the slave. The slave-side proxy in turn gets control back, extracts the result from the message, and returns the result to the caller task. Since processors may have different endian formats, the kernel service proxy should also handle endian problems.

It should be noted that remote invocation cannot directly support call-by-reference evaluation even though many system calls involve passing pointers as parameters. The reason is that the call-by-reference relies on the existence of a shared address space, in which the referenced data exist and is accessible to the callee. In remote invocation with distributed memory, the caller and callee generally have separate address spaces, and referenced data thus cannot be accessed by the callee. A well-studied technique to this problem is call-by-copy-restore. Call-by-copy-restore copies the referenced data and makes it accessible to the callee. When the call is complete, all the modifications made during the call are reproduced on the original data. This call-by-copy-restore has almost the same effect as call-by-reference.

Note that remote invocation can also be used between application tasks. When an application task requires a certain function provided by a remote application task, then user-level remote invocation would be useful. However, the user-level remote invocation is beyond the scope of operating system, and thus we consider only the kernel-level remote invocation in this paper.

3 Performance Optimization Considerations

In this section, we discuss two important performance issues in the master-slave approach. The first issue is associated with the design choice for remote invocation mechanism. We show that if not properly designed, the remote invocation mechanism may lead to serious performance problems such as indefinite task blocking and priority inversion. We then show that a priority-based approach can overcome these problems. The second issue is closely related with the inherent architectural problem of the master-slave approach. As mentioned earlier, the master may become a hot spot of contention as the number of slaves increases. To tackle this problem, we suggest a component-based kernel architecture and application-specific kernel configuration.

3.1 Priority-Based Remote Invocation in Top Half

There are two ways of servicing a remote invocation request for kernel operation, top half and bottom half approaches. Note that in UNIX terminology [16], the kernel can be divided into top half and bottom half. The top half of the kernel provides services in response to system calls or synchronous traps and the bottom half of the kernel provides services in response to hardware interrupts. Both the top half and bottom half execute in a privileged execution mode, but the top half runs in a task context while the bottom half runs with no task context. Note that the terms top half and bottom half have different meanings when used within the context of interrupt handling [3].

The bottom half of kernel is simply a set of routines that are invoked to handle interrupts. Consequently, servicing a remote invocation request within bottom half may be straightforward and fast. However, the bottom half approach has three serious drawbacks. First, it can only be used for operations that can be executed safely in an interrupt handler. In most operating systems, interrupt handlers are not allowed to block. Since blocking and resuming requires a per-task context, an interrupt handler cannot relinquish the processor in order to wait for resources, but must run to completion. Second, the bottom half approach may lead to indefinite blocking of the current task that was running before the interrupt occurs. When a series of remote invocation requests arrive at the master, the application tasks on the master kernel will be interrupted and must wait until all the requests are completely serviced. Furthermore, new requests may arrive during the old requests are being serviced. This implies that the application tasks can suffer from indefinitely long blocking. Third, since interrupt handling commonly has a higher priority than application tasks, high priority tasks may be interrupted by the remote requests initiated by low priority tasks.

On the other hand, servicing a remote invocation request within top-half allows the kernel service handler to execute in a task context. This permits the requested operation to block during its execution. Note that the top half approach may be slower since it requires a context switch out of the interrupt handler and may require synchronization with other tasks running in the kernel. It also requires more complicated implementation. In spite of these, the fact that many system calls involve blocking and the other two drawbacks of the bottom half approach necessitate the top half approach.

The other two drawbacks of the bottom half approach can be overcome by combining the top-half approach with a priority scheme. A priority is associated with each request for remote invocation. The priority may be designated by the programmer or may simply be inherited from the task that made the request. When the kernel service handler gets control, it services all the pending requests that have higher priorities than the current task. The lower priority requests can be checked and serviced later when the handler gets control again. Note that the handler can get control in two cases, when a new request for remote invocation arrives or when the processor switches from kernel space to user space. In this way, we can ensure that high priority tasks never be preempted by requests originated from low priority tasks.

3.2 Application-Specific Kernel Configuration

The master-slave architecture does not scale well since the master may become a bottleneck. To tackle this problem, we propose to use a kernel that has a highly modular architecture and to configure each kernel to match application-specific requirements. By including only the kernel components required by the slave's tasks, the number of interactions between master and slave kernels can be minimized. This approach raises another difficult problem of software modularization and composition. Fortunately, there have been some recent advances in this area and now many useful solutions are available.

Configurability has been heavily investigated in the fields of programming languages, middleware, and operating systems. Friedrich *et al.* provide an exhaustive survey ranging from statically configurable operating systems to dynamically reconfigurable operating systems [17]. Many of the configurable operating systems described in [17] can be used for our purpose. Microkernel-based operating systems such as L4 [18] and Pebble [19] allows the operating system to be tailored to application-specific requirements. However, the microkernel-based operating systems have limited configurability in that they always require a fixed set of core functionality to be included in all the application. The IPC (inter-process communication) cost is another issue since operating system services are implemented as user-level servers in microkernel approaches. On the other hand, component-based operating systems like OS Kit [20] and eCos [21] do not assume any fixed set of functionality, and seem to be more appealing for the general problem of application-specific kernel configuration.

We are currently working on improving the configurability of APRIX that has almost a monolithic structure. We have the same goal of application-specific configurability as that of [18, 19, 20, 21]. Our approach, however, is a bit different in that we consider system calls as a unit of operating system configuration. In our master-slave architecture, system calls are the main source of performance degradation. As a result, if we can organize each kernel such that only the fine-grained components required to run the application are included within the kernel, then the number of interactions between the master and slaves may be significantly reduced. Here we briefly describe our strategy for the development of highly-configurable APRIX.

The first step of our strategy is to partition all APRIX functions including system calls and in-kernel functions into groups according to their functional similarity. The second step is to refine the initial groups into a number of fine-grained components according to the level of cohesion between functions. Bieman *et al.* provides a useful method for quantitatively measuring functional cohesion [22], in which the method is based on a program slice that is the portion of program text that affects a specified program variable. We make use of the method in [22] for the second step, and the resulting components are considered as a unit of configuration. In the last step, we encapsulate state and functionality within each component to reduce coupling between components. We then explicitly define system calls that exist in each component as the exported interfaces for that component.

4 Case Study: Developing APRIX on MPSoC-II

An initial version of APRIX has been implemented with a uniprocessor kernel—QURIX that we had developed for academic purpose in year 2004. The development of APRIX is still going on and the current version provides a limited subset of the features that we describe in Section 2 and 3.

QURIX is a library that can be statically linked with an application. It is able to execute a multithreaded application within the single address space. The overall structure of QURIX is shown in Fig. 1. At the lowest level, it has a HAL (Hardware Abstraction Layer) that consists of three parts, CPU-dependent code, variant-dependent code for managing MMU, FPU, DMA, interrupt controller, and I/O-dependent code for I/O devices such as UART and timer. Above the HAL layer, QURIX has six components: a priority-based scheduler, a thread manager that is responsible for thread creation and termination, a light weight flash file system, a network manager for serial communications, a memory manager, and a device manager, which together provide fundamental operating system services to application software via system call layer. QURIX provides a subset of POSIX 1003.1 standard interfaces at the system call layer to support thread management, synchronization, I/O operations, dynamic memory management, and a number of standard C library functions.

APRIX runs on a four-processor board—MPSoC-II that is a general-purpose prototyping board for developing MPSoC-based electronic designs. The major components of the prototyping board include four ARM926EJ-S processors, 1MB shared memory, and 64MB local memory for each processor. The current implementation of APRIX is shown in Fig. 2 (A). APRIX makes use of the 1 MB shared memory to exchange messages between the master and slaves. APRIX maintains four types of queues within the shared memory. They include task allocation queue, task status queue, kernel service call queue, and kernel service return queue. Each of these queues is created and maintained on a per-slave basis, thus four queues for each slave.

When the master's global scheduler decides to assign a task to a slave, it puts a message onto the task allocation queue associated with the slave. The message contains a thread identifier, priority, the name of function to be created as a thread, and parameters to the function. The slave's local dispatcher then takes the message from the queue and requests its thread manager to create the requested task. When some important events occur such as blocking or termination, the local dispatcher informs the master via the task status queue. When a task running on a slave invokes a system call, the local dispatcher performs marshaling and puts a message onto the kernel service call queue. The message contains the slave's identifier, task's identifier, system call number, and parameters to the system call. The master's kernel service handler in turn executes the requested operation and return results via the kernel service return queue.

Table 1. Overheads of allocating tasks and reporting task status

Operation	Ave.	Max.	Min.
Master's writing to task allocation queue	47.2 us	53.8 us	46.8 us
Slave's reading from task allocation queue	99.8 us	113.4 us	100.4 us
Slave's writing to task status queue	37.6 us	40.2	37.4 us
Master's reading from task status queue	71.8 us	77.2	70.2 us

Table 2. Overheads of requesting a system call and returning results

Operation	Ave.	Max.	Min.
Slave's writing to kernel service call queue	35.0 us	37.2 us	34.2 us
Master's reading from kernel service call queue	52.4 us	53.4 us	51.0 us
Master's writing to kernel service return queue	34.4 us	34.8 us	34.2 us
Slave's reading from kernel service return queue	56.4 us	60.0 us	56.4 us

We performed experiments to measure communication costs with all ARM processors running at 50 MHz. The overheads of allocating tasks and reporting task status are given in Table 1 and the overheads of requesting a system call and returning a result are given in Table 2.

We performed another experiments to concentrate exclusively on the computational performance of APRIX kernels. We constructed a simple benchmark application that does not involve any system calls, thus minimizing the cost for communications between master and slaves. The benchmark creates three threads and each thread independently executes 1000 multiplications of 6x6 matrices. Running this benchmark gave 2.9 seconds with a single processor, 1.7 seconds with two processors, and 1.1 seconds with three processors. This indicates that the speedups over the single processor are 1.705 and 2.636 with two processors and three processors, respectively. From the above experiment results, we may be certain to some extent that the APRIX kernels have been correctly designed and implemented, and that reasonable performance can be achieved through our design.

5 Conclusion

In this paper we presented an effective master-slave operating system architecture for heterogeneous multiprocessors with distributed memory. We reported our experience with the initial APRIX development, and this will provide some degree of guidance to the implementor who wants to employ the master-slave model. We also identified performance issues that are inherent to the master-slave architecture and proposed effective techniques including the priority-based handling of remote invocations and application-specific kernel configuration.

We could not perform thorough evaluation for the proposed techniques since the implementation was not complete at the moment of this writing. We will continue to implement the techniques described in Section 2 and 3, with a special emphasis on factoring APRIX kernels into fine-grained components.

References

1. Kagstrom, S., Grahn, H., Lundberg, L.: Experiences from implementing multiprocessor support for an industrial operating system kernel. In: Proceeding of 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, pp. 365–368. IEEE Computer Society Press, Los Alamitos (2005)
2. Lehey, G.: Improving the FreeBSD SMP Implementation. In: Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference, pp. 155–164 (2001)
3. Love, R.: Linux Kernel Development, 2nd edn. Novell Press (2005)
4. Vahalia, U.: UNIX Internals: The New Frontiers. Prentice Hall, Upper Saddle River New Jersey (1996)
5. Bach, M.J.: Design of the UNIX Operating System. Prentice Hall, Englewood Cliffs (1986)
6. Clark, R., O'Quin, J., Weaver, T.: Symmetric Multiprocessing for the AIX Operating System. In: Proceedings of the 40th IEEE Computer Society International Conference in Compcon, pp. 110–115. IEEE Computer Society Press, Los Alamitos (1996)
7. Janssens, M.D., Annot, J.K., Van De Goor, A.J.: Adapting UNIX for a Multiprocessor Environment. Communications of the ACM 29, 895–901 (1986)
8. Russell, C.H., Waterman, P.J.: Variations on UNIX for Parallel-Processing Computers. Communications of the ACM 30, 1048–1055 (1987)
9. Kleiman, S., Voll, J., Eykholt, J., Shivalingiah, A., Williams, D.: Symmetric Multiprocessing in Solaris 2.0. In: Proceedings of the Thirty-Seventh International Conference on COMPCON, pp. 181–186 (1992)
10. Goble, G.H., Marsh, M.H.: A Dual Processor VAX 11/780. In: Proceedings of the 9th Annual Symposium on Computer Architecture, pp. 291–298 (1982)
11. Kagstrom, S., Lundberg, L., Grahn, H.: A Novel Method for Adding Multiprocessor Support to a Large and Complex Uniprocessor Kernel. In: Proceedings of 18th International Parallel and Distributed Processing Symposium (2004)
12. Muir, S., Smith, J.: AsyMOS-an Asymmetric Multiprocessor Operating System. In: Proceedings of Open Architectures and Network Programming, pp. 25–34 (1998)
13. Muir, S., Smith, J.: Functional Divisions in the Piglet Multiprocessor Operating System. In: Proceedings of the 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications, pp. 255–260. ACM Press, New York (1998)

14. Wolf, W.: The Future of Multiprocessor Systems-on-Chips. In: Proceedings of the 41st Annual Conference on Design Automation, pp. 681–685 (2004)
15. Maeda, S., Asano, S., Shimada, T., Awazu, K., Tago, H.: A Real-Time Software Platform for the Cell Processor. IEEE Micro. 25, 20–29 (2005)
16. Carbone, J.: A SMP RTOS for the ARM MPCore Multiprocessor. ARM Information Quaterly 4, 64–67 (2005)
17. Friedrich, L.F, Stankovic, J., Humphrey, M., Marley, M., Haskins, J.: A Survey of Configurable, Component-Based Operating Systems for Embedded Applications. IEEE Micro. 21, 54–68 (2001)
18. Liedtke, J.: On micro-kernel construction. In: Proceedings of the 15th ACM Symposium on Operating Systems Principles, ACM Press, New York (1995)
19. Gabber, E., Small, C., Bruno, J., Brustoloni, J., Silberschatz, A.: The Pebble Component-Based Operating System. In: Proceedings of the USENIX Annual Technical Conference (1999)
20. Ford, B., Lepreau, J., Clawson, S., Maren, K.V., Robinson, B., Turner, J.: The Flux OS Toolkit: Reusable Components for OS Implementation. In: Proceedings of the 6th Workshop on Hot Topics in Operating Systems (1997)
21. eCos, http://sources.redhat.com/ecos
22. Bieman, J.M., Ott, L.M.: Measuring Functional Cohesion. IEEE transactions on Software Engineering 20, 644–657 (1994)
23. Kumar, R., Tullsen, D.M., Jouppi, N.P., Ranganthan, P.: Heterogeneous Chip Multiprocessors. IEEE Computer 38, 28–32 (2005)
24. Kahle, J.A., Day, M.N., Hofstee, H.P., Johns, C.R., Maeurer, T.R., Shippy, D.: Introduction to the Cell Multiprocessor. IBM Journal of Research and Development 49, 589–604 (2005)
25. Carpenter, J., Funk, S., Holman, P., Srinivasan, A., Anderson, J., Baruah, S.: A Categorization of Real-Time Multiprocessor Scheduling Problems and Algorithms. In: Leung, J.Y.-T. (ed.) Handbook of Scheduling: Algorithms, Models, and Performance Analysis, Chapman Hall/CRC Press, Boca Raton, USA (2004)

Optimal Placement of Frequently Accessed IPs in Mesh NoCs

Reza Moraveji[1, 2], Hamid Sarbazi-Azad[3,1], and Maghsoud Abbaspour[2,1]

[1] IPM School of Computer Science, Tehran, Iran
[2] Department of Electrical and Computer Engineering, Shahid Beheshti University, Tehran, Iran
[3] Department of Computer Engineering, Sharif University of Technology, Tehran, Iran
moraveji_r@ipm.ir, azad@sharif.edu, maghsoud@ipm.ir

Abstract. In this paper, we propose the first interrelated power and latency mathematical model for the Networks-on-Chip (NoC) architecture with mesh topology. Through an analytical approach, we show the importance of tile selection in which the hot (frequently accessed) IP core is mapped. Taking into account the effect of blocking in both power and latency models, causes the estimated values to be more accurate. Simulation results confirm the reasonable accuracy of the proposed model. The major output of the model which is the average energy consumption per cycle in the whole network is the efficacious parameter that is most important and must be used by NoC designers.

Keywords: Power model, Latency model, Network on chip, System on chip, Mesh, IPs/Cores mapping.

1 Introduction

NoC has been emerging as potential tile-base architecture which offers scalable and highly utilized bandwidth architectures for implementing system-on-chips (SoCs) [1, 2]. Since in upcoming years, it is expected that the number of tiles in NoCs becomes enormous, one of the key factors in designing tile-base NoCs is topological placement or mapping of IPs/cores onto the tiles.

The mapping problem affects the average message latency and also overall power consumption in NoC. In [3, 4], authors used a branch-and-bound algorithm to map IPs onto tiles in which total communication energy consumption is minimized under performance constraints. The same work is done by S. Murali et al. [5] considering the mesh NoC architecture. In [6], a two-step genetic algorithm is proposed for mapping an application on to NoC architecture with a two-dimensional mesh of switches as a communication backbone.

In this work, we model the degree of intensity of events, tasks, and elements that have an impact on the average power and latency of NoCs through an analytical model; these include degree of buffer multiplexing, blocking of the message, leakage power, routing algorithm, traffic pattern, minimum distance, message length and etc. Since mathematical modeling is perhaps the most cost-effective and versatile way, which can be used for testing and evaluating the system's performance (latency and power) under different working conditions and configurations, therefore the aim of

© Springer-Verlag Berlin Heidelberg 2007

this study is to make an overall view and perception on the effects of sensitive parameters to mapping problem.

2 Energy Consumption Model

An analytical energy consumption model in a two-dimensional mesh that uses wormhole switching is derived and validated. As presented in figure 1, each node in NoC architecture consists of a crossbar switch which has *X* input physical channels as well as an injection channel and *X* output physical channels in addition to an ejection channel. Each physical channel is associated with *V* buffers (virtual channels). Arbiter unit which chooses the path and connects and disconnects the input channel with an appropriate output channel based on the routing algorithm and network status information, is the other part of the NoC node. Here, the measure of interest is the power consumption per message. In order to achieve an accurate power model for a message crossing the network, we have to take into account both static and dynamic powers a message may consume. Static power is the power dissipated when the message crosses the network without contention. This implies that the message does not encounter any blocking while traveling the network. Therefore, for a specific pair of source and destination nodes, the static energy consumed by a flit message crossing the given pair is directly proportional to the Manhattan distance between source and destination. Thus,

$$E_{static}^{flit}(s,d) = (dis(s,d)+1)\times E_{node} + dis(s,d)\times E_{link}, \quad dis(s,d) = |X_d - X_s| + |Y_d - Y_s| \qquad (1)$$

where E_{node} is the power dissipated in a node per flit transfer or intra-node power dissipation per flit transfer. E_{link} is the inter-node (link) power dissipation per flit transfer and $dis(s,d)$ is the Manhattan distance between source and destination.

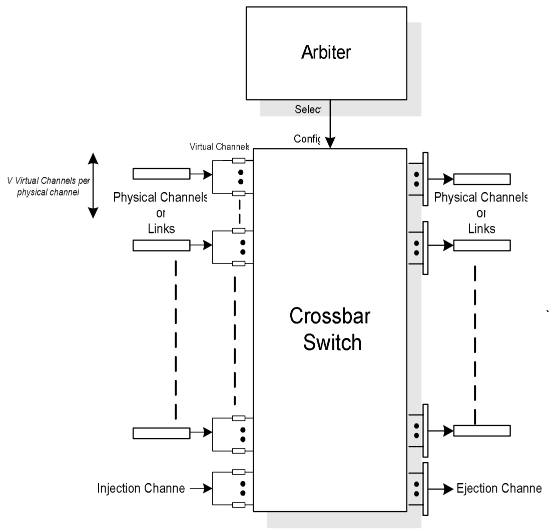

Arbiter
Select
Config
Virtual Channels
V Virtual Channels per physical channel
Physical Channels or Links
Crossbar Switch
Physical Channels or Links
Injection Channel
Ejection Channel

Fig. 1. NoC node structure

Intra-node power dissipation in turn is divided into three parts. According to figure 1, when a flit emerges at the head of the virtual channel (buffer) an amount of energy is dissipated for writing/reading the flit message to/from the input buffer. The energy dissipated during writing and reading process is referred to as buffering energy. Another part of intra-node energy dissipation is as a result of traversing the flit through crossbar switch, E_{sw}. The decision made by the arbiter to send the header flit to the desired virtual channel of physical channel (link) of interest is the last part of the intra-node energy consumption. Note that the last part of energy is solely considered for the header flit, according to the fact that the header flit reserves the path in which the tail follows, therefore only the header deals with the arbiter. Although arbitration unit consumes negligible energy, but in order to have an exhaustive model we distinguish between the intra-node energy consumption for the header flit and the rest of the message flits (tail). Thus [7, 8]

$$E_{node} = \begin{cases} E_{node,h} = E_{wrt} + E_{rd} + E_{arb} + E_{sw} & \text{for header flit} \\ E_{node,t} = E_{wrt} + E_{rd} + E_{sw} & \text{for tail flit} \end{cases} \tag{2}$$

By considering the message length, *L*, and the effect of pipelining, the static part of the energy dissipated by such a message crossing from source node *s* to destination node *d* is calculated as

$$\begin{aligned} &E_{static}(s,d) = (dis(s,d)+1)\cdot(E_{node,h} + (L-1)\cdot E_{node,t}) \\ &+ L \times dis(s,d) \times E_{link} \end{aligned} \tag{3}$$

Wormhole routers stop the message “in place” when its head is blocked. The head of the message stops, and the remainder of the message is stopped, holding the buffers and channels along the path it has already formed [9].

As presented in [10], virtual channels (buffers) used in the router architecture are implemented as SRAM arrays. SRAM cells contribute to three different phases referred to as write, read and idle phase [11]. The leakage current associated with data retention in idle phase is a considerable source of power dissipation. The effect of the leakage power is highlighted especially when the number of messages in the network increases (number of contentions goes up). Therefore, the effect of blocking of the message in the network may vary the total energy consumption per cycle in the network. During the blocking of message, all buffers occupied by the message are in idle phase and if we assume E_{idle} as the energy dissipated in one clock cycle of idle phase in a flit buffer, then the product of blocking time, $T_{blocking}$, and E_{idle} results in considerable energy dissipation for each flit buffer. Figure 2 shows a sample scenario on which the calculations are based. In order to calculate blocking time of the message in node a given that the message generated at a specific node, *s*, and destined to another specific node, *d*, it is necessary to compute the blocking probability of the message in node *a*, the probability that the given node is traversed by the message and the minimum waiting time seen by the message in that node to acquire one free virtual channel. Therefore, the blocking time can be expressed as

$$T_{blocking}(a) = P_{(s,d)}^{pass}(a) \times P_{(s,d)}^{block}(a) \times W_{\min}(a) \tag{4}$$

Probability $P_{(s,d)}^{pass}(a)$ can be calculated by computing the probability of traversing the permitted channels to node *a* by the message and adding up all theses probabilities. The permitted channel to node *a* is an incoming channel to the given node located in at least one of the minimal paths between *s* and *d*. Therefore, these channels can be mathematically represented by

$$INch = \{ b \mid dis(b,d) = dis(a,d) + 1 \} \tag{5}$$

Then, we can write

$$P_{(s,d)}^{pass}(a) = \sum_{b \in INch} P_{(s,d)}^{pass} < b,a > \tag{6}$$

In order to calculate $P_{(s,d)}^{pass} < b,a >$, first it is necessary to calculate the probability that channel $< b,a >$ is traversed given that the node *b* is traversed, $P_{(s,d)}^{pass}(< b,a > \mid b)$.

Since the topology used in this study is two-dimensional mesh, possible outgoing physical channels from an intermediate node toward the destination may be in dimension *X*, *Y* or both. If there is just one possible outgoing physical channel from the given node to the destination (node h and g in figure 2), the mentioned conditional probability will be simply 1,

$$P_{(s,d)}^{pass}(< g,d > \mid g) = 1 \tag{7}$$

If the message can be routed in both dimensions *X* and *Y*, the probability that the message traverses dimension *X* is given by

$$\begin{aligned} &P_{(s,d)}^{pass}(< b,c > \mid b) = (1 - P_{ada} < b,c >) P_{ada\&\det} < b,a > \\ &+ \frac{1}{2}(1 - P_{ada} < b,c >)(1 - P_{ada\&\det} < b,a >) \\ &+ \{ P_{ada} < b,c > \cdot P_{ada\&\det} < b,a > \}_{W(\min) = W(a)} \end{aligned} \tag{8}$$

where $P_{ada} < b,c >$ is the probability that all the adaptive virtual channels associated to the physical channel $< b,c >$ are busy and $P_{ada\&\det} < b,a >$ is the probability of all the adaptive and deterministic virtual channels of the physical channel $< b,a >$ being busy. The first term in the above equation expresses the probability that all adaptive virtual channels associated to the physical channel located along with dimension *Y* is occupied by other messages and all deterministic and adaptive virtual channels associated to the physical channel located in dimension *X* is free; therefore, the message is routed through dimension *X*. The second term shows the probability that both physical channels are free so the routing function uses one of them selectively with the fair probability of 0.5. The last term considers the situation where both channels are busy. Therefore, choosing the channel with minimum waiting time is inevitable and the term is added when the calculated waiting time associated with dimension *X* is smaller. The probability that the message traverses dimension *Y* is calculated in the same manner with a little difference. Since the deterministic routing function used in the model is dimension-ordered, it should be noted that if a permitted physical channel exists in dimension *X* as an alternative to one exists in dimension *Y*, the channel belongs to dimension *X* is always chosen as the skip channel [14]. Thus,

$$\begin{aligned} P_{(s,d)}^{pass}(<b,c>|b) &= (1-P_{ada}<b,c>)P_{ada\&\det}<b,a> \\ &+\frac{1}{2}(1-P_{ada}<b,c>)(1-P_{ada\&\det}<b,a>) \\ &+\{P_{ada}<b,c>\cdot P_{ada\&\det}<b,a>\}_{W(\min)=W(c)} \end{aligned} \tag{9}$$

The probability of traversing a specific channel, $<b,a>$, by the message originated from node *s* and destined to node *d* is achieved by

$$P_{(s,d)}^{pass}<b,a> = P_{(s,d)}^{pass}(b)\times p_{(s,d)}^{pass}(<b,a>|b) \tag{10}$$

Therefore, $P_{(s,d)}^{pass}(a)$ can be obtained by the following iterative process: Step 1) $P_{(s,d)}^{pass}(a)=\sum_{b\in INch} P_{(s,d)}^{pass}<b,a>$; Step 2) $P_{(s,d)}^{pass}<b,a> = P_{(s,d)}^{pass}(b)\times p_{(s,d)}^{pass}(<b,a>|b)$; Step 3) if (*b* != *s*) then go to step 1 else $P_{(s,d)}^{pass}(s)=1$.

$P_{(s,d)}^{block}(a)$ is the probability that all adaptive virtual channels of all the physical channels that can be traversed and the deterministic virtual channel of the only physical channel that can be traversed deterministically are all busy. Therefore,

$$P_{(s,d)}^{block}(a) = P_{ada\&\det}<a,e>\times P_{ada}<a,f> \tag{11}$$

The minimum waiting time tolerated by the message on node *a*, $W_{\min}(a)$, to have at least one free virtual channel among all physical channels of dimensions still to be traversed may be treated as an *M/G/*1 queue. Therefore, as an example the waiting time on channel $<a,e>$ can be written as

$$W_{<a,e>} = \frac{\lambda_{<a,e>}.Ser_{<a,e>}^2.\left(1+\frac{(Ser_{<a,e>}-L)^2}{S_{<a,e>}^2}\right)}{2(1-\lambda_{<a,e>}.Ser_{<a,e>})} \tag{12}$$

in which $\lambda_{<a,e>}$ is the traffic rate on channel $<a,e>$, $Ser_{<a,e>}$ is the average service time of the given channel and $(Ser_{<a,e>}-M)^2$ is the variance of the service time distribution, as suggested in [12]. Therefore, the (dynamic) energy consumed due to message blocking on node *a* is given by

$$E_{dynamic}(a) = T_{blocking}(a)\times(dis(s,a)+1)\times E_{idle}\times B_{flit} \tag{13}$$

where B_{flit} is the size of a virtual channel of the router in flits.

The dynamic energy dissipated for the message which is transferred from a specific source node, *s*, and destined to another specific node, *d*, can be calculated as the aggregate of the energy consumed for all intermediate nodes located in at least one of the minimal paths between *s* and *d*. Thus,

$$E_{dynamic}(s,d) = \sum_{a \in G_{(s,d)}} T_{blocking}(a) \times (dis(s,a)+1) \times E_{idle} \times B_{flit}$$
$$G_{(s,d)} = \{a \mid dis(s,a) + dis(a,d) = dis(s,d)\} \tag{14}$$

Now taking into account both parts of energy dissipation (dynamic and static) per message transfer, the total energy consumed for the aforementioned message is

$$E_{total}(s,d) = E_{static}(s,d) + E_{dynamic}(s,d) \tag{15}$$

Since the mesh topology is not symmetric and the traffic is not evenly distributed over network channels, average energy consumption in the network can be achieved by averaging over all per message energy dissipation for all pairs of source and destination. Thus

$$\overline{E}_{total} = \frac{1}{N \cdot (N-1)} \cdot \sum_{(s,d) \in G \times G} E_{total}(s,d) \tag{16}$$

3 The Latency Model

When developing the latency model we use assumptions that have been widely used in similar modeling studies [12, 13, 14 and 15]. Messages are generated at each node according to a Poisson function with an average of λ_{node} messages per cycle. This study assumes that there is only a single hot IP in the network. The main reason behind this restriction is to keep the notation used for describing the model at a manageable level. Some directions are made to extend the model to deal with different nodes with different traffic generations.

The mean message latency is composed of the mean network latency, $\overline{S}$, that is the time to cross the network, and the mean waiting time seen by a message in the source node, W_s . However, to capture the effects of virtual channels multiplexing, the mean message latency has to be scaled by a factor, $\overline{V}$, representing the average degree of virtual channels multiplexing that takes place at a given physical channel. Therefore, we can write the mean message latency as [13, 14]

$$\mathrm{T} = (\overline{S} + W_s) \times \overline{V} \tag{17}$$

For each node like *i* in the network, we can define a set of probabilities $\{\alpha_{i,1}, \alpha_{i,2} \ldots \alpha_{i,i-1}, \alpha_{i,i+1} \ldots \alpha_{i,N}\}$ in which $\alpha_{i,j}$ is the probability that the generated message in node *i* is directed to node *j*. The similar set of probabilities, $\{\beta_{i,1}, \beta_{i,2} \ldots \beta_{i,i-1}, \beta_{i,i+1} \ldots \beta_{i,N}\}$, can be defined where $\beta_{i,j}$ is the probability that the received message in node *i* is sent from node *j*. The following formula shows the relation between these two probabilities

$$\beta_{j,i} = \alpha_{i,j} \cdot \prod_{x=1, x \neq i}^{N} (1 - \alpha_{x,j}) \tag{18}$$

In this study, for simplicity of the model, we assume that each generated message has a predefined probability α of being directed to the hot IP and probability $(1-\alpha)$ of being directed to the other network nodes (the destination node of a message is randomly chosen among other network nodes). Therefore, $\overline{S}$, which is the average network latency, can be computed as

$$\overline{S} = (1-\alpha)\cdot\overline{S_l} + \alpha\cdot\overline{S_h} \tag{19}$$

where $\overline{S_h}$ is the average network latency for the messages being directed to the hot IP and $\overline{S_l}$ is the network latency for other messages in the network. $\overline{S_l}$ can be determined by averaging the latency of messages generated at all possible source nodes, destined to all possible destination nodes except the hot node as destination. Therefore, we have

$$\overline{S_l} = \frac{1}{N\,(N-1)} \cdot \sum_{(s,d)\in G\times G,\, d\neq h} S_{(s,d)} \tag{20}$$

$\overline{S_h}$ is determined by averaging the latency of the messages whose destination is the hot node.

$$\overline{S_h} = \frac{1}{(N-1)} \sum_{s\in G,\, d=h} S_{(s,d)} \tag{21}$$

where $S_{(s,d)}$ is the network latency seen by a message originated from a specific source node, *s*, and destined to another specific node, *d*. Therefore,

$$S_{(s,d)} = dis(s,d)\cdot(t_r + t_s) + L\cdot t_s + \sum_{a\in G_{(s,d)}} T_{blocking}(a) \tag{22}$$

t_s is the transfer time of a flit between any two routers and t_r is the time in which the arbiter makes the decision to send the header to the desired output channel.

For a specific node *s* in the network, the average latency seen by a message originated at that node to enter the network, $\overline{S_s}$, is equal to the average of all $S_{(s,d)}$ for $d \in G - \{s\}$, resulting in

$$\overline{S_s} = \frac{1}{N-1}\left((1-\alpha) \sum_{d\in G-\{s,h\}} S_{(s,d)} + \alpha\cdot\sum_{d=h} S_{(s,d)} \right) \tag{23}$$

The mean waiting time in the source node is calculated in a similar way to that for a network channel. A message in the source node can enter the network through any of the *V* virtual channels. Therefore, modeling the local queue in the source node as an *M/G/*1 queue with the mean arrival rate λ_{node}/V and service time $\overline{S_s}$ with an approximated variance $\left(\overline{S_s} - L\right)^2$ yields the mean waiting time as [12]

$$W_s = \frac{(\lambda_{node}/V)\cdot \overline{S_s}^2 \left(1+\left(\frac{\overline{S_s}^2 - L}{\overline{S_s}}\right)^2\right)}{2\left(1-(\lambda_{node}/V)\cdot \overline{S_s}\right)} \tag{24}$$

The probability $P_{v,<a,e>}$ that v virtual channels are busy at a specific physical channel $<a,e>$ can be determined by a Markovian model as shown in [15]. That is

$$P_{v,j} = \begin{cases} \frac{1}{\sum_{i=0}^{V} Q_{i,<a,e>}} & \text{v} = 0 \\ P_{0,j}\cdot Q_{v,<a,e>} & 1 \le v \le V-1 \end{cases} \qquad Q_{v,j} = \begin{cases} \left(\lambda_{<a,e>}\cdot Ser_{<a,e>}\right)^v & 0\le v\le V-1 \\ \frac{\left(\lambda_{<a,e>}\cdot Ser_{<a,e>}\right)^v}{1\text{-}\lambda_{<a,e>}\cdot Ser_{<a,e>}} & v=V \end{cases} \tag{25}$$

When multiple virtual channels are used per physical channel, they share the bandwidth in a time-*multiplexed* manner. The average degree of virtual channel multiplexing for a given physical channel $<a,e>$ and (by averaging over all channel) a typical physical channel can be estimated by [15]

$$\overline{V}_{<a,e>} = \frac{\sum_{v=1}^{V} v^2\cdot P_{v,<a,e>}}{\sum_{v=1}^{V} v\cdot P_{v,<a,e>}} \quad \text{and} \quad \overline{V} = \frac{\sum_{<a,e>\in \text{ all channels}} \overline{V}_{<a,e>}}{2N\,(N-1)} \tag{26}$$

Since the dimension-ordered routing needs only one virtual channel to overcome deadlock occurrence in the network, by considering V virtual channels per physical channel, $V-1$ virtual channels are assigned to adaptive routing function. Thus, $P_{ada} <a,e>$ can be written as

$$P_{ada} <a,e> = P_{v,<a,e>} + \frac{P_{v-1,<a,e>}}{\binom{V}{V-1}} \tag{27}$$

and $P_{ada\&\text{det}} <a,e>$ is equal to $P_{v,<a,e>}$. The final two parameters which are skipped in both power and latency models are arrival traffic rate on the channel, $\lambda_{<a,e>}$, and average service time of the channel, $Ser_{<a,e>}$. The rate of messages generated at a specific node, *s*, and destined to another specific node, *d*, that traverse a specific channel $<a,e>$ on its minimal path, is calculated as product of the probability that the message crosses the given channel, $P_{(s,d)}^{pass} <a,e>$, and the probability that the message is sent to the node *d*, and the probability that the generating message is of the desired type. We divide the messages into three types. Messages originated from the hot IP, messages destined to the hot IP and all the other messages. Thus

$$\begin{gathered} \left\{\lambda_{(s,d),<a,e>}\right\}_{s\neq h,d\neq h} = (1-\alpha)\cdot\lambda_{node}\cdot P_{(S,D)}^{Pass} <a,e>\cdot\frac{1}{N-2} \\ \left\{\lambda_{(s,d),<a,e>}\right\}_{s=h} = (1-\alpha)\cdot\lambda_{node}\cdot P_{(S,D)}^{Pass} <a,e>\cdot\frac{1}{N-1} \\ \left\{\lambda_{(s,d),<a,e>}\right\}_{d=h} = \alpha\cdot\lambda_{node}\cdot P_{(S,D)}^{Pass} <a,e> \end{gathered} \tag{28}$$

Message arrival rate for a specific channel can be calculated as the aggregate of the rates in which each type of the messages generated at the given channel.

$$\lambda_{<a,e>} = \sum_{(s,d)\in G\times G} \lambda_{(s,d),<a,e>} \tag{29}$$

Since the message lengths are assumed to be greater than the network diameter, the service time of a channel is equal to the average of network latencies of all messages crossing the given channel. Therefore, the service time on channel <*a, e*> can be calculated as the average of the $S_{(s,d)}$ of all source and destination nodes that have at least one path between each other that traverse channel $< a, e >$ as

$$Ser_{l,<a,e>} = \sum_{d\neq h} S_{(s,d),<a,e>}, \quad Ser_{h,<a,e>} = \sum_{d=h} S_{(s,d),<a,e>} \tag{30}$$

Therefore, the mean service time at the channel, considering all types of messages with their appropriate weights, can be written as

$$Ser_{<a,e>} = \frac{\sum_{d\neq h} \lambda_{(s,d),<a,e>}}{\lambda_{<a,e>}} \times S_{l,<a,e>} + \frac{\sum_{d=h} \lambda_{(s,d),<a,e>}}{\lambda_{<a,e>}} \times S_{h,<a,e>} \tag{31}$$

4 Analytical Comparison

In this section, we apply the proposed model to a two-dimensional mesh ($Mesh_{5\times 5}$) to which the tile the hot IP is mapped varies around all possible nodes (quadrant of mesh). Figure 3 shows the region where the mapping is done. Note that due to the partial symmetry in the mesh topology, all other nodes are similar to one of the

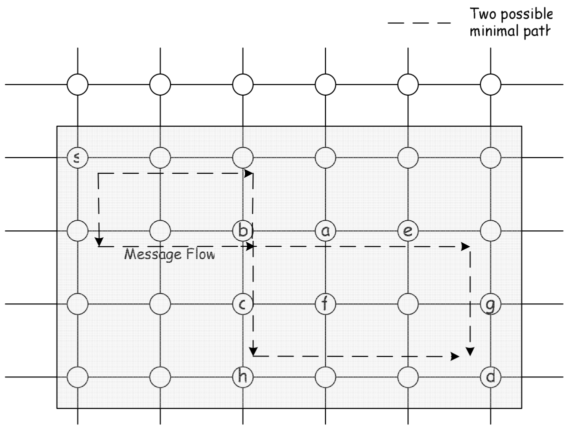


Fig. 2. The message may emerge at each node and channel in the shaded area with different probabilities

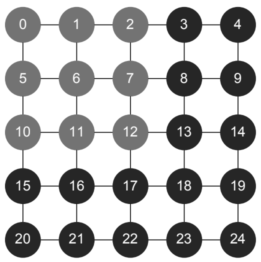
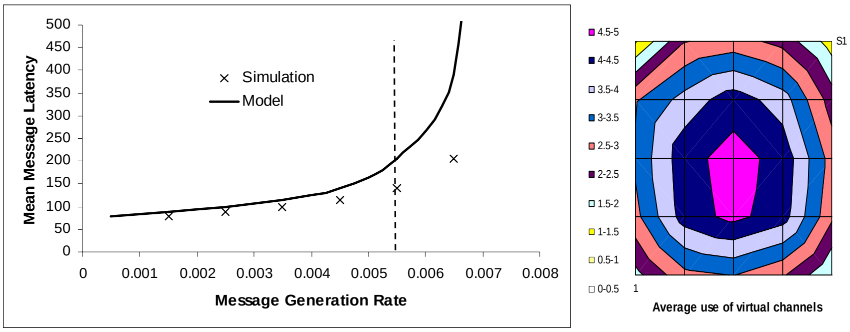
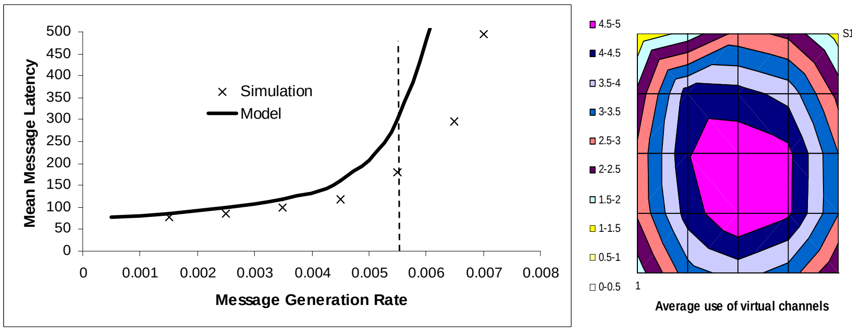


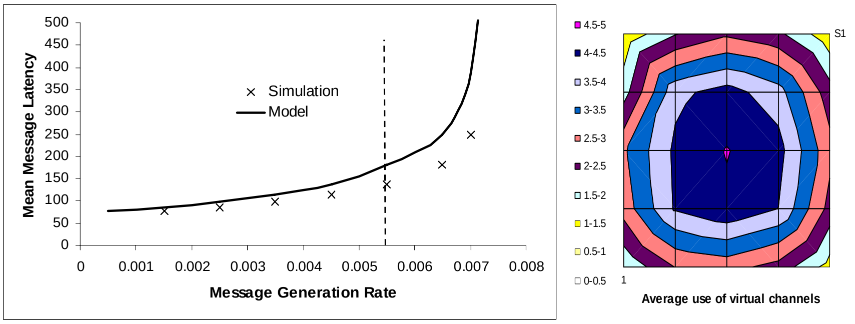
Fig. 3. A 5x5 Mesh



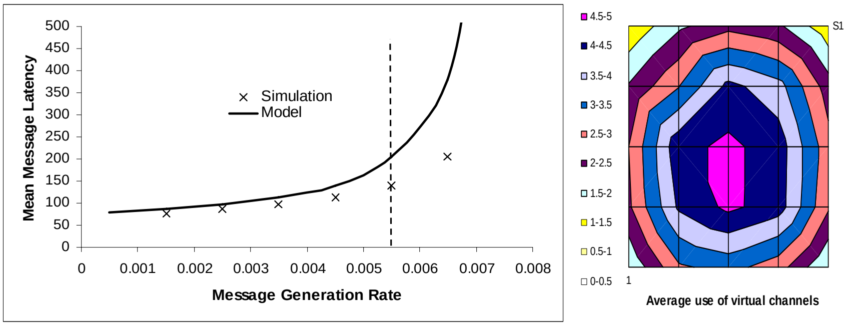
0



1

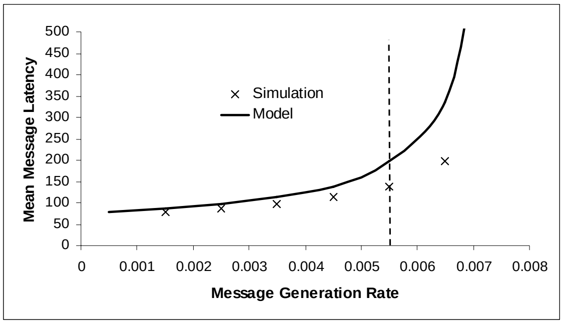


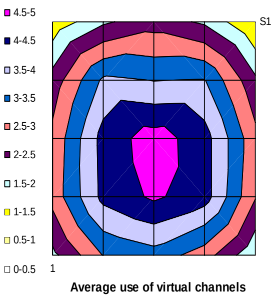
5



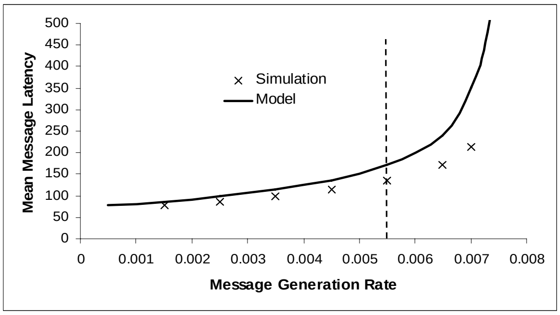
6

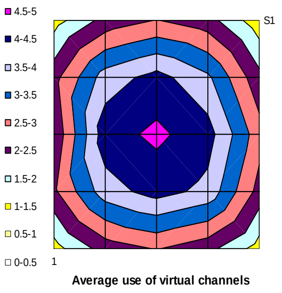
Fig. 4. Mapping the hot IP to nodes 0, 1, 5, 6, 10, 11, and 12; (left) Model validation; (right) Average use of virtual channels on 2-dimensional surface



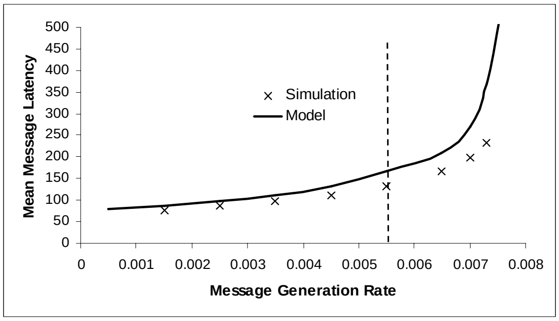


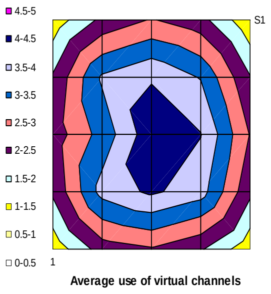
7



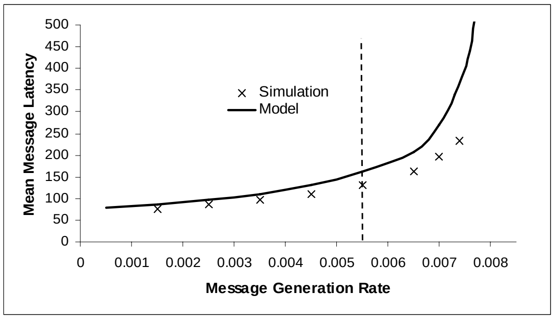


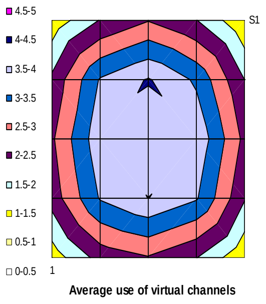
10





11





12

Fig. 4. (*continued*)

nodes considered in the gray sub-mesh. The analytical model is validated through a discrete-event simulator that mimics the behavior of both the fully adaptive routing and dimension-ordered routing at the flit level in two-dimensional meshes. In each simulation experiment, a total number of 100,000 messages were delivered. Statistical results gathered for the first 10,000 messages were thrown off to avoid distortion due to initial start-up conditions. Numerous scenarios were considered and simulated and analyzed but, for the sake of brevity, figure 4 depicts the latency results predicted by the model explained in the previous section against those provided by the simulator for a 5x5 mesh with message length of 64 flits and 2 virtual channels per physical channel only. We assume that each generated message has a finite probability 0.1 ($\alpha = 0.1$) of being directed to the hot IP and probability 0.9 of being directed to the other network nodes. The horizontal axis in the figures shows the traffic generation rate at each node, while the vertical axis shows the mean message latency. These figures reveal that the analytical model predicts the mean message latency within acceptable accuracy in the steady-state regions. Also the traffic rate on the channels in the whole network is presented in figure 4 (right). The traffic rate is the average use of virtual channels by the messages and λ_{node} in this case was fixed to 0.0055 in figure 4.

5 Conclusion

In this study, we proposed some analytical model that considers inter-relations between power consumption and network latency. The important findings of this study are as follows: I) The way we modeled the parameters and elements in NoC leads the reader to analyzing the effects of these parameters on power and latency of message; therefore, making a good view in proposing heuristic algorithms to find the best IPs/Cores mapping in NoC. II) As discussed in section 3, the model can be extended to deal with the uniform and other non-uniform traffic patterns. Also, the model does not depend on any routing algorithms. Therefore, omitting these constraints makes the model suitable to be used for any routings, traffic patterns, and topologies. III) To our best knowledge, no power model has been reported for NoC where the effect of blocking of the messages in the network is considered. This work is the first to do so. IV) The final result of the model which predicts the average energy consumption per cycle in the whole network is the valuable parameter can be used by network designers.

References

1. Chandrakasan, A., Brodersen, R.: Minimizing power consumption in digital CMOS circuits. Proc. IEEE 83, 498–523 (1995)
2. Chandrakasan, A.P., Sheng, S., Brodersen, R.W.: Low power CMOS digital design. IEEE J. Solid-State Circuits 27, 473–484 (1992)
3. Hu, J., Marculescu, R.: Energy-aware mapping for Tile-based NOC architectures under performance constraints. Design Automation (2003)
4. Hu, J., Marculescu, R.: Exploiting the routing flexibility for energy/performance aware mapping of regular NoC architectures. Design Automation, and Test 1 (2004)

5. Murali, S., De Micheli, G.: Bandwidth-constrained mapping of cores onto NoC architectures. Design, Automation and Test 2, 896–901 (2004)
6. Lei, T., Kumar, S.: A two-step genetic algorithm for mapping task graphs to a network on chip architecture. Digital System Design, 180–187 (2003)
7. Wang, H.-S., Zhu, X., Peh, L.-S., Malik, S.: Orion: A power-performance simulator for interconnection networks. Microarchitecture, 294–305 (2002)
8. Wang, H.-S., Peh, L.-S., Malik, S.: Power-driven design of router microarchitectures in on-chip networks. Microarchitecture, 105–116 (2003)
9. Chien, A.A.: A cost and performance model for k-ary n-cubes wormhole routers. IEEE Trans. Parallel and Distributed Systems 9, 150–162 (1998)
10. Wang, H.S., Peh, L.S., Malik, S.: A power model for routers: modeling Alpha 21364 and InfiniBand routers. High Performance Interconnects, 21–27 (2002)
11. Mamidipaka, M., Khouri, K., Dutt, N., Abadir, M.: Leakage power estimation in SRAMs. CECS Technical Report, Center for Embedded Computing Systems, University of California, Irvine, CA (2003)
12. Draper, J.T., Ghosh, J.: A comprehensive analytical model for wormhole routing in multicomputer systems. J. Parallel and Distributed Computing 23, 202–214 (1994)
13. Sarbazi-azad, H., Ould-Khaoua, M., Mackenzie, L.M.: Analytical modeling of wormhole-routed k-ary n-cubes in the presence of hot-spot traffic. IEEE Trans. Parallel and Distributed Systems 50, 623–634 (2001)
14. Sarbazi-azad, H.: Performance analysis of wormhole routing in multicomputer interconnection networks. PhD Thesis, University of Glasgow, Glasgow, UK (2002)
15. Dally, W.J.: Virtual channel flow control. IEEE Trans. Parallel and Distributed Systems 3, 194–205 (1992)

An Efficient Link Controller for Test Access to IP Core-Based Embedded System Chips

Jaehoon Song, Hyunbean Yi, Juhee Han, and Sungju Park

Department of Computer Science & Engineering, Hanyang University, Korea
{jhsong,bean,hanjh,sjpark}@mslab.hanyang.ac.kr

Abstract. It becomes crucial to test and verify embedded hardware systems precisely and efficiently. For an embedded System-on-a-Chip (SoC) comprised of multiple IP cores, various design techniques have been proposed to provide diverse test access link configurations. In this paper, a Flag-based Wrapped Core Link Controller (FWCLC) is introduced to enable efficient accessibility to embedded cores as well as seamless integration of IEEE 1149.1 TAP'd cores and IEEE 1500 wrapped cores. Compared with other state-of-the-art techniques, our technique requires no modification on each core, less area overhead, and provides more diverse link configurations for design-for-debug as well as design-for-test.

Keywords: Embedded System, Boundary Scan, SoC Testing, Test Access Mechanism, Wrapper.

1 Introduction

Embedded systems are increasingly designed by using System-on-a-Chip which embed reusable IP cores such as processors, memories, and peripheral interfaces. Today's system on boards become tomorrow's IC's, whereby today's IC's become tomorrow's cores. The major bottleneck in SoC design embedding multiple IP cores is the testing and debugging, and then, efficient test access architecture for IP cores in an SoC has become the key challenge in narrowing the gap between design and manufacturing capability.

Test access architecture for testing and debugging consists of mainly three components. Two of them are test access mechanisms (TAMs) and test wrappers, and have been proposed as important components of SoC test access architecture [1]–[8], [14], [16]–[18]. TAMs include internal scan chains which mainly bring the test sequences into embedded cores for internal testing, while test wrappers such as IEEE 1149.1 and 1500 translate the test sequence into test patterns for either internal testing of a target core or external testing of interconnect nets. The third key component is the link controller which controls the test access architecture for linkage between embedded cores and the external test equipment [9], [10], [12], [13], [19]. The test cost, which can be estimated by the memory depth required on the ATE as well as by the test application time, is greatly affected by the integration of well designed TAMs, test wrappers, and the link controller into the overall design. Therefore, The design of efficient test access architectures, which includes the integration of TAMs, test

© Springer-Verlag Berlin Heidelberg 2007

wrappers and the link controller for IEEE 1149.1 TAP'd and 1500 wrapped cores, have become significant issues in core test integration [8]–[13], [15], [19]. This paper is focused on introducing a new link controller which can be effectively adopted for an SoC embedding IP cores with wrappers such as IEEE 1149.1 or 1500.

When one or more cores have built-in 1149.1-compliant TAPs and are used in an 1149.1 compliant IC [7], like the TMS437 microcontroller manufactured by TI, a hierarchical access of embedded TAP problem arises, since the 1149.1 standard provides for only one set of test pins for a 1149.1-compliant IC [13]. The hierarchical access covers the access to the embedded cores which are hierarchically embedded at the core level as well as embedded in the SoC level. In order to solve the problem in accessing multiple cores, several systematic approaches have been proposed [9]-[13], [15], [19]. To overcome the deficiency in hierarchical access of IEEE 1500 cores with 1149.1 control, a unified test architecture of enhancing IEEE 1500 Serial Interface Layer (SIL) has been proposed [15]. This paper introduces an efficient Flag-based Wrapped Core Link Controller (FWCLC) which can provide hierarchical integration of IEEE 1149.1 and 1500 wrapped cores, completely compatible with the standards.

This paper is organized as follows. Various TAP linking techniques for the 1149.1-compliant cores are reviewed in section 2, and linking techniques including IEEE 1500 wrapped cores are described in section 3. A proposed wrapped core link controller is described in section 4 and design results are reported in section 5, followed by the conclusion in section 6.

2 Link Controllers for IEEE 1149.1

IEEE 1149.1 boundary scan is a design for testability technique to simplify the application of test patterns at the board or system levels. The standard Test Access Port (TAP) includes TDI, TDO, TMS, TCK and optionally TRST [7]. The standard includes the boundary scan, bypass or other test data register which are connected to the TDI-TDO path upon the instruction decoded by instruction decoder of IEEE 1149.1. Mandatory instructions include bypass, sample/preload, extest, and other optional user defined instructions include runbist, clamp, highz, and idcode.

Although the IEEE boundary scan was initially intended for board and system level testing, recently the SoCs comprised of reusable IP cores are tested and debugged through the boundary scan chains [9], [10], [12]-[19]. As a result various methods have emerged [9]-[19].

Reference [10] introduced the technique that the least significant bits of the SoC boundary scan instructions are used to differentiate the SoC TAP from core TAPs. A serial connection of 1149.1-compliant embedded cores to the SoC TAP as the test data register could potentially make it possible to debug a certain processor core, but every core must be in bypass mode except the target one, that is all the cores must be orchestrated simultaneously. The main objective of [10]'s approach was to debug a certain processor core by disconnecting the SoC TAP and setting all other cores to bypass mode but the one being debugged, which makes the connection between the ICBSR and CBSRs untestable.

In [9] the technique based on the TAP Linking Module (TLM) is presented in order to conveniently link the SoC TAP to any subset of embedded core TAPs, but standard IEEE 1149.1 boundary scan of both SoC and embedded cores must be slightly modified to add extra test logic which may not be allowed for the hard IP cores (DSP, CPU etc.).

The Hierarchical TAP (HTAP) architecture was developed to keep the IEEE 1149.1 compatibility and scalability [12], [13]. The HTAP architecture requires an augmented version of the 1149.1 TAP called a Snoopy TAP (SNTAP) in the top-level of the design hierarchy. The HTAP has all the advantages that the TLM has over the ad-hoc approaches but unlike the TLM does not need a second TAP controller, which leads to some hardware savings. However, this approach has two key weak points: one is that entrance to and wake-up from Snoopy-State is not seamless in a point of IEEE 1149.1 protocol, and the other is that the interconnects between the SoC and the core boundaries are not testable.

3 Link Controller for IEEE 1149.1 and 1500 Wrapped Cores

IEEE 1500 is the standard for testing embedded cores while preserving scalability for hierarchical test access. IEEE 1500 provides a flexible hardware interface between an embedded core and its environment so that predefined test patterns can be efficiently delivered to and from the embedded core. The core test wrapper standardized by IEEE 1500 has the following features [8].

- Provide core test, interconnect test and bypass modes which are a subset of IEEE 1149.1 modes.
- Can connect the core boundary chain (wrapper) to any internal scan chain to perform internal testing of the cores.
- The various modes of the core test wrapper are operated by several control signals in general generated through SoC TAP controller.

The IEEE 1500 architecture consists of wrapper register, TAM connection, instruction register, and control signals provided externally. The control signals include UpdateWR, CaptureWR, ShiftWR, SelectWIR, WRSTN and WRCK which is the test clock.

Like the boundary scan description language (BSDL) for IEEE 1149.x, the core test description language (CTL) is defined to transfer the test information from core provider to core user. Test-related information includes test methods, test modes, test patterns, design-for-test(and debug) information, etc.. A few core link techniques have been published for an SoC with IEEE 1149.1 cores [9], [10], [12]-[19]. The hierarchical Test Access Mechanism for SoCs with both IEEE 1149.1 and 1500 cores [19] was presented to achieve complete compatibility without any additional test pin, instead, the IEEE 1149.1 TRST pin is used to change the configuration. However, in order to adopt this technique, all the hierarchical cores must possess the optional

TRST pin, and by activating the TRST although the configuration mode can be changed but the TAP controller is transited to the Test Reset state, thus concurrent testing which is very important to reduce testing time of multiple cores is not feasible.

In this paper a simple Flag-based Wrapped Core Link Controller (WCLC) is proposed that can be hierarchically and concurrently applicable to an SoC with both IEEE 1149.1 and 1500 wrapped cores without requiring any additional pins and IP core modification.

4 Architecture of Flag-Based Wrapped Core Link Controller

4.1 Functional and Structural View

Fig. 1 shows our new test access architecture with FWCLC. In order to provide hierarchical test access to embedded cores, our Flag-based Wrapped Core Link Controller (FWCLC) in Fig. 1 coordinates the connection between the test bus and wrapped cores.

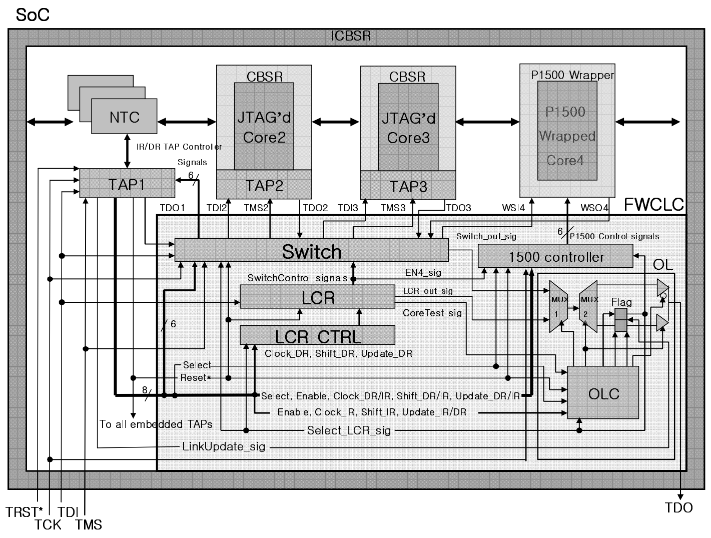


Fig. 1. Detailed view of FWCLC

Observation 1**:** To test concurrently the SoC including hierarchical cores while preserving compatibility with IEEE 1149.1 and 1500 standards, no additional pin is allowed to change the configuration mode in choosing the target cores.

Justification**:** Only five pins of TDI, TDO, TMS, TCK and TRST are allowed in any level of the SoC hierarchy to keep the IEEE 1149.1 standard, hence no extra test pin can be adopted. If TRST is chosen to change the configuration mode, the cores tested in different mode have to be reset upon activation of TRST making concurrent testing unfeasible.

***Observation 2*:** Only by taking a dedicated instruction for hierarchical core test and an internal flag register for reconfiguration, can concurrency and compatibility in hierarchical core testing be achieved.

***Justification*:** Since no additional test pin is allowed, an extra instruction must be used to get into core testing mode. Without interrupting the cores being tested, if other target cores need to be tested, the configuration has to be changed while still sustaining the current configuration. Hence, the flag identifying the reconfiguration mode is required. The flag register is then set or reset by Update_IR state after shift-IR of IEEE 1149.1 protocol.

In Fig. 1 TAP1 for the SoC allows its boundary scan register to access the TAP'd, the non-TAP'd, and the IEEE 1500 wrapped cores. TAP2 and TAP3 are associated with each IP core to access the corresponding CBSR. However, IEEE 1500 wrapped core4 is accessed via the IEEE 1500 controller in the FWCLC. The multiple TAPs and IEEE 1500 wrappers are connected to 1149.1-compliant SoC test bus through the FWCLC.

Switch module, Link Control Register (LCR) module, LCR Controller (LCR_CTRL) module, IEEE 1500 controller module, and Output Logic (OL) constitute the basic framework of our test access mechanism. A TCK signal is directly connected to each TAP and FWCLC. The key components of this architecture are described as follows:

1) Switch

The switch is comprised of a crossbar switch used to set the test path, a gating circuit to either select or de-select SoC TMS signal and SoC TAP control signals, and a circuit to initialize the core TAP without TRST*. The key functions include:

a) Initialize the embedded cores upon the SoC TAP initialization, and then only the SoC TAP is located on the TDI-TDO path.
b) Based on the Link Control Register (LCR) information, the switch provides the embedded core test path from TDI to TDO of the SoC.
c) When the core test path is reconfigured upon the LCR, current wrappers and instructions are guaranteed to keep the current values. To perform the concurrent testing with precedence constraints [4], this function is crucial.

2) Link Control Register (LCR)

The link information among embedded cores including SoC TAP is stored in this LCR. In order to change the link, the LCR must be located on the SoC TDI-TDO scan path, and the LCR information is changed at Update-DR after Shift-DR state according to IEEE 1149.1 protocol. The CoreTest_sig of the LCR indicates whether any embedded core is involved or just the TAP1 which is the SoC TAP. The signal controls the flag register in the OL block. For example, if any embedded core is not selected upon LCR, then the CoreTest_sig is set to ‘0’, telling not in core test mode. This condition can occur in the case of power on, power reset, or the start/completion of test by a test scenario.

The LCR is activated when the Select_LCR_sig is ‘1’ which is triggered by the flag register, and shifts the link information during Shift-DR state.

3) Output Logic (OL)
Fig. 2 shows the structure of the OL which connects either wrapped core test path or LCR to TDO output. Either Switch_out_sig for the wrapped core test path or LCR_out_sig for the LCR is connected to TDO.

Also included is a 1-bit flag register which commands the test controller to change the link configuration. This flag register is attached to the last of the instruction registers during Shift-IR and Update-IR states in the core test mode which is indicated by the CoreTest_sig. Therefore, the value in the register is shifted and updated at Shift-IR and Update-IR states, respectively. The value of '1' at this flag register indicates that the LCR is on the TDI-TDO path by setting Select_LCR_sig to '1'.

When CoreTest_sig is set to '0', the signal de-activates the flag register by gating the control signals for the register, and it is not on the TDI-TDO path. In this case, the flag register can be set to '1' by LinkUpdate_sig from the SoC TAP, and this signal is triggered by the additional CoreTestMode instruction of the SoC TAP.

Table 1 shows the TDO output value on the CoreTest_sig, Select_LCR_sig and TAP controller states.

4) IEEE 1500 Controller
An IEEE 1500 controller is shown in detail in Fig. 3 where each output signal used to control IEEE 1500 wrapper of core 4 is directly connected to IEEE 1500 Wrapper Serial Ports (WSP) except the Wrapper Serial Input (WSI) 4 and Wrapper Serial Output (WSO) 4. The IEEE 1500 controller consists of the Wrapper Enable Logic (WEL) associated with each IEEE 1500 wrapped core, and a multiplexer(MUX).

The WEL gates control signals of the SoC TAP and applies them to the WSP (other than the WSI and WSO). The MUX is controlled by the Select signal of the SoC TAP controller which is the TAP1. In the DR-Scan state, DR control signals are connected to the output of IEEE 1500 controller, and in the IR-Scan state, IR control signals are connected.

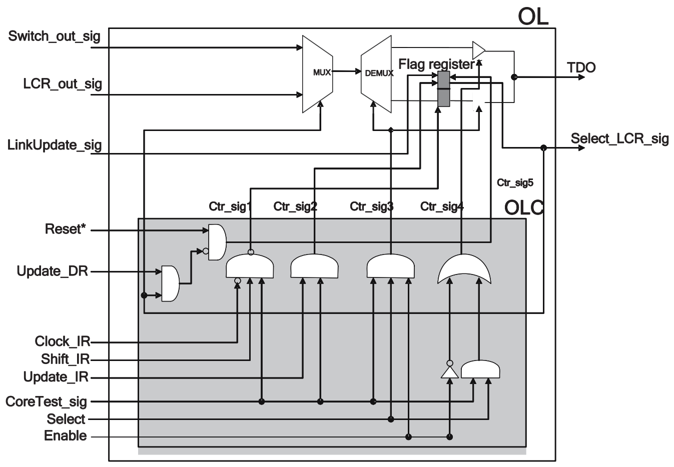


Fig. 2. Output Logic (OL) structure

Table 1. Output of TDO by key control signals

Control signal		TAP controller states	TDO output signal
CoreTest_sig	Select_LCR_sig		
0	0	Scan-DR	Switch_out_sig
0	0	Scan-IR	Switch_out_sig
0	0	Other than Scan-DR and Scan-IR	HighZ
0	1	Scan-DR	LCR_out_sig
0	1	Other than Scan-DR	HighZ
1	0	Scan-DR	Switch_out_sig
1	0	Scan-IR	Switch_out_sig
1	0	Other than Scan-DR and Scan-IR	HighZ
1	1	Scan-DR	LCR_out_sig
1	1	Other than Scan-DR	HighZ

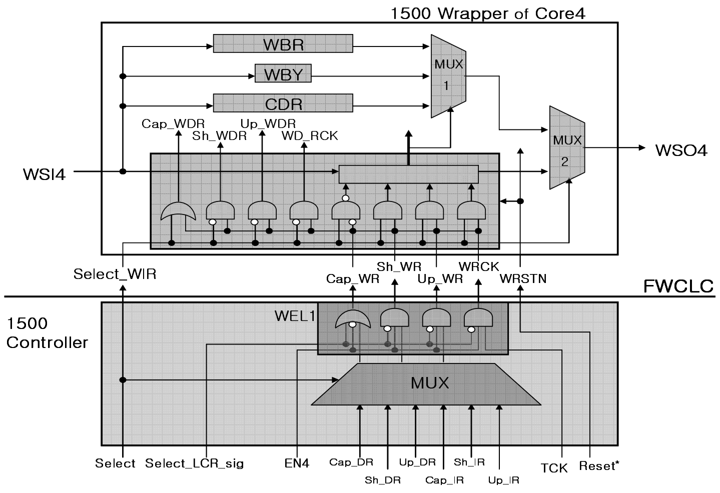


Fig. 3. Detailed view of IEEE 1500 controller

4.2 Operation and Timing

Upon power on or test reset, the FWCLC is designed to enable only the SoC TAP by setting the leftmost bit of the LCR to ‘1’. Thus, only the SoC TAP is placed on the TDI-TDO path, while all wrapped cores are disconnected from the test bus.

The SoC TAP controller supervising FWCLC is always activated to track the test bus status although the controller is disabled under LCR information.

In this architecture, the key signal of the FWCLC is the Select_LCR_sig triggered by the flag register whose value must be provided during Shift-IR state and updated at Update-DR state. Upon receipt of the Select_LCR_sig from the flag register, the LCR of the FWCLC is connected to TDI-TDO path. Then, the link information regarding the TAPed and IEEE 1500 wrapped cores selected is shifted into the LCR through the TDI of the SoC. The link configuration of wrapped cores is updated at the Update-DR state after new link information is shifted in.

The timing diagram which shows the transfer of control from TAP2 to TAP3, is illustrated in Fig. 4. The TAP2 initially linked becomes disconnected and the TAP3 unlinked is getting linked at Update-DR state. FWCLC drives the synchronous transition of all wrapped cores, and allows linking/unlinking operations to occur only at the “Update-DR” state and all unlinked wrapped cores to enter the “RunTest/Idle” state.

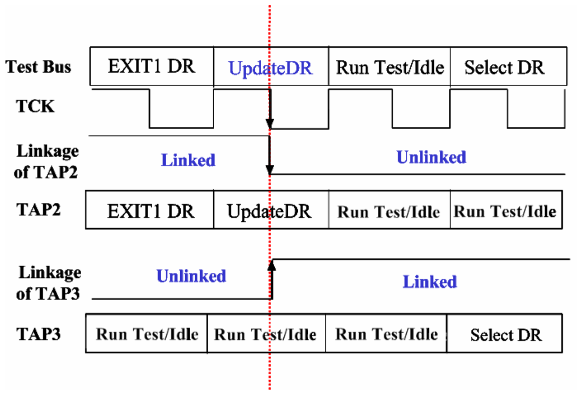


Fig. 4. Sample timing diagram of link change using FWCLC

5 Design Results for FWCLC

The key differences between the approaches cited in [9], [10], [12], [13], [19] and our approach regarding the FWCLC method are summarized in Table 2. It should be noted that TAP_EN and SEL pins are necessary for the [9]’s TLM requiring TAPed core modification. The HTAP architecture in [12], [13] also has key deficiencies in that entrance to and wake-up from Snoopy-State is not seamless in a point of IEEE 1149.1 protocl, and the connections among chip and core boundaries can not be tested. The TAP controller in the SNTAP is a modified 1149.1-compliant TAP controller. In addition to the 16 states necessary to perform all 1149.1-specified functions, this controller has an extra set of 16 states designated "Snoopy States" which are used to snoop on the test bus of an IC, when appropriate. However, our method does not require such complicated extra states, and instead, makes use of the 1149.1 standard only. Particularly, our architecture can support IEEE 1500 wrapped cores as well as 1149.1 cores. To reduce SoC test time with various constraints such as available TAM width, test pins and power dissipation [4], the IP cores must be dynamically reconfigured without interrupting the current testing and while preserving the current status by a core link controller to maximize test concurrency. This dynamic link configuration capability is provided by our FWCLC, and the effectiveness is demonstrated, experimentally.

The key link control blocks for each technique are listed in Table 3 along with the area overhead estimated. Each design was synthesized by Synopsys Design Compiler with TSMC 0.25 CMOS technology, and the gate counts are presented as the number of equivalent two-input NAND gates in the last column of Table 3. It can be seen that

Table 2. Key differences between prior techniques and our method

	Link config.	TAP'd core modification	Chip TAP controller modification	Dynamic link config.	IEEE 1500 support
[9]	any	required	required	support	not support
[10]	serial	not required	not required	not support	not support
[12]	among cores	not required	required	support	not support
[13]	among cores	not required	required	support	not support
[19]	among cores	not required	not required	not support	support
Proposed	Any	not required	not required	support	support

Table 3. Estimated area overhead of the prior techniques and our method

	Key link blocks	Area estimated (NAND gates)
[9]	TLM TAP controller, LinkUpdate register, Decoder, Shift register	1104
[10]	Lengthy SoC, Instruction register	Size of the instruction register
[12]	16 states machine, Control register, Programmable switch	940
[19]	FSM block, Switch block, Wrapper control block, Configuration register, Boundary Reg. control block	1200
Proposed	LinkControl register, Switch, Output Logic	908

FWCLC requires much less area overhead than the [9], [12] and [19]. Although [10] requires the least area overhead, diverse and dynamic link configurations are not achievable. For these reasons, it is certain that our flag-based link update technique provides the most effective and convenient test access solution.

Our technique has been applied to an SoC in Fig. 5 which includes four IP cores, TAP controller and FWCLC. The characteristics of each IP core are described in Table 4, where "Required test clocks" indicates the number of test clocks needed to test each IP core with the test patterns generated by Synopsys TetraMAX. Scan

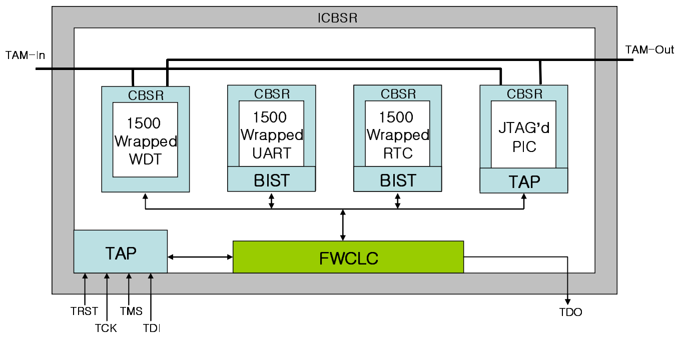


Fig. 5. Experimental chip

Table 4. Characteristics of the IP cores used in the experimental chip

	IP Cores			
	WDT	UART (BISTed)	RTC (BISTed)	PIC
Function	Watch Dog Timer	Universal Asynchronous Receiver Transmitter	Real Time Clock	Peripheral Interface Controller
Wrapper	IEEE 1500	IEEE 1500	IEEE 1500	JTAG
Required test clocks	837,480	7,339,976	2,359,260	728,820
Number of scan chains	10	-	-	10

Table 5. Effectiveness of concurrent testing for the different link configurations

		IP Cores				Total test clocks
		WDT	UART (BIST)	RTC (BIST)	PIC	
Test sequence cases	Case1	1C	2C	3C	4C	11,265,536
	Case2	1C	2	2	1C	8,177,406
	Case3	2C	1	1	2C	7,339,976
			3	3		

insertion for two IP cores, WDT and PIC, was done by Synopsys DFT compiler, and BIST logic was implanted in UART and RTC core by Mentor Graphics LBISTArchitect. In this experiment, two 10-bit parallel TAM buses were used as test bus in top level for TAM-In and TAM-Out.

To show the effectiveness of dynamic link configuration supported by the proposed technique, test patterns were applied sequentially and concurrently for different link configurations, and the results are described in Table 5. The total test time is the summation of the required test clocks for each core and the time taken to reconfigure the links. The number in each IP core denotes the order of reconfigurations, and the suffix ‘C’ implies that the corresponding core is on the test link path until the completion of the testing. For example, in case2 both WDT and PIC cores are initially chosen for internal testing, and after the completion of this testing, the BISTs for UART and RTC are activated. In case3, at the beginning, the BIST operations for UART and RTC cores are activated, and then the test path is reconfigured to choose WDT and PIC cores for internal testing while BISTs for UART and RTC are running excluded from the test path. After the completion of the internal testing for WDT and PIC cores, the link is reconfigured to include the UART and RTC on the test path, and then BIST results are scanned out after completion. In the sequel, the shortest testing time was achievable by concurrently testing the cores with our FWCLC.

To test an SoC embedding hierarchical cores which embed other cores in hierarchically lower level, the appropriate test access architecture must be provided. The following procedures show that our FWCLC can be efficiently used to test an SoC such as Fig. 6, which includes a hierarchical core (HCORE).

a) The LCR in chip level FWCLC is located on the TDI-TDO path.
b) The LCR is assigned with “0001” to include the HCORE on the test path.
c) The LCR in FWCLC of the HCORE is located on the TDI-TDO path.
d) The LCR is assigned with “001” to include the PIC core on the test path.
e) Finally the PIC core embedded in the HCORE can be accessed through the SoC level test pins.

In summary, the FWCLCs at the SoC and HCORE levels can provide diverse dynamic link configurations, concurrent testing and hierarchical test access for IP debugging as well as testing.

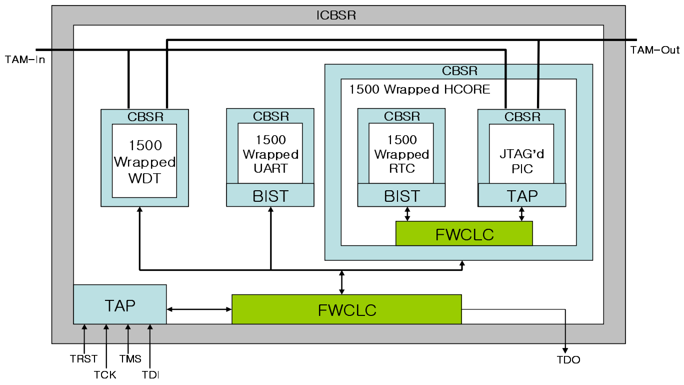


Fig. 6. Effectiveness of concurrent testing for the different link configurations

6 Conclusion

In order to verify and test embedded system chips with multiple IP cores, an efficient Flag-based Wrapped Core Link Controller (FWCLC) is introduced in this paper. An SoC embedding IEEE 1149.1 and 1500 wrapped cores can be tested concurrently and hierarchically with our FWCLC, which also allows diverse interconnect testing. Various test scheduling techniques requiring concurrent test for the optimization of test power, application time, and test pins can be efficiently implemented with this FWCLC. Therefore, complicated embedded system can be precisely and economically tested with our scheme.

Acknowledgemets

This work was supported by grant No. R01-2006-000-11038-0(2007) from the Basic Research Program of the Korea Science & Engineering Foundation and IT R&D Project funded by Korean Ministry of Information and Communications.

References

1. Marinissen, E.J., Zorian, Y., Kapur, R., Taylor, T., Whetsel, L.: Towards a standard for embedded core test: an example. In: Proc. IEEE Int. Test Conf., pp. 616–627. IEEE Computer Society Press, Los Alamitos (1999)
2. Zorian, Y.: System-chip test strategies. In: Proc. IEEE/ACM Design Automation Conf., pp. 752–757 (1998)
3. Zorian, Y., Marinissen, E.J., Dey, S.: Testing embedded-core based system chips. In: Proc. IEEE Int. Test Conf., pp. 130–143 (1998)
4. Iyengar, V., Chakrabarty, K.: Precedence-based, preemptive, and power-constrained test scheduling for system-on-a-chip. In: Proc. IEEE VLSI Test Symp., pp. 368–374. IEEE Computer Society Press, Los Alamitos (2001)
5. Marinissen, E.J., Dingemanse, H., Arendsen, R., Lousberg, M., Bos, G., Wouters, C.: A structured and scalable mechanism for test access to embedded reusable cores. In: Proc. IEEE Int. Test Conf., pp. 284–293. IEEE Computer Society Press, Los Alamitos (1998)
6. Iyengar, V., Chakrabarty, K., Marinissen, E.J.: Test wrapper and test access mechanism co-optimization for system-on-chip. In: Proc. IEEE Int. Test Conf., pp. 1023–1032. IEEE Computer Society Press, Los Alamitos (2001)
7. IEEE Std. 1149.1-2001
8. IEEE Std. 1500–2005
9. Whetsel, L.: An IEEE1149.1 based test access architecture for ICs with embedded cores. In: Proc. IEEE Int. Test Conf, pp. 69–78. IEEE Computer Society Press, Los Alamitos (1997)
10. Oakland, S.F.: Considerations for implementing IEEE1149.1 on system-on-a-chip integrated circuits. In: Proc. IEEE Int. Test Conf., pp. 628–637. IEEE Computer Society Press, Los Alamitos (2000)
11. Whetsel, L.: Addressable test ports an approach to testing embedded cores. In: Proc. IEEE Int. Test Conf., pp. 1055–1064. IEEE Computer Society Press, Los Alamitos (1999)
12. Bhattacharya, D.: Hierarchical test access architecture for embedded cores in an integrated circuit. In: Proc. IEEE VLSI Test Symp., pp. 8–14. IEEE Computer Scoiety Press, Los Alamitos (1998)
13. Bhattacharya, D.: Instruction-driven wake-up mechanisms for snoopy TAP controller. In: Proc. IEEE VLSI Test Symp., pp. 467–472. IEEE Computer Society Press, Los Alamitos (1999)
14. Harrison, S., Noeninckx, G., Horwood, P., Collins, P.: Hierachical boundary-scan a scan chip-set solution. In: Proc. IEEE Int. Test Conf., pp. 480–486. IEEE Computer Society Press, Los Alamitos (2001)
15. Dervisoglu, B.I.: A unified DFT architecture for use with IEEE 1149.1 and VSIA/IEEE P1500 compliant test access controllers. In: Proc. IEEE/ACM Design Automation Conf., pp. 53–58 (2001)
16. Jianhua, F., Jieyi, L., Wenhua, X., Hongfei, Y.: An improved test access mechanism structure and optimization technique in system-on-chip. In: Proc. Asia and South Pacific – Design Auto. Conf., pp. D/23-D/24 (2005)
17. Xu, Q., Nicolici, N.: Multifrequency TAM design for hierarchical SOCs. IEEE Trans on Comput.-Aided Des. Integr. Circuits Syst. 25(1), 181–196 (2006)
18. Sehgal, A., Chakrabarty, K.: Optimization of Dual-Speed TAM Architectures for Efficient Modular Testing of SOCs. IEEE Trans. on Comput. 56(1), 120–133 (2007)
19. Chou, C.W., Huang, J.R., Hsiao, M.J., Chang, T.Y.: A hierarchical test access mechanism for SoC and the automatic test development flow. In: Proc. IEEE Circuits and Systems Midwest Symp., pp. 164–167. IEEE Computer Society Press, Los Alamitos (2002)

Performance of Keyword Connection Algorithm in Nested Mobility Networks

Sang-Hoon Ryu and Doo-Kwon Baik

Software System Lab., Dept. of Computer Science and Engineering,
Korea University, 5 Anam-dong, Sungbuk-ku, Seoul, 136-701, Korea
boy1004@gmail.com,baik@software.korea.ac.kr

Abstract. As wireless techniques are developing, a mobile node has an ability to move everywhere. Some nodes become a group and transfer to other coverage together. Therefore, a mobile node has to identify where it is and establish the communication session rapidly, although it changes its point of attachment. We suggest Information-based mechanism and simulate it by NS-2. The result shows suggested mechanism reduces the handoff latency and end-to-end delay.

Keywords: Nested Mobile Networks, RS and RA Format, Information-Based Connection.

1 Introduction

As wireless techniques are developing, a mobile node has the ability to move everywhere. Therefore, a mobile node has to identify where it is and establish the communication session rapidly, although it changes its point of attachment. As a mobile node moves several times, the handoff latency and end-to-end delay get longer.

A mobile router (MR) or a mobile network node (MNN) changes it point of attachment, but there is a number of nodes behind the mobile router. The ultimate objective of a network mobility solution is to allow all nodes in the mobile network to be reachable via their permanent IP addresses, as well as maintain ongoing sessions when the mobile router changes its point of attachment within the Internet. Network mobility support should allow a mobile node or a mobile network to visit another mobile network. The mobile network is operated by a basic specification to support network mobility called Network Mobility (NEMO) Basic Support [1]. Furthermore, when a host has several IPv6 addresses to choose between, it is said multihomed [2]. This happens for instance when a mobile host or a mobile router has several interfaces simultaneously connected to the fixed network or when a mobile network has multiple MRs. However so many tunnels are used for supporting the sessions when a MR or MNN moves to other coverage. This is called pinball problem.

Some studies for avoiding the problem and for maintaining the session between a mobile network node (MNN) and a correspondent node (CM) have been

© Springer-Verlag Berlin Heidelberg 2007

researched. Most solutions proposed to enhance handoff in mobile IP environments by performing a pre-registratin of the MN with the new access router (AR) and avoid packet loss. A handoff scheme using bi-directional edge tunneling [3] is made even with the nested mobile networks. To avoid this problem, Mobile Ad-hoc Networks Extension [4] is proposed. However, we reduce the number of tunnels into one tunnel and make the MR or MNN choose the effective point of attachment among possible points of attachement.

In case of the nested NEMO, the more a mobile network node (MNN) moves to another area several times continually, the heavier the overhead and delay, Round Trip Time (RTT) and disconnection are. These have an effect on real time multimedia communication between a correspondent node (CN) and a MNN. Therefore, this paper proposes information procedure for guaranteeing QoS and reducing disconnection which occurs in nested mobile network.

This paper is organized as follows. Section 2 represents some essential technology required at suggested information-based connection mechanism. In section 3, we describe some problems in nested NEMO and propose the scheme of handoff delay by applying new mechanism. Section 4 shows the information-based connection mechanism. In section 5, we evaluate the suggested new mechanism accomplishing simulation by using NS-2. Lastly in section 6, we give some remarks with result of simulation.

2 Related Works

In this section, we explain underlying techniques of current issued network mobility and routing mechanism, and describe how we make use of them.

2.1 Network Mobility

Network mobility support is concerned with managing the mobility of an entire network. This arises when a router connecting a network to the Internet dynamically changes its point of attachment to the fixed infrastructure, thereby causing the reachability of the entire network to be changed in relation to the fixed Internet topology. Such a network is referred to as a mobile network.

Each AR has several Mobile Routers (MR) and Mobile Network Nodes (MNN). Those MRs and MNNs get into a group under the AR to the Internet and make one or more level of hops from the AR. All MRs and MNNs can freely move in their own direction. Therefore, the connectivity and reachability should be ensured in Network Mobility [5].

Once the binding process finishes, a bi-directional tunnel is established between the HA and the MR. The tunnel end points are the Mobile Routers Care-of Address and the HAs address. If a packet with a source address belonging to the Mobile Network Prefix is received from the Mobile Network, the MR returns the packet to the HA through this tunnel.

2.2 Nested Network Mobility

An entire network which moves as a unit dynamically changes its point of attachment to the Internet and thus its reachability in the topology. The mobile network is composed of one or more IP-subnets and is connected to the global Internet via one or more Mobile Routers (MR). The internal configuration of the mobile network is assumed to be relatively stable with respect to the [5].

Nested mobility occurs when there is more than one level of mobility. That is to say, a mobile network acts as an access network and allows visiting nodes to attach to it. This situation is called the nested NEMO. There are two cases of nested mobility. First, the attaching node is a single visiting mobile node (VMN). For instance, when a passenger carrying a mobile phone gets Internet access from the public access network deployed on a bus. Second, the attaching node is a MR with nodes behind a mobile.

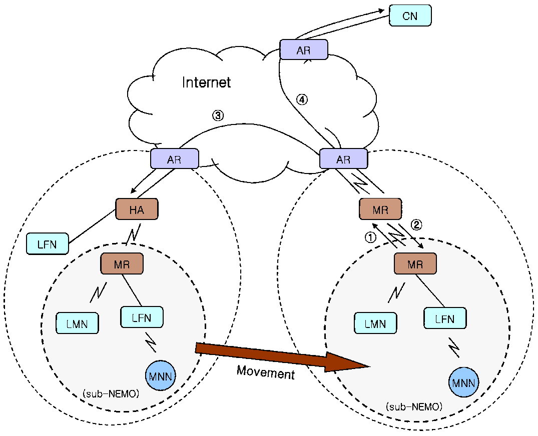


Fig. 1. A sub-NEMO moves into new coverage under another AR in Nested Network Mobility

Figure 1 shows that sub-NEMO which is a mobile network migrates into new coverage under neighboring AR. The sub-NEMO consists of several nodes and the all nodes in the sub-NEMO moves altogether. After the movement of the sub-NEMO, the procedure for receiving packets from the CN is very similar to that in section 2.1. However the MR should work for transmitting packets towards the nodes under itself by mapping old addresses to new addresses.

A mobile network is said to be nested when a mobile network (sub-NEMO) is attached to a larger mobile network (parent-NEMO). The aggregated hierarchy of mobile networks becomes a single nested mobile network. The root-NEMO is the mobile network at the top of the hierarchy connecting the aggregated nested mobile networks to the Internet. The parent-NEMO is the upstream mobile network providing Internet access to another mobile network further down the hierarchy. The sub-NEMO is the downstream mobile network attached to another mobile network up in the hierarchy. It becomes subservient of the parent-NEMO. The sub-NEMO is getting Internet access through the parent-NEMO and does not provide Internet access to the parent-NEMO. The root-MR is the MR(s) of the root-NEMO used to connect the nested mobile network to the fixed Internet. This is referred to as Top-Level Mobile Router (TMLR). The parent-MR is the MR(s) of the parent-NEMO. The sub-MR is the MR(s) of the sub-NEMO which is connected to a parent-NEMO [6][7].

2.3 Information Based Routing

Information based routing is with Beacon Routing[8] for a basis in order to transmit Smart Packet to destination with a high degree of efficiency. Beacons are peculiar active nodes to broadcast routing information for specific Smart Packets and Beacons also are operated as traditional router. On the average, active nodes are connected to one Beacon or more and Active Packets should be transmitted to target host based on the methods within the Smart Packets. With the intention of deciding routing path, Beacon broadcasts specific information and then sets up new link to a Beacon holding target host address.

In information based routing, active node can select adjacent beacon based on keyword to be transmitted to the beacon. Accordingly, this means beacon connected to active node is needless to be fixed and active node can change a beacon to be connected depending on a Smart Packet it receives. Since routing path which that active packets are going through is restricted within specific limit routing paths, it is so efficient that we can deliver our packets more selectively[9][10].

3 Route Optimization Based on Information

The Nested NEMO has a multi-level hierarchical architecture and this results in handoff latency and end-to-end delay by pinball problem. In other words, several tunnels are used in nested NEMO. Therefore, it is very important to reduce the number of tunnels that are formed when a MNN changes its point of attachment to the Internet.

First of all, BU message includes label option field and all MNN send BU message that contains a label to upper MRs.

3.1 Communication Between Different Parent MRs

Figure 2 shows the movement of a MNN to neighboring Sub-MRs. The movement is divided into movement1 and movement2. The MNN migrates between different Parent MRs in movement1. And in movement2 the MNN moves within the same Parent MR. The HA1, HA2, HA3 and HA4 are the HAs of MNN, Sub-MR1, MR and P-MR respectively.

First, we consider the movement1 case. When the MNN detects that the MNN moves into new coverage under other Sub-MR1, the MNN creates a label and insert the label into the label option of a BU message. And then the MNN sends the BU message to upper Sub-MR1. After the Sub-MR1 receives the BU message, the Sub-MR1 makes binding cache. In binding cache of the Sub-MR1, a row of table consists of label, home address and source address which means next hop. The label is used to make a link between the Sub-MR1 and the MNN. Thus, whenever the Sub-MR1 needs to forwards the packet destined to the MNN, the Sub-MR1 looks up at the binding cache based on the label for the MNN and sends the packets to recorded source address in the binding cache, because the link from the Sub-MR1 to the MNN was created for the MNN beforehand. And then the Sub-MR1 changes the source address with its own source address in the BU message. Furthermore, the Sub-MR1 does not need to encapsulate packets any longer. When the Sub-MR1 sends packets to the Internet, the Parent MR directly send to a HA1 of the MNN instead of a HA2 of Sub-MR1. That is because the label instead of the home address is used within a nested NEMO.

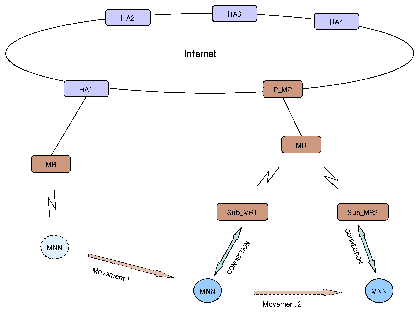


Fig. 2. The Movement of a MNN in Network Mobility

The Sub-MR1 gives the BU message to the upper MR and the binding cache is made in the MR. As a result, the MR makes a link from itself to the Sub-MR1 based on the label. Whenever the MR gets packets destined to the MNN, the MR sends packets to the Sub-MR1 according to the label of the MNN, home address of the MNN and source address of the Sub-MR1 which is used for next hop. And then the MR changes the source address with its own source address in the BU message. The MR does not encapsulate packets and send to a HA3 of the MR in the same way as the Sub-MR1.

Finally the P-MR receives the BU message and begins to make a binding cache that consists of the label, home address and source address. The label is used within the nested NEMO. The home address is the address of a MNN. This address is used to find HA1 of the MNN and creates a tunnel between the HA1 of the MNN and the P-MR. Besides, when the P-MR receives packets from the HA1, the P-MR looks up the next hop in binding cache. And then the P-MR sends packets to the Sub-MR1 because the link from the P-MR to the Sub-MR1 was created for the MNN in advance.

Consequently, all MRs except for P-MR does not need to capsulate packets because of the label usage, the only one tunnel is set up by P-MR. All MRs notice whether themselves are the Parent MR or not by means of the depth of RS and RA message that are proposed in section 3. This proposed procedure resolves the pinball problem that the nested NEMO suffer from.

3.2 Communication Within the Same Parent MR

We consider the movement2 case in Figure 2. The links through P-MR, MR, Sub-MR1 and MNN are made for packets destined to MNN using the label. In movement2, the MNN changes its point of attachment to Sub-MR2. Hence, the MNN need to make new link in order to receive packets at new location. The MNN sends a BU message with the label to the Sub-MR2 after giving the RS message and getting the RA message. The Sub-MR2 establishes a link from itself to the MNN by inserting new row with the Label, home address and source address in the binding cache after receiving the BU message from the MNN. And then the Sub-MR2 forwards the BU Message to the MR. Most of all, the MR examines whether the same label and home address exist in its own binding cache. If the same label exists in the binding cache, the MR swaps the existed home address and source address for new home address and source address in the same row. Thus new link from the MR to the Sub-MR2 is created and old link from the MR to the Sub-MR1 is broken for packets destined to the MNN. As a result of this procedure, the MR begins to forward all packets to the Sub-MR2 instead of the Sub-MR1 without the pinball problem.

3.3 Communication Through the Internet

The NEMO basic support has pinball problem using several IP-in-IP encapsulation. This causes end-to-end delay and high handoff latency. Therefore, the MNN can not receive packets from a CN continuously.

In figure 2, only the P-MR establishes a tunnel between itself and a HA1 of the MNN. In order to perform this procedure the P-MR searches for the home address of the MNN in the BU message from the MNN. And then the P-MR makes a tunnel with HA1 using the home address and its own address. Hence, the encapsulation is performed once and only one tunnel is set up. In the NEMO case, four tunnels are created. When HA1 sends packets to the MNN, the route is HA1, HA2, HA3, HA4, P-MR, MR, Sub-MR2, MNN. Namely the pinball problem occurs. However, in the case of proposed method, the route is HA1, P-MR, MR, Sub-MR2, MNN. As compared to legacy method, the route is shorter by resolving the pinball problem.

4 Connection Mechanism with Information

As wireless techniques are developing, a mobile node has an ability to move everywhere. An increasing techniques support wireless and mobile movement.

While a MNN communicates with a CN, the MNN may migrate into coverage of a neighbor AR. In this situation, the MNN has to search for a possible channel to the neighbor AR and a MR and a HA configure a binding cache in itself, respectively. Moreover, as long as a MNN changes its attachment point, the communication session must be not disconnected between a MNN and a CN. In case of nested Network Mobility, a MNN can connect to some attachment points, i.e, a MR, a sub-MR, a Local Fixed Node (LFN) and a Local Mobile Node (LMN) in a neighbor AR which the MNN heads for. Therefore the MNN need to select the best attachment point for effective traffic.

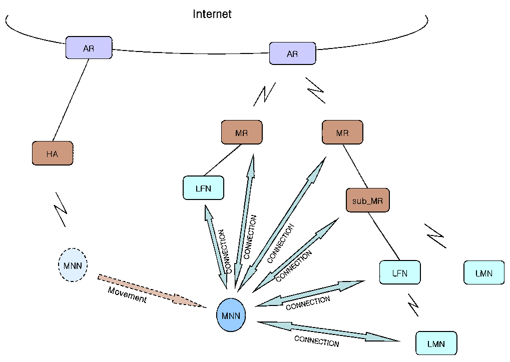


Fig. 3. A MNN attaches to sub-Network of neighbor AR in Nested NEMO

Figure 3 represents that a MNN changes into anther coverage of a neighbor AR. When the MNN which is located on the left AR migrates to the coverage of a right AR, the MNN should maintain current communication session between the MNN and the CN. Therefore, the MNN attempts to connect to some attachment nodes which supply mobile nodes with connectivity. Above figure shows three possible attachment nodes, that is to say, a MR, a LFN, a LMN. In other words, when a MNN change its point of attachment into the coverage of another AR, the MNN may receive RA signals from several attachment nodes. Thus the MNN has to choose which point the MNN attaches to, and check which point is more efficient in the aspect of processing faculty.

4.1 Router Advertisement and Router Solicitation

The router solicitation message[11] is used to ask a router advertisement message by a MNN. When the MNN needs to find out the prefix of neighbor attachment nodes address, the MNN broadcasts a RS message. If a new attachment node within a neighbor AR receives the RS message, this attachment node broadcasts the RA messages at once.

A MNN can receive several RA messages from a variety of attachment nodes in a neighbor AR under the nested NEMO. Thus the MNN has to choose the most efficient one of some attachment nodes. Therefore the MNN requests the information about attachment nodes which broadcast the RA message. However legacy mechanism does not support this information. Consequently we suggest a new RS format as subsequent figure.

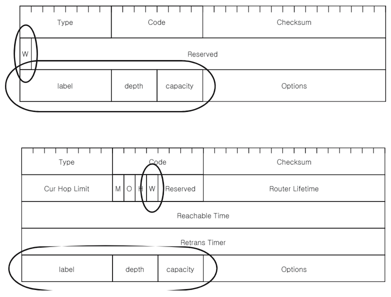


Fig. 4. Router Solicitation Format and Router Advertisement Format

The suggested RS format of figure 4 is equal extremely to existing router solicitation format. But a new bit W is inserted in the field of reserved bytes and information such as the label, the depth and the capacity are generated in the field of options. This bit W is used to request the information about the attachment node by a MNN. If the MNN receives only one RA message, the MNN sets this bit W to 0 and sends a RS message. If the MNN receives one or more RA messages from different attachment nodes, the MNN sets bit W to 1 and generates a keyword and then forwards the RS messages. This bit indicates query for the information of attachment node such the label with keyword, as the depth from parent-MR and the capacity about whether the attachment node can accept communication with the MNN or not.

Figure 4 shows a bit called W within Reserved bytes in second row. This bit W is used to indicate whether information about the attachment point is contained or not. The information of the attachment node includes the label, the depth and the capacity. First of all, the label designates the keyword to use for communication between a MNN and the attachment node. Second, the depth means the deepness from the parent-MR to the current attachment point in nested NEMO. Finally the capacity stands for the processing faculty of the attachment node. That is about whether a MR can accept the connection with a new MNN or not.

This information which includes the label, the depth and the capacity is inserted in the field of options. An attachment node sets the bit W to 1 only if the attachment node receives a request message from a MNN. Otherwise, the bit W is set to 0. If a MR gets a RS message from a MNN, the MR sets the bit W to 1 and adds the information about the label, the depth and the capacity of itself within options field. In addition, the bit W of the RS message is set to 1 and the RS message contains the label, the MR stores the label in its binding cache and then replies to the MNN with the RA message which has flag W.

4.2 Connection to New Attachment Point

A MNN moves into the coverage of a neighbor AR on the right side in figure 3. As soon as the MNN gets RA message, the MNN yields CoA and sets the bit W to 1 and inserts a keyword within options field. The MNN multicasts RS messages. A left MR, a LFN and a right MR, a sub-MR, a LFN, a LMN receive the RS message according to figure 3. After these attachment nodes which have a shot at being a new connection point for the MNN receive the RA message with the bit W that is set to 1 and the label, these nodes figure out the label with a keyword received from the MNN, the depth from parent-MR, the and the capacity for processing faculty respectively. And then those nodes transit the RA message to the MNN by modifying the value of bit W into 1. Since the MNN gets the RA messages from several attachment nodes, the MNN decides which attachment node is more efficient and the best node by checking out the label, the depth and the capacity of respective attachment nodes.

The four RA messages may be arrived at the MNN. If a right MR is parent-MR, the depth of the parent MR has 0. The depth of sub-MR, LFN and LMN are 1, 2, 3 in turn. And if the parent-MR can accommodate a new MNN for connection to outer Internet, the capacity of the parent-MR is 1. Unless the parent-MR serves any other node, the capacity is set to 0.

The MNN compares all information from the advertisement messages and then attempts to select a relevant node according to the depth. First of all, the MNN puts the low value of the depth to the higher priority. If the depth values of two nodes are 1 and 2, the MNN tries to investigate whether the capacity value of the node which has depth 1 is 1 or not. If the capacity is 1, the MNN connects to the attachment node with depth 1 and capacity 1, because the attachment node with depth 1 is located at the place which is closer to a parent-MR than the attachment node with depth 2.

If the capacity of node with depth1 is set to 0, the node can not accept a new MNN any more. Therefore, the MNN attempts to check the capacity value of another node with depth 2. If the capacity value is 1, a MNN connects to the attachment node with depth 1 and capacity 1. Thus the MNN searches through the neighbor area for a probable attachment node in this way.

5 Simulation and Evaluation

In this section, by using a network simulator[12][13], we generate the scenario with 100 MNNs which move together or alone and we simulate an existing simple handoff in nested NEMO and the proposed information-based connection procedure. The simulation environment is FreeBSD 4.7 and NS-2 2.1b6.

5.1 Throughput

The graph of figure 5 shows the result of packet throughput between the nested NEMO of the legacy mechanism and the information-based connection of the proposed mechanism during simulation. When a MNN moves to the coverage of neighbor attachment node, the MNN sends or receives the packets to or from a CN fast if the MNN selects the most effective attachment node. Figure 5 shows the proposed mechanism is better than the legacy mechanism.

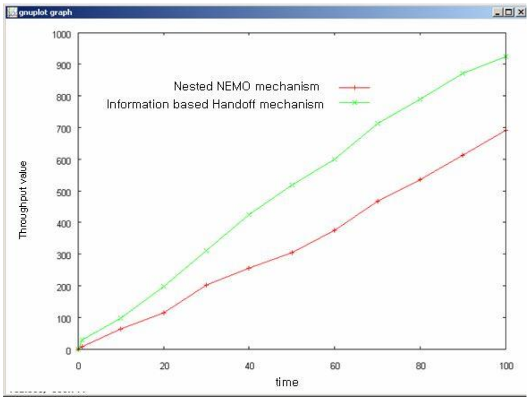


Fig. 5. Comparison result of packet throughput

In case of the legacy mechanism, the MNN chooses anyone of the attachment nodes. This legacy mechanism causes the heavy traffic and the unnecessary signals over the network. If the MNN changes its point of attachment and the upper node is full of many packets, the MNN dons not communicate with the CN well.

The result of Figure 5 shows that our proposed mechanism is of benefit to process many packets on real time during people walk around street. It is because the MNN seeks for the most effective attachment node when the MNN moves.

In addition, the MNN uses the information within the RS and RA message in case of the proposed mechanism. This information is activated with flag W in the messages and contains the label, the depth and the capacity. The label is used to search the effective attachment node fast, and the depth is used to check the attachment node close to parent-MR. The nearest node to the parent-MR has a chance to send packets very fast. The capacity is used to find that the MNN has the processing faculty enough to process the packets.

5.2 Traffic

We analyze the network traffic generated by applications. Traffic occurs whenever intermediate nodes process the packets which become an exponential growth. Figure 6 shows that the traffic of the proposed mechanism is decreased less than the traffic of legacy mechanism. The reason is that network overhead is reduced by decreasing the messages both from MNNs to HA and from MNNs to CNs. Our proposed mechanism enables the attachment nodes to process the packets faster. Therefore, the proposed mechanism optimizes the transmission path and reduces the delay time using the Information-based connection algorithm and

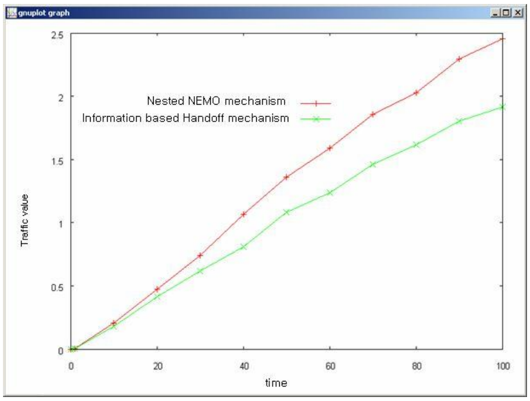


Fig. 6. Comparison result of traffic

the traffic distribution algorithm after a MNN migrates to switching area and an attachment node can not process the packets in time.

Moreover, the all of the attachment nodes perform the traffic distribution algorithm whenever the attachment nodes are filled with a lot of packets and begin to discard some packets, which are out of the processing faculty of the attachment node. Thus, all of MNNs search for the more effective attachment node and connect to that attachment node. In the end, the network overload and traffic are decreased and the nested NEMO enable the MNNs to be satisfied with communication. The transmission cost for packet is reduced from 1.2 to 0.9 as mentioned in section 5 and the transmission cost is applied to the traffic result of figure 6. For packet throughput, the proposed mechanism processes much more 145.4906 Byte/ms (17.39 percent). For throughput rate, the proposed mechanism improves 15.11 percent as decreasing at the rate of 0.1373. Our proposed mechanism is more effective in the two aspects of the throughput. For traffic, the proposed mechanism reduces 0.347684 Byte/ms (14.15 percent) on the average. For data transfer delay, 1.753958 Byte/ms (14.85 percent) is decreased.

6 Conclusion

In this paper, we suggest the information-based connection algorithm. The MNN checks out which node is the most effective attachment node of the probable nodes which reply with the RA message with flag W according to the priority of the depth, the capacity and the label. The MNN selects one attachment node with the lower depth and the capacity value which is set to 1. And then the MNN stores the label in binding cache. This value is used to search for new attachment node when the upper MR does not process the packet transmission because the MR gets a lot of packets out of its processing faculty. Therefore, with the ongoing communication session remained, the MNN can move freely using the information-based connection algorithm and the traffic distribution algorithm.

The simulation results of new algorithm which is proposed in this paper show that it improves the packet throughput and the packet rate, and it also reduces the application traffic avoiding the data transfer delay which is caused by the heavy attachment nodes.

References

1. Devarapali, V., Wakikawa, R., Petrescu, A., Thubert, P.: Network Mobility (NEMO) Basic Support Protocol, RFC3962 (2005)
2. Ng, C., Paik, E., Ernst, T.: Analysis of Multihoming in Network Mobility Support, draft-ietf-nemo-multihoming-isuues-02.txt (August 2005)
3. Ryu, H.-K., Kim, D.-H., Cho, Y.-Z.: Improved Handoff Scheme for Supporting Network Mobility in Nested Mobile Networks. Springer, Heidelberg (2005)
4. Watari, M., Wakikawa, R., Ernst, T., Murai, J.: Optimal path Establishment for Nested Mobile Networks. IEEE, Los Alamitos (2006)
5. Lamsel, P.: Network Mobility, research seminar on hot topics. In: Internet protocols (2005)
6. Ernst, T., Lach, H-Y.: Network Mobility Support Terminology, draft-ietf-nemo-terminology-03.txt (August 2005)
7. Ernst, T.: Network Mobility Support Goals and Requirements, draft-ietf-nemo-requirement-04 (August 2005)
8. Beacon Routing in Active Network, `http://www.ittc.ku.edu/ananth/845.html`
9. Miller, M.A.: PE, Inside Secrets SNMP Managing Internetworks. SamGakHyung Press (1998)
10. Wang, Y., Chen, W., Ho, J.S.M.: Performance Analysis of Adaptive Location Management for Mobile IP. Technical Report 97-CSE-13, Southern Methodist University (1997)
11. Hagen, S.: IPv6 Essentials, Oreilly (July 2002)
12. The CMU Monarch Project's Wireless and Mobility Extension to ns: The CMU Monarch Project (August 1999)
13. Widmer, J.: Network simulations for A Mobile Network Architecture for vehicles, International Computer Science Institute Technology Report TR-00-009 (May 2000)

Leakage Energy Reduction in Cache Memory by Software Self-invalidation

Kiyofumi Tanaka and Takenori Fujita[⋆]

School of Information Science, Japan Advanced Institute of Science and Technology,
1–1 Asahidai, Nomi-city, Ishikawa, 923–1292 Japan
{kiyofumi,t-fujita}@jaist.ac.jp

Abstract. Recently, energy dissipation by microprocessors is getting larger, which leads to a serious problem in terms of allowable temperature and performance improvement for future microprocessors. Cache memory is effective in bridging a growing speed gap between a processor and relatively slow external main memory, and has increased in its size. However, energy dissipation in the cache memory will approach or exceed 50% of the increasing total dissipation by processors. An important point to note is that, in the near future, static (leakage) energy will dominate the total energy consumption in deep sub-micron processes. In this paper, we propose cache memory architecture, especially for on-chip multiprocessors, that achieves efficient reduction of leakage energy in cache memories by exploiting gated-Vdd control and software self-invalidation. In the simulation, our technique reduced 46.5% of leakage energy at maximum, and 23.4% on average, in the execution of SPLASH-2 programs.

1 Introduction

In recent years, energy consumption of a microprocessor is getting larger due to increasing transistor counts according to Moore's Law and acceleration of operation clock frequency. The high energy consumption makes a lifetime of increasingly common battery-powered devices short. In addition, the increase of energy dissipation raises the temperature of LSIs and consequently violates operational conditions or becomes an obstacle to progress of microprocessor's running clock frequency. Therefore, reduction of energy consumption is indispensable to performance improvement of future microprocessors.

On the other hand, large-scale and sophisticated software is spreading and working set size in applications is getting larger. Therefore, high performance processing requires a large amount of cache memory in order to bridge a speed gap between a processor and external memory. Consequently, energy dissipation in cache memory exceeds 50% of the total consumption by a processor [1]. The energy reduction in cache memories is essential and some solution to the problem must be provided for future microprocessor architecture, especially in terms of leakage energy that would be more serious in future sub-micron processes.

[⋆] Presently, the author is with Semiconductor Company, Matsushita Electric Industrial Co., Ltd.

© Springer-Verlag Berlin Heidelberg 2007

In recent years, on-chip multiprocessors are becoming popular since they have the advantage of high performance. In this paper, we propose cache memory architecture for on-chip multiprocessors, that exploits gated-Vdd transistors and explicit gated-Vdd control by some kind of load and store instructions, and achieves substantial reduction of static energy consumption. In addition, we show the effectiveness by cycle-based simulations using SPLASH-2 benchmarks.

Section 2 describes several related works on leakage energy reduction in cache memories and on self-invalidation techniques. In Section 3, we propose the cache memory architecture that enables software self-invalidation and reduces leakage energy. Section 4 shows effects of the technique we propose with simulation results, and Section 5 concludes this paper.

2 Related Work

There are several architectural techniques proposed for leakage energy reduction in cache memories. Dynamically ResIzable instruction cache (DRI i-cache) reduces energy dissipation by dynamically downsizing effective caching areas [2]. Whether downsizing or upsizing is performed depends on the number of cache misses that occurred in some interval. When the miss count is fewer than a bound given in advance, the cache is downsized, and vice versa. The area that is not to be accessed is turned off by controlling gated-Vdd transistors and does not consume static energy after that [3]. This method focuses only on an instruction cache and the cache is divided into only two parts, active and sleeping areas.

There are other methods that are based on fine-grain gated-Vdd control. Cache decay is an energy-reduction scheme that controls gated-Vdd per cache block [4]. A block is in a dead-time state when it is in the interval between the last access to the block and replacement. Blocks in the dead-time state are turned off by gated-Vdd control and then any static energy is not wasted for the blocks. However, it is impossible for a hardware mechanism to decide exactly whether a block is in dead-time or not. In their hardware organization, the decision depends on a counter value for each block. The counter counts cycles or ticks during which the block is not accessed. When the counter gets saturated, the corresponding block is regarded as having entered dead-time. This mechanism requires extra hardware for the counters and cannot eliminate misjudgment completely due to various access patterns in applications that include both short and long access intervals.

Cache blocks that are turned off cannot preserve data values in the methods mentioned above. Therefore, reaccessing such blocks causes a cache miss and involves a miss penalty. On the other hand, there are state-preserving techniques, ABC-MT-CMOS (Auto-Backgate-Controlled Multi-Threshold CMOS) [5] and drowsy cache [6]. ABC-MT-CMOS is a technique where threshold voltages are dynamically manipulated and leakage energy is reduced. Memory cells can retain values even in a sleep mode. However, reaccessing the sleep cells requires waiting for the cells to wake up. MT-CMOS requires complex circuitry and therefore tends to increase the hardware size. Drowsy cache prepares two different supply-voltage modes, high-power and low-power modes, instead of turning off. Cache blocks in the low-power mode cannot be read or written. Although the amount of energy reduction is smaller than the gated-Vdd control, blocks even in the low-power mode can preserve data values. Each block periodically falls into the low-power mode, and is woken up to the high-power mode when the block is reaccessed. The penalty for waking up a low-powered block is much smaller than that in the gated-Vdd controlled caches. This mechanism expects the characteristics in programs that there are a limited number of memory blocks that are frequently accessed in some short period, and effectively reduces leakage energy.

The defects of the gated-Vdd control are an additional cache miss penalty caused by data disappearance and increase of dynamic energy consumption for accessing the next level memory hierarchy on the misses. On the other hand, those of state preserving ones such as drowsy caches are relatively large additional hardware and lower efficiency of leakage energy reduction since any memory cells always keep some voltage. Our method proposed in the next section aims at reducing extra cache misses while achieving as much energy reduction as gated-Vdd control, by using software self-invalidation. Self-invalidation was originally a technique for mitigating overheads of cache coherence management in distributed shared memory [7,8]. We apply the concept of self-invalidation to energy reduction in cache memory. The self-invalidation methods proposed in [7,8] were controlled fully by hardware. Therefore, they are not appropriate energy-reduction mechanisms since they require special hardware, version number directory or signature tables, that consumes dynamic energy by itself. Then we propose a software self-invalidation technique in this paper.

3 Software Self-invalidation

This section describes a self-invalidation technique by software. The technique is achieved by last-touch memory reference instructions and cache hardware mechanisms invoked by the instructions.

3.1 Last-Touch Memory Reference Instructions

For efficiency of software self-invalidation, we introduce the instructions, last-touch load/store, execution of which invalidates addressed cache blocks after accessing them. There are two types of condition for invalidation as follows.

- A block is invalidated at the same time as it is accessed.
- A word is marked when it is accessed. The block is invalidated when all words in it get marked.

We call the former type of instructions last-touch-block load/store (ltb ld/st), and the latter last-touch-word load/store (ltw ld/st). When write-back policy is employed and a block that is designated to be invalidated is of a modified state, the invalidation is performed after write-back operation or insertion into a write buffer if it exists.

For example, load/store instructions that access each address only once before it is invalidated by other processor caches can be replaced by the last-touch load/store instructions. Similarly, ones that access each address only once in a generation (between block filling and replacement) can be replaced by the last-touch load/store instructions.

Fig. 1 shows examples of application of the last-touch memory reference instructions. In the left figure, the variable "Globalid" is updated in the critical section and then the corresponding cache block would be invalidated by a subsequent processor that enters the critical section. Therefore, the last-touch-block store can be applied to the update of the variable provided that other variables in the same block are not referenced after the unlock. On the other hand, in the right figure, sequential references to the array "src" and "dest" in the loop are basically last access to each word provided that the loop count is large enough to replace read blocks. Therefore, by applying the last-touch-word load instructions to the array references, self-invalidation can be performed without extra cache misses. In this example, when the array consists of 4-byte word elements and the block size is 16 bytes, self-invalidation is performed once every two iterations (references to four elements).

```
        :
LOCK ( Global lock );
    MyNum = Globalid;
    Globalid += 1;
UNLOCK ( Globallock );
        :
```

(a)

```
        :
for ( i = 0; i < n1; i++ ) {
    dest[2*i] = src[2*i];
    dest[2*i+1] = src[2*i+1];
}
        :
```

(b)

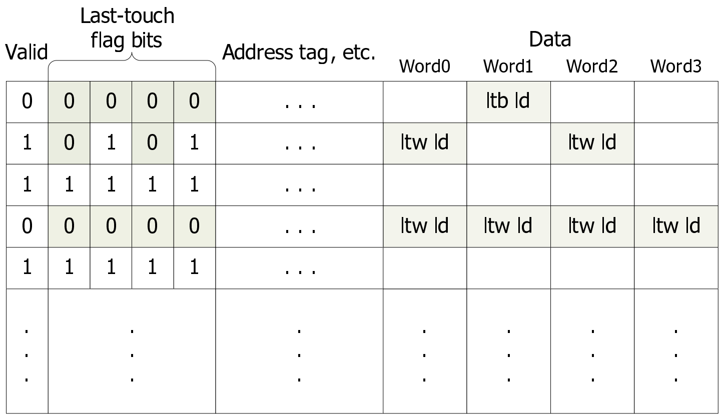
Fig. 1. Application of last-touch memory reference instructions

3.2 Hardware Mechanisms

It is necessary to give a small modification to conventional cache memory structure to reduce energy dissipation by using the last-touch instructions.

Last-touch flag bits are a part of cache tag information and indicate which word in the cache block has been accessed by the last-touch load or store instruction. Fig. 2 shows the cache memory structure including the last-touch flag bits, when the block size is 16 bytes and a word size is 4 bytes. A single last-touch flag bit corresponds to a word in the block. When a last-touch-word load/store instruction is executed, the corresponding flag bit is cleared. On the other hand, when a last-touch-block load/store instruction is executed, all flag bits are cleared (as depicted in the top set in the figure). Then, a block is invalidated when all the flag bits are cleared.

A valid bit of the whole cache block can be generated by a logical disjunction (OR) of the last-touch flag bits. In other words, last-touch flag bits are regarded as a valid bit of each word. The last-touch flag bits are additional hardware



Valid	Last-touch flag bits				Address tag, etc.	Data			
						Word0	Word1	Word2	Word3
0	0	0	0	0	. . .		ltb ld		
1	0	1	0	1	. . .	ltw ld		ltw ld	
1	1	1	1	1	. . .				
0	0	0	0	0	. . .	ltw ld	ltw ld	ltw ld	ltw ld
1	1	1	1	1	. . .				
⋮	⋮				⋮	⋮	⋮	⋮	⋮

Fig. 2. Cache memory structure

to conventional cache tag information. We choose flag bits per word, not per byte, considering that the additional hardware amounts should be small and that applications often process data on a word basis.

We assume that the gated-Vdd (or gated-Vss) is implemented by following the technique proposed by Yang, et al. [2]. This is a wide NMOS dual-Vt gated-Vdd with a charge pump and has about 5% of area overheads. The gated-Vdd transistor is inserted between ground and SRAM cells (virtual ground). When the gated-Vdd transistor is turned off, the leakage energy is virtually eliminated. Fig. 3 is a conceptual diagram of a cache block. The address tag and data parts of the block are connected with one or more gated-Vdd transistors. The gated-Vdd transistors are controlled by the valid bit. When the valid bit is one, the gated-Vdd is turned on, otherwise, turned off and leakage energy in the address tag and data areas is eliminated. (When the valid bit is prepared separately from last-touch flag bits, the last-touch flag bits can be turned off as well).

After a block is turned off, it takes a certain delay to wake the block up again. This wakeup latency depends on the LSI process used and the number of

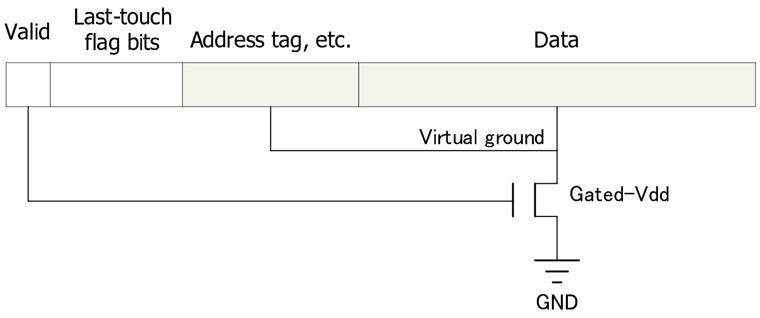


Fig. 3. Gated-Vdd control

bits that a single gated-Vdd transistor takes charge of. Short wakeup latency is desirable from a performance point of view [9]. Kaxiras, et al. were optimistic about the wakeup latency, since they estimated that the latency is hidden by an L1 cache miss penalty [4]. Similarly, several researches adopted relatively short time, a few cycles, as the wakeup latency [10][11]. We follow the same (optimistic) assumption in this paper.

4 Evaluation

4.1 Environment

We developed a scalar processor simulator that executes the SPARC version 9 instruction set [12]. The simulator executes an instruction per cycle (several instructions such as multiplication and division take three or more cycles) and outputs the total execution cycles and other informations; the number of cache misses, write buffer stall cycles, consumed energy, etc. The simulator has two target processors for multiprocessor configuration. Each processor has L1 instruction/data split caches that follow write-back policy and a write buffer. The L1 caches are connected by a shared bus and managed based on a write invalidate protocol. The L2 cache is shared by all processors. The configurations of the caches and write buffer are shown in Table 1.

Table 1. Cache configuration

L1 I-&D-cache	32KB, 16-byte block 4-way set associative (LRU) 1-cycle latency on hit
L2 unified cache	Infinite size, 16-byte block 10-cycle latency on hit
Write buffer	Infinite size 1-cycle latency for insertion

In the simulation, the simulator calculated leakage energy in the L1 caches by using the following formula.

$$\begin{aligned} \textit{Leak energy} &= \textit{Active cells} \times \textit{Active leakage per cell} \times \textit{Active cycles} \\ &+ \textit{Standby cells} \times \textit{Standby leakage per cell} \times \textit{Standby cycles} \end{aligned}$$

Active leakage is for a turned-on cell and standby leakage is for a turned-off cell. The parameter values that were shown in [3] were applied to the above formula (1740 nJ/s for an active cell and 53 nJ/s for a standby cell, under 0.18μm).

For evaluation, we used five kernel programs in the SPLASH-2 suite [13]. The input data sizes or input file in the five programs are shown in Table 2.

Table 2. Input data sizes or input file for five programs

Program	Input data size / input file
FFT	65,536 complex
LU contig	256x256 matrix
LU non-contig	256x256 matrix
RADIX	262,144 keys
CHOLESKY	wr10.O

4.2 Application of Last-Touch Instructions

In this evaluation, we applied the last-touch load/store instructions by referring execution traces. This enables optimal application of the last-touch instructions and is helpful for finding the maximum effects of the proposed method, although it might be impractical in actual software execution environment.

The programs were simulated in advance in order to generate traces of memory accesses. After that, we updated the program (assembly) codes by replacing load/store instructions with last-touch ones, based on the traces. There are two types of traces, "address-based trace" and "PC-based trace". The traces were generated as follows.

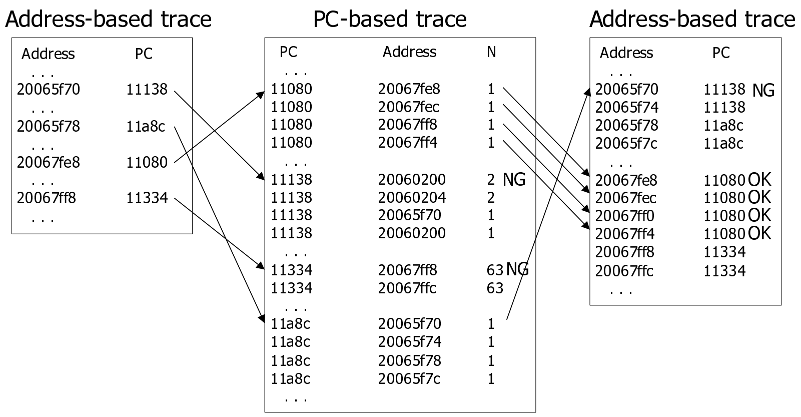
1. Whenever load/store instructions are executed, the instruction address (program counter) is recorded in the entry for the referenced address in the address-based trace. When the corresponding entry is not found in the trace, a new entry is generated and initialized to the instruction address.
2. Whenever load/store instructions are executed, the number of times of accessing the address by the instruction is incremented in the PC-based trace. When the corresponding entry is not found in the trace, a new entry is generated and initialized to 1.
3. As for receipt of an invalidate request, the addresses concerned that are referenced after the invalidation are regarded as different from those before the invalidation.

The procedure for applying the last-touch instructions is as follows.

1. Instructions that correspond to PCs in the address-based trace are "weak candidates" for the last-touch instructions.
2. The PC-based trace is scanned with a weak candidate as a key. If it is found that all addresses referenced by the candidate were accessed once, the candidate becomes a "strong candidate" for the last-touch instructions.
3. The address-based trace is re-scanned with all the addresses that the strong candidate referenced. If the PC that the address in the trace indicates differs from the PC of the strong candidate, the candidate is removed from the candidate group.
4. The remaining strong candidate instructions are replaced by a last-touch-block instruction if they referenced no more than one address per memory

block, otherwise, by a last-touch-word instruction. (We did not apply the last-touch instruction when candidates were an instruction for smaller data than a word).

Fig. 4 shows an example of the procedure mentioned above. The "Address" column in the address-based trace indicates addresses that were referenced by load or store instructions in the program execution. The "PC" column indicates the program counter value of a load or store instruction that accessed the corresponding address last. On the other hand, "N" column in the PC-based trace shows the number of times of accesses to the corresponding address by the PC instruction.



Address-based trace

Address	PC
...	
20065f70	11138
...	
20065f78	11a8c
...	
20067fe8	11080
...	
20067ff8	11334
...	

PC-based trace

PC	Address	N	
...			
11080	20067fe8	1	
11080	20067fec	1	
11080	20067ff8	1	
11080	20067ff4	1	
...			
11138	20060200	2	NG
11138	20060204	2	
11138	20065f70	1	
11138	20060200	1	
...			
11334	20067ff8	63	NG
11334	20067ffc	63	
...			
11a8c	20065f70	1	
11a8c	20065f74	1	
11a8c	20065f78	1	
11a8c	20065f7c	1	
...			

Address-based trace

Address	PC	
...		
20065f70	11138	NG
20065f74	11138	
20065f78	11a8c	
20065f7c	11a8c	
...		
20067fe8	11080	OK
20067fec	11080	OK
20067ff0	11080	OK
20067ff4	11080	OK
20067ff8	11334	
20067ffc	11334	
...		

Fig. 4. Decision to apply last-touch instructions

In the figure, first, the instruction whose PC value is "11138" is chosen as a weak candidate for a last-touch instruction. The PC-based trace shows that the instruction referenced the address "20060200" twice. Therefore, the instruction cannot be a strong candidate. The next weak candidate whose PC value is "11a8c" is searched in the PC-based trace. This instruction becomes a strong candidate since all of the referenced addresses have 1 for N. Then, the address-based trace is re-scanned for the address "20065f70" as a key. In the address-based trace, the address has "11138" as the last-touch PC, which does not equal to "11a8c", and therefore, this instruction is removed from the strong candidate group. Next, the instruction whose PC is "11080" is a weak candidate and searched in the PC-based trace. It is found that this instruction accessed each address once. Therefore, this instruction becomes a strong candidate. Then, the address-based trace tells that, for all the accessed addresses, the last-touch PC is "11080". Consequently, this instruction can be replaced by a last-touch instruction. The last-touch word load/store instruction is used in this case, since the distance between the accessed addresses is shorter than the block size. For example, "20067fe8" and "20067fec" reside in the same memory block. Finally, the PC "11334" has multiple accesses to an address in the PC-based trace, and therefore, cannot be a candidate.

For the last-touch instructions, we exploited load/store instructions from/to an alternate space that are implementation-dependent instructions in the SPARC architecture. These instructions can specify an address space identifier (ASI). We used a discrete ASI value for each of last-touch-block and last-touch-word instructions.

4.3 Results

The results of leakage energy consumption in the L1 cache memories are shown in Fig. 5. In the figure, the "without GV" indicates the execution without gated-Vdd control or self-invalidation. Therefore, this execution does not lead to any energy reduction. The "with GV" means the execution with gated-Vdd control for blocks that are naturally invalidated by invalidation messages between processor caches, not by self-invalidation. The "with GV&SI" is the execution with gated-Vdd control for blocks that are invalidated by invalidation messages and by self-invalidation operation. All results are normalized to the results of the "without GV" execution.

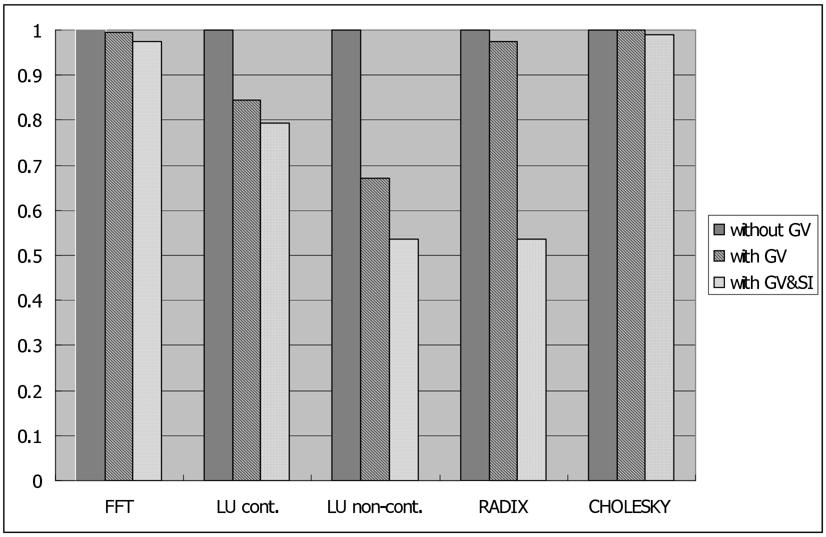


Fig. 5. Leakage energy

The "with GV" execution reduced 15.5% of leakage energy in LU cont, 33.0% in LU non-cont, and 2.5% in RADIX. For FFT and CHOLESKY, the execution could not reduce leakage energy. (0.6% and 0.08%, respectively.) Table 3 shows the number of invalidation by invalidate messages. The figure 5 and table 3 mean that the amount of leakage reduction by "with GV" was roughly proportional to the number of invalidation by invalidate messages.

Table 3. The number of invalidation

Program	# of invalidation by invalidate messages
FFT	39
LU contig	130,720
LU non-contig	232,019
RADIX	812
CHOLESKY	106

Table 4. The number of self-invalidation

Program	# of self-invalidation		# of last-touch-word instructions
	Last-touch-block	Last-touch-word	
FFT	36	66,044	264,190
LU contig	18	82,150	396,571
LU non-contig	18	157,156	908,555
RADIX	25	272,420	1,179,675
CHOLESKY	9	14,839	71,630

Table 5. Execution cycles

Program	without GV exec.	with GV&SI exec.
FFT	118,641,649	118,641,584
LU contig	100,084,316	100,084,040
LU non-contig	98,001,681	97,693,117
RADIX	122,541,690	122,418,679
CHOLESKY	39,266,223	39,258,305

Table 6. The number of cache misses

Program	without GV exec.	with GV&SI exec.	Reduction rate (%)
FFT	118,641,649	118,641,584	0.0012
LU contig	100,084,316	100,084,040	0.021
LU non-contig	98,001,681	97,693,117	3.0
RADIX	122,541,690	122,418,679	4.7
CHOLESKY	39,266,223	39,258,305	0.32

For all of the five programs, the proposed method ("with GV&SI") reduced more leakage energy than the simple "with GV" execution; 2.5% of leakage energy in FFT, 20.6% in LU cont, 46.3% in LU non-cont, 46.5% in RADIX, and 1.0% in CHOLESKY. Table 4 shows the number of self-invalidation by last-touch-block and that by last-touch-word instructions, and the number of last-touch-word instructions executed. Roughly, one self-invalidation operation is performed every four executions of the last-touch-word instructions. The figure 5 and table 4 show that a large amount of leakage energy was reduced by "with GV&SI" for LU non-cont and RADIX, which included many self-invalidations.

Table 5 shows the number of execution cycles for "without GV" execution and "with GV&SI" execution. For all the programs, the execution "with GV&SI" decreased the execution cycles, although the difference was small. This is because the number of cache misses was decreased. Table 6 shows the number of cache misses that occurred in each program execution. The caches basically employed LRU replacement policy where a block that was least recently used was replaced. On replacement, an invalid block entry, if it existed, was selected for an entry that was filled with a missing block. Therefore, the self-invalidation facilitates optimal replacement decision by invalidating blocks that are already in dead-time. On the other hand, without self-invalidation, the simple LRU might replace blocks that are still in live-time and lead to cache misses later. This is why the execution time "with GV&SI" was shorter than that "without GV".

5 Conclusion

In this paper, we proposed a method for reducing leakage energy dissipation by gated-Vdd control and software self-invalidation. This method can be implemented by introducing load/store instructions that explicitly indicate that the accesses are last-touch to the addresses, and by a small amount of additional hardware. By using trace-based translation into codes that included the last-touch instructions, our technique achieved substantial leakage energy reduction by 23.4% on average, and by 46.5% at best, for the SPLASH-2 kernel programs, while 10.4% on average and 33.0% at best by the execution with only gated-Vdd control for naturally invalidated blocks. In addition, it was found that the execution time was shorter than the execution without self-invalidation, since the self-invalidation increased the number of invalid blocks and decreased the number of cache misses, as a result.

The evaluation in this paper used codes that were generated by translation based on execution traces, which brought optimal application of last-touch instructions, but would not be practical in actual software execution. In the future, we will explore other methods of automatic code generation. For example, the literature [14] showed that cache misses could be decreased by compiler optimization that made load and store instructions have information about temporal and spatial locality between instructions. We have prospects of applying similar compiling techniques for leakage energy reduction.

References

1. Malik, A., Moyer, W., Cermak, D.: A Low Power Unified Cache Architecture Providing Power and Performance Flexibility. In: Proc. of ISLPED, pp. 241–243 (2000)
2. Yang, S.H., Powell, M.D., Falsafi, B., Roy, K., Vijaykumar, T.N.: An Integrated Circuit/Architecture Approach to Reducing Leakage in Deep-Submicron High-Performance I-Caches. In: Proc. of HPCA, pp. 147–158 (2001)

3. Powell, M., Yang, S.H., Falsafi, B., Roy, K., Vijaykumar, T.N.: Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories. In: Proc. of ISLPED, pp. 90–95 (2000)
4. Kaxiras, S., Hu, Z., Martonosi, M.: Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. In: Proc. of 28th ISCA, pp.240–251 (2001)
5. Nii, K., Makino, H., Tujihashi, Y., Morishima, C., Hayakawa, Y., Nunogami, H., Arakawa, T., Hamano, H.: A Low Power SRAM using Auto-Backgate-Controlled MT-CMOS. In: Proc. of ISLPED, pp. 293–298 (1998)
6. Flautner, K., Kim, N.S., Martin, S., Blaauw, D., Mudge, T.N.: Drowsy Caches: Simple Techniques for Reducing Leakage Power. In: Proc. of 29th ISCA, pp.148–157 (2002)
7. Lebeck, A.R., Wood, D.A.: Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors. In: Proc. of ISCA, pp.48–59 (1995)
8. Lai, A.C., Falsafi, B.: Selective, Accurate and Timely Self-Invalidation Using Last-Touch Prediction. In: Proc. of ISCA, pp.139–148 (2000)
9. Heo, S., Barr, K., Hampton, M., Asanovic, K.: Dynamic Fine-Grain Leakage Reduction Using Leakage-Biased Bitlines. In: Proc. of 29th ISCA, pp.137–147 (2002)
10. Li, Y., Parikh, D., Zhang, Y., Sankaranarayanan, K., Stan, M., Skadron, K.: State-Preserving vs. Non-State-Preserving Leakage Control in Caches. In: Proc. of DATE'04, vol. 1, pp.22–27 (2004)
11. Meng, Y., Sherwood, T., Kastner, R.: On the Limits of Leakage Power Reduction in Caches. In: Proc. of 11th HPCA, pp.154–165 (2005)
12. SPARC International, Inc.: The SPARC Architecture Manual Version 9. Prentice-Hall, Inc. (1994)
13. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 Programs: Characterization and Methodological Considerations. In: Proc. of ISCA, pp.24–36 (1995)
14. Wang, Z., McKinley, K.S., Rosenberg, A.L.: Improving Replacement Decisions in Set-Associative Caches. Technical Report, University of Massachusetts, UM-CS-2001-002 (2001)

Exploiting Task Temperature Profiling in Temperature-Aware Task Scheduling for Computational Clusters

Daniel C. Vanderster, Amirali Baniasadi, and Nikitas J. Dimopoulos

Department of Electrical and Computer Engineering,
University of Victoria,
P.O. Box 3055 STN CSC,
Victoria, B.C. V8W 3P6
Canada
{dvanders,amirali,nikitas}@ece.uvic.ca

Abstract. Many years of CMOS technology scaling have resulted in increased power densities and higher core temperatures. Power and temperature concerns are now considered to be a primary challenge for continued scaling and long-term processor reliability. While solutions for low-power and low-temperature circuits and microarchitectures have been studied for many years, temperature-awareness at the computational cluster level is a relatively new problem. To address this problem, we introduce a temperature-aware task scheduler based on task temperature profiling. We study the task characteristics and temperature profiles for a subset of SPEC'2K benchmarks. We exploit these profiles and suggest several scheduling algorithms aimed at achieving lower cluster temperature. Our findings show a clear trade-off between the overall queue servicing time and the cluster peak temperature. Whether the temperature reductions achieved are worth the extra delay is left to the designer/user to decide based on the case by case performance restrictions and temperature limitations.

1 Introduction

In recent years, the issues of power dissipation and energy consumption have come to the forefront of the minds of system designers [1] [2]. One specific area where this issue is relevant is in the realm of computational clusters [3]. These large systems often feature hundreds of servers and processors. Exploiting highly integrated high power density servers and processors in such systems has resulted in new thermal challenges. In fact, under new semiconductor technologies power density of the microprocessor core has exceeded $200\,{}^{\mathrm{W}}/{}_{\mathrm{cm}^2}$. Such high power densities can result in high temperatures which in turn can cause transient faults or permanent failures. Reducing the heat and lowering the cluster temperature is therefore a vital and important challenge. To address this challenge, two different approaches may be considered: First, we can develop more effective and often expensive ventilation and cooling systems to remove more heat at a higher rate.

© Springer-Verlag Berlin Heidelberg 2007

Second, we can develop new design techniques at several levels to reduce the production of the excess heat.

As effective cooling mechanisms become more expensive, it is important that designers develop temperature reduction techniques at different levels including scheduling. This work aims at exploring such solutions. We introduce the *Profile-based Temperature-aware Scheduler*, or *PTS*, to address this problem at the scheduler level. PTS relies on identifying and using processor-task combinations resulting in better temperature conditions in the computational cluster.

In particular, we make the following contributions:

– First, we study task temperature profiles. A task temperature profile indicates the amount by which a given task will raise the temperature of a host processor. We use our findings to differentiate between hot and cold tasks.
– Second, we use task temperature profiles of a subset of the SPEC'2K benchmarks and show that early knowledge of such profiles can be used to produce a temperature-aware schedule reducing the cluster peak temperature.

The rest of the paper is organized as follows. In section 2 we present task temperature profiling. In section 3 we present and evaluate our scheduling policies. In section 4 we discuss some related work. In section 5 we offer concluding remarks.

2 Task Temperature Profiling

In this study, a task temperature profile is a time series measure of the amount by which a task will raise the temperature of a host processor. Our observations have shown that not all tasks generate the same amount of heat on a given processor. This could be explained by the task's behaviour including the time it spends performing integer or floating point calculations, reading and writing to memory, or waiting for asynchronous events. For example, a processor bound task can generate more heat compared to an I/O bound task.

Based on how a task impacts processor temperature, we can observe a spectrum of tasks ranging from *hot*, or high-temperature, tasks to *cold*, or low-temperature, tasks.

Our goal is to reduce the cluster peak temperature by assigning hot tasks to cold processors. To achieve this, a temperature-efficient schedule should exploit the variance in the cooling ability of the processors in a cluster. We have observed this variance is related to the physical proximity of a processor relative to a cooling vent, and is indicated by the fan-input temperature of the chassis housing the processor. We therefore classify processors based on their fan-input temperatures into groups, or *processor classes*. For example, the 168 processors in a typical rack may be classified into six processor classes, one for each of the rack-mounted chassis[1].

[1] In a typical computational cluster, the chassis closest to the floor vents will have the lowest fan-input temperature, and thus the best cooling ability.

In order to measure the task temperature profiles and classify the processors, we developed a tool which monitors several temperature metrics for a task executing on a blade in an IBM BladeCenter. Each blade contains two Intel Xeon processors and with 2 gigabytes of memory and runs Red Hat Enterprise Linux AS 3. The measurements are performed using the on-chassis and on-board thermal sensors present on the IBM BladeCenter chasses and blades. A monitoring daemon periodically polls the BladeCenter management interface using the Simple Network Management Protocol (SNMP). The measured metrics include the processor core temperatures of the both processors in a blade, the chassis fan-input temperature, the temperature of the management console located at the rear of the chassis, and the chassis blower speeds measured as of percentage relative to maximum blower RPM. To avoid interference from background applications, both processors in a blade are exclusively reserved for the duration of the tests.

We generated task temperature profiles (shown in Figure 1) for a subset of the SPEC'2K benchmarks. Note that a single script executing each benchmark in succession is used to ensure the execution environment is identical for each benchmark. Prior to the execution of each benchmark, the processor is idled for 300 s to allow the system to return to its idle temperature. While these plots represent the profiles attained with a single execution of the benchmarks, further experiments on other processors demonstrated similar profiles.

In Figure 1 we see that longer benchmarks (e.g., *mcf* and *swim*) show higher peak temperatures. Accordingly, tasks with shorter runtimes (e.g., *crafty* and *gzip*) cause less heat and therefore lower peak temperatures. *Wupwise* has plateaued at between 20 °C and 25 °C, which is a lower peak compared to *mcf* and *swim*. It is notable that some of the tasks (e.g., *crafty*, *gzip*, and *vortex*) appear not to have reached a steady-state temperature – it is reasonable to expect that had those benchmarks continued executing for longer, their temperatures would have continued to rise. While noting this situation, we have chosen not to increase their execution times in this study because we are interested in varied profiles, including those that have and do not have a steady state. However, it is clear that further tests of processor, I/O, and memory bound tasks are needed to explore the full spectrum of temperature profiles.

If we assume that our selection of SPEC'2K benchmarks represents a set of tasks queued at a computational cluster, then their temperature profiles demonstrate that a computational cluster will have a spectrum of tasks ranging from hot to cold. The hot tasks are those having the highest peak temperature, while the cold tasks are those having the lowest peak temperature. We exploit this spectrum of task profiles to come up with a schedule which minimizes cluster peak temperature.

3 Temperature-Aware Scheduling

Our goal is to decrease the cluster peak temperature by distributing tasks amongst the processors so that hot tasks are assigned to processors with the

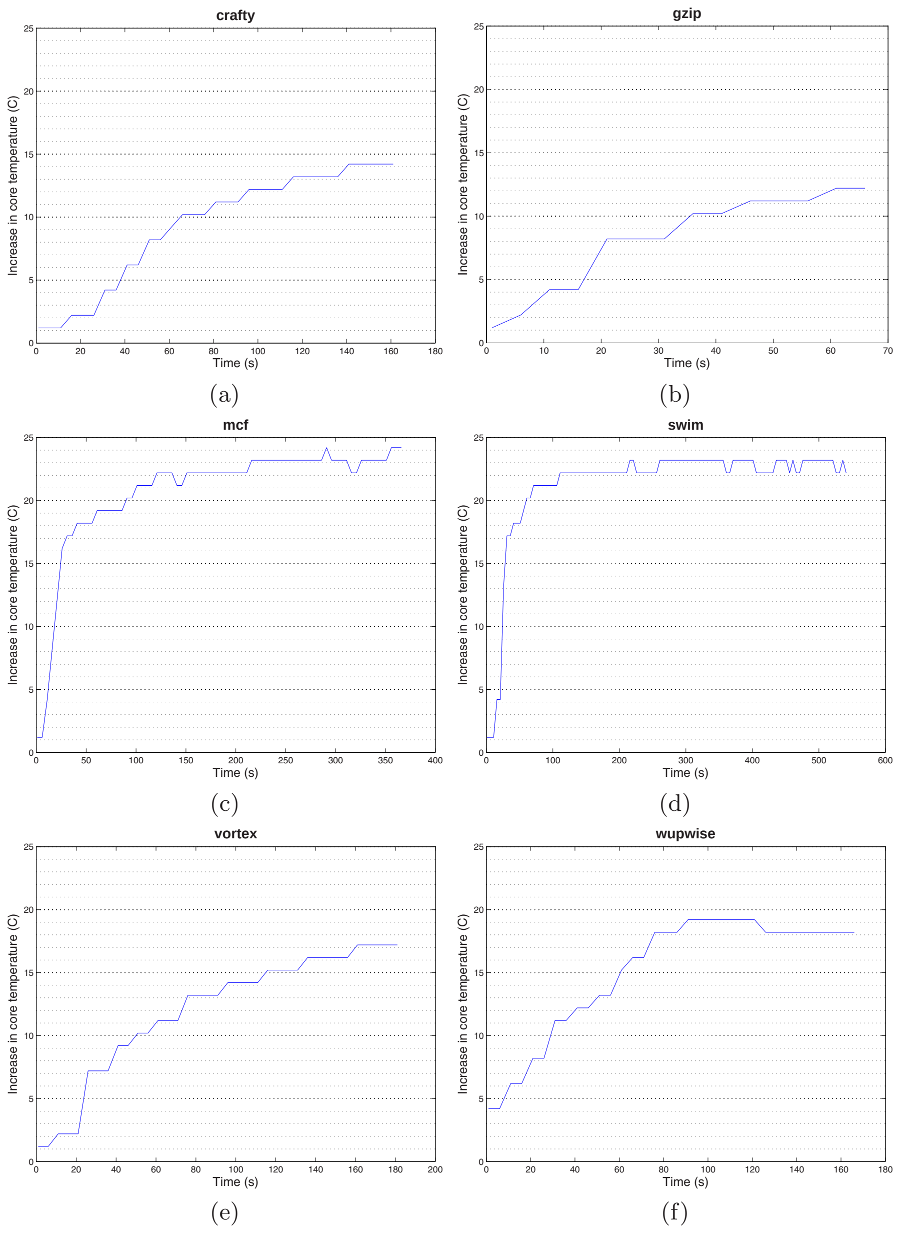


Fig. 1. Task temperature profiles for the SPEC'2K benchmarks (a) *crafty*, (b) *gzip*, (c) *mcf*, (d) *swim*, (e) *vortex*, and (f) *wupwise*

best cooling ability. Our scheduling techniques rely on the assumption that the task temperature profiles are relatively invariant between subsequent executions of similar tasks. We assume that there exists a mechanism that can recognize the similarity of tasks by comparing task metadata including the submitting user, the task name, and invocation arguments. When a submitted task is recognized by this mechanism, a previously measured profile is used to schedule the task. When a new or unrecognized task is submitted to the scheduler, its metadata and temperature profile are recorded for subsequent executions of the task. Additionally, we simplify the environment by assuming that task preemption (and subsequent migration) at the batch scheduler level is disallowed[2]. All processors are assumed to be equal in performance.

3.1 A Profile-Based Temperature-Aware Scheduler

A simple computational cluster scheduler, shown in Figure 2, features a single queue which allocates jobs to all processors. Since there is no notion of temperature awareness, hot tasks may be assigned to the hottest processor, resulting in a high peak cluster temperature. The PTS scheduler, shown schematically in Figure 3, divides the processors into processor classes based on their cooling ability as described by their fan-input temperatures. The processors in each processor class are managed by a first-come-first-served task queue specific to the processor class. A task director assigns each task to a processor class queue by inspecting the temperature profiles and sorting the tasks in the global queue by their peak temperature. The task director allocates the tasks in the sorted global queue by assigning an equal number of tasks to each processor class queue starting with the coolest processor class. In general, this results in the hot tasks being assigned to the coolest processors and the cold tasks being assigned to the hottest processors.

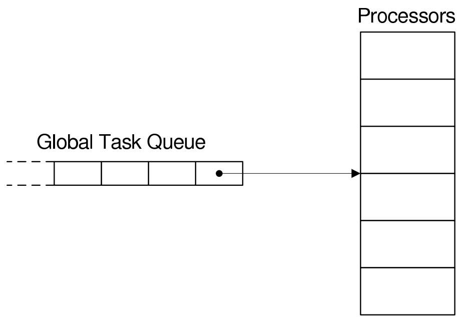


Fig. 2. Schematic of a simple scheduler having a single queue which assigns tasks to any processor in the cluster

[2] By assuming an absence of task preemption, we not only simplify the problem for simulation, but also allow the provided solution to be applicable to task sets in which the majority of tasks do not support checkpointability (which is common in our experience). A derivative adaptive scheduling strategy that re-evaluates and reallocates jobs is clearly possible in environments supporting preemption.

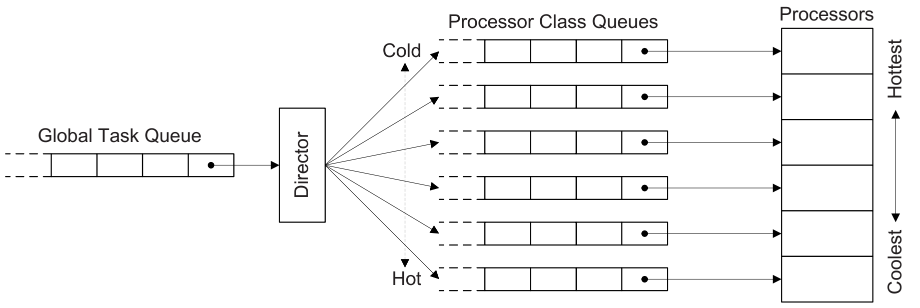


Fig. 3. Schematic of the PTS scheduler, featuring a director which sorts tasks in the global queue into sub-queues ranging from cold to hot. Each of the sub-queues assigns tasks to a processor in its associated processor class.

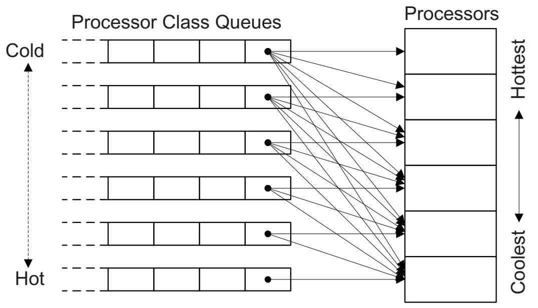


Fig. 4. Schematic of queue-processor relationship in the PTS-FM scheduler, which allows tasks to favourably migrate into colder idle queues

Because the queues in Figure 3 will not necessarily empty at the same rate, the division of tasks equally into separate queues may result in an increase in the global queue-servicing-time. For example, there may be a number of tasks waiting to be executed on the hottest processors, while some colder processors are idle because their corresponding queues are empty. Recognizing that cold tasks may be assigned to any of the processor classes, we can allow these tasks to *favourably migrate* into colder queues. Namely, in the case that a queue Q is empty, we allow a colder task to migrate into Q (where "colder" implies that the processor class associated with Q has a lower fan-input temperature than that of the task's original queue). This technique, designated PTS with Favourable Migration (PTS-FM), is shown in Figure 4.

To further increase processor utilization, the scheduler can be configured to allow the thermally unfavourable migration of tasks to a tunable number (n) of hotter idle queues. Shown in Figure 5, the PTS with Unfavourable Migration (PTS-UM$_n$) strategy realizes a trade-off between an increase in the peak cluster temperature and a decrease in the queue servicing time. The dotted lines in Figure 5 represent the allocation of tasks to thermally unfavourable processor

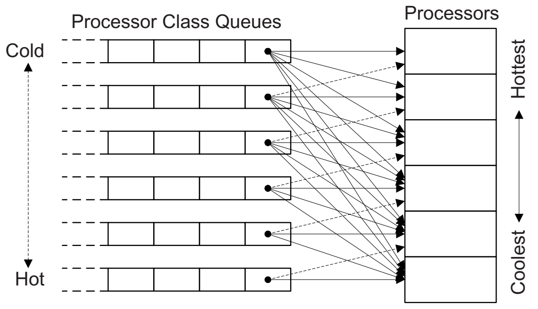


Fig. 5. Schematic of queue-processor relationship in the PTS-UM$_n$ scheduler, which allows tasks to unfavourably migrate into a tunable number (n) of hotter idle queues, in addition to the favourable migration into colder idle queues. PTS-UM$_1$ is shown.

classes. The shown PTS-UM$_1$ strategy allows tasks to unfavourably migrate by only one processor class. By increasing the allowed degree of unfavourable migrations, a further decrease in execution time will be realized along with a corresponding increase in the peak temperature. In the extreme case, where all of the queues can allocate tasks to all of the processor classes, the scheduler becomes a simple first-come-first-served (FCFS) scheduler.

For each scheduling strategy we measure the peak processor temperature as well as the overall queue servicing time (the time required to execute all tasks in a given queue). In all cases we compare our results with a baseline policy where the FCFS algorithm services the global queue in the order that tasks have arrived (referred to as the FCFS$_6$ policy). Additionally, we compare the PTS strategies to a simple FCFS-based power saving strategy, where the hottest processors are simply turned off. In the results below, these are referred to as FCFS$_5$ through FCFS$_1$, where the subscript represents the number of available processors.

It should be noted that in the following simulations we assume that the director has proceeded through a training period and all task temperature profiles are known. However, in practise, when the director receives a new, unrecognized, task, a conservative strategy could be followed while its temperature profile is recorded. Specifically, a conservative policy would specify that new tasks should be allocated only to the coldest processor class. This ensures that the peak cluster temperature will not be negatively affected by the execution of a hot task on a hot processor, but possibly increases the time that the task would have to wait in the queue. By employing the favourable migration strategy, this overhead may be decreased by executing the new task earlier on one of the marginally hotter processors.

3.2 Simulation Framework

In order to evaluate the performance of the presented scheduling algorithms, we have developed a cluster simulator within the SimGrid[3] framework [10]. To sim-

[3] SimGrid provides a framework for task scheduler simulations. Tasks are characterized by a cost (the run-time) and processors are weighted with a relative performance.

plify the simulation, we model a simple six-processor cluster, with each processor representing a blade chassis in a typical six-chassis IBM BladeCenter. Each processor has an associated fan-input temperature; the fan-input temperatures vary linearly from 18.6 °C to 23.8 °C (corresponding to the temperatures measured for a six-chassis BladeCenter). Note that the maximum steady-state operating temperature of modern microprocessors is typically around 60 °C.

3.3 Results and Discussion

Figure 6 presents the total execution time and peak temperature results of the scheduler simulations having a global queue length of 6000 tasks (1000 instances of each of the six profiled SPEC'2K benchmarks). As expected, the $FCFS_6$ strategy has the fastest queue execution time of 288852 s, which represents nearly 100 % processor utilization, yet also results in the highest peak temperature (48.0 °C). The effect of turning off hot processors is shown in $FCFS_5$ through $FCFS_1$. In these results we see that the execution time increases proportionally to the number of processors turned off. Further, the peak temperatures decrease as the hotter processors are turned off.

By incorporating temperature-awareness into the scheduler, PTS results in the lowest temperature (42.8 °C, which corresponds to the ideal placement of the workload on the processors) but increases execution time to 712000 s, indicating that a number of processors were idle for a large portion of time. By allowing favourable migration, the PTS-FM strategy maintains the optimal temperature, but compared to PTS improves the execution time to 564616 s.

Finally, the strategies denoted by PTS-UM_1 through PTS-UM_5 demonstrate that by taking a more conservative approach, we can allow for faster execution

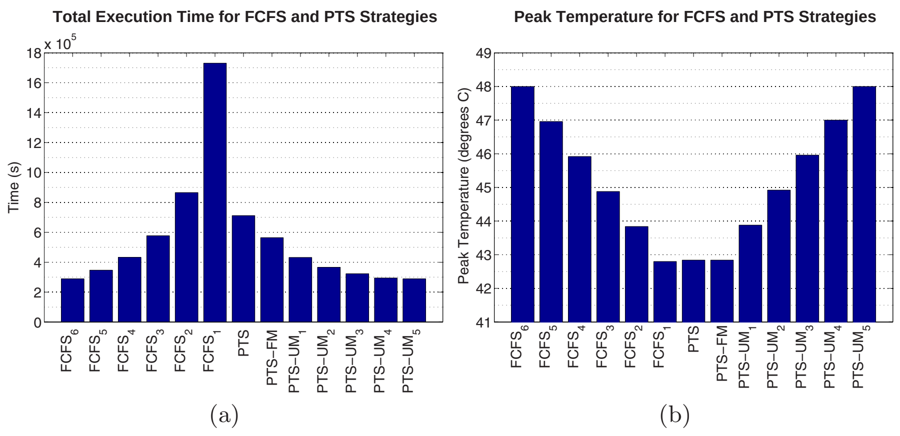


Fig. 6. Results for the FCFS and PTS scheduling strategies: (a) overall queue servicing time and (b) cluster peak temperature

times at the expense of higher temperatures. When we compare the execution times of $FCFS_n$ and PTS-UM_n strategies that have similar peak temperatures, the PTS-UM_n method performs better. For example, both PTS-UM_3 and $FCFS_4$ provide a decrease in peak temperature of approximately 2 °C. However, PTS-UM_3 achieves this at a slight cost in execution time in comparison to the 100000 s penalty created by $FCFS_4$.

It is important to note that the numerical results shown here are highly dependent on the cluster configuration and task profiles used. For example, the overall peak temperature reduction from 48.0 °C to 42.8 °C corresponds to the best possible placement with this set of tasks and processors. When used in environments with more diverse task temperature profiles, the PTS strategies are expected to achieve more dramatic temperature reductions.

As presented, the PTS schedulers can be used to decrease the cluster peak temperature. In cases of uncompromised temperature performance, system designers are encouraged to use the PTS-FM strategy, as it realizes the optimal temperature while minimizing the possible execution time (i.e. any decrease in the execution time would have required an increase in the peak temperature). In cases where the system designer is satisfied with a sub-optimal peak temperature, one of the PTS-UM_n strategies can be used to improve execution time. The PTS-UM_n strategy thus provides a mechanism for system designers to tune the time/temperature performance trade-off according to institutional priorities.

4 Related Work

Temperature-aware task scheduling for the computational cluster is a relatively new field of study. Bianchini and Rajamony presented a review of the energy and power management issues in computational clusters [3]. Much of the initial work in this area has been performed by Hewlett Packard ([4]-[7]). For example, in [6] Moore et al. present algorithms that leverage a thermodynamic formulation of steady-state hot- and cold-spots to achieve up to a factor of two reduction in cooling costs.

Patel et al. have presented a workload placement strategy for global computational Grids in which computational facilities are assigned energy-efficiency coefficients and workloads are allocated to the cluster having the best energy characteristics [8]. Weissel and Bellosa have developed OS-level power and thermal management methods that can be used to improve the thermal characteristics of the data center [9].

A recent study by Kurson et al. at IBM T. J. Watson Research Center investigated the decrease in on-chip temperatures seen while using thermally-aware thread scheduling [12]. Their *MinTemp* scheduling policy effectively lowers on-chip temperature by selecting the thread which has the lowest temperature for the current cycle's hottest thermally critical block.

In our work, we study task temperature profiles and exploit them to develop a temperature-aware data center task scheduler. This approach operates at the macroscopic system level to discover thermally beneficial workload placements within the data center.

5 Conclusion

We introduced temperature-aware scheduling policies for computational clusters. We used a task's temperature profile to quantify a task's heat producing capacity and to differentiate between cold and hot tasks. Our policies use different approaches to trade queue servicing time for lower peak temperatures. With the cluster configuration and temperature profiles we used, we conclude that a relatively balanced scheduling policy can effectively reduce cluster peak temperature at the expense of an increase in the queue servicing time.

In the future we plan to improve this work by evaluating the presented strategies using a more diverse set of tasks. Additionally, we will introduce the notion of task temperature profiling into the knapsack-based scheduler [11] in order to determine the potential for temperature-awareness in the presence of complex Quality-of-Service policies.

References

1. Srinivasan, J., Adve, S.V., Bose, P., Rivers, J.A.: The Impact of Technology Scaling on Processor Lifetime Reliability. In: Proceedings of the International Conference on Dependable Systems and Networks (DSN-2004) (June 2004)
2. Borkar, S.: Design challenges of technology scaling. IEEE Micro., 23–29 (July-August 1999)
3. Bianchini, R., Rajamony, R.: Power and Energy Management for Server Systems. IEEE Computer 37(11) (November 2004)
4. Patel, C.D.: A vision of energy aware computing - from chips to data centers. In: The International Symposium on Micro-Mechanical Engineering. ISMME2003 -K 1 5 (December 2003)
5. Moore, J., et al.: Going Beyond CPUs: The Potential of Temperature-Aware Solutions for the Data Center. In: Proc. 1st Workshop Temperature-Aware Computer Systems, `www.cs.virginia.edu/~skadron/tacs/rang.pdf`
6. Moore, J., Chase, J., Ranganathan, P., Sharma, R.: Making Scheduling "Cool":Temperature-Aware Workload Placement in Data Centers. In: Proceedings of the USENIX Annual Technical Conference, Anaheim, CA, April 2005 (2005)
7. Moore, J., et al.: A Sense of Place: Towards a Location-aware Information Plane for Data Centers. Hewlett Packard Technical Report TR 2004-27
8. Patel, C.D., Sharma, R.K., Bash, C.E., Graupner, S.: Energy Aware Grid: Global Workload Placement based on Energy Efficiency. In: IMECE 2003-41443. International Mechanical Engineering Congress and Exposition, Washington, DC (2003)
9. Weissel, A., Bellosa, F.: Dynamic thermal management for distributed systems. In: Proceedings of the First Workshop on Temperature-Aware Computer Systems (TACS'04) (June 2004)

10. Casanova, H.: Simgrid: A toolkit for the simulation of application scheduling. In: Proceedings of the IEEE Symposium on Cluster Computing and the Grid (CC-Grid'01) (May 2001)
11. Parra-Hernandez, R., Vanderster, D., Dimopoulos, N.J.: Resource Management and Knapsack Formulations on the Grid. In: Proceedings of Grid 2004 - 5th IEEE/ACM International Workshop on Grid Computing, November 2004, pp. 94–101. ACM Press, New York (2004)
12. Kursun, E., Cher, C-Y., Buyuktosunoglu, A., Bose, P.: Investigating the Effects of Task Scheduling on Thermal Behaviori. In: Third Workshop on Temperature-Aware Computer Systems, Held in conjunction with ISCA-33, Boston, MA, USA, June 17-21, 2006 (2006)

Runtime Performance Projection Model for Dynamic Power Management

Sang-Jeong Lee[1], Hae-Kag Lee[1], and Pen-Chung Yew[2]

[1] Department of Computer Science and Engineering, Soonchunhyang University
Asan, Choongnam, 336-745, Korea
{sjlee,lhk7083}@sch.ac.kr
[2] Department of Computer Science and Engineering, University of Minnesota
Minneapolis, MN 55455
yew@cs.umn.edu

Abstract. In this paper, a runtime performance projection model for dynamic power management is proposed. The model is built as a first-order linear equation using a linear regression model. It could be used to estimate performance impact from different p-states (voltage-frequency pairs). Workload behavior is monitored dynamically for a program region of 100M instructions using hardware performance monitoring counters (PMCs), and performance for the next region is estimated using the proposed model. For each 100M-instructions interval, the performance of all processor p-states is estimated and the lowest frequency is selected within specified performance constraints. The selected frequency is set with a low-overhead DVFS-based (dynamic voltage-frequency scaling) p-state changing mechanism for the next program region. We evaluate the performance degradation and the amount of energy saving of our dynamic power management scheme using the proposed projection model for SPEC CPU2000 benchmark on a Pentium M platform. We measure the execution time and energy consumption for 4 specified constraints – 10%, 20%, 40%, 80%, on the maximum allowed performance degradation. The result shows that our dynamic management scheme saves energy consumption by 3%, 18%, 38% and 48% with a performance degradation of 3%, 19%, 45% and 79% under 10%,20%,40% and 80% constraints, respectively.

Keywords: Dynamic Power Management, Dynamic Voltage-Frequency Scaling, Performance Monitoring.

1 Introduction

Power-aware computing has become a critical component of computer system design. In high-performance systems, thermal dissipation has always been a major challenge. Also, in mobile and embedded systems, energy efficiency is critical to extend battery life. Although hardware design has a direct impact on the system’s power consumption, application workload is also an important contributor to power consumption. Therefore, dynamic power management considering application workload is critical to an efficient power management of a computer system [5][7][11][15].

© Springer-Verlag Berlin Heidelberg 2007

Current power-aware microprocessors provide multiple operating frequency-voltage pairs for dynamic power management using dynamic voltage and frequency scaling (DVFS) techniques. DVFS can trade off system performance with power consumption. Processors that support DVFS have to balance the achieved energy savings with a pre-determined limit of performance impact on applications. Advanced Configuration and Power Interface (ACPI) is one of industrial standards that define active (p-states) and standby power management for the processors [17].

In general, the processor's power consumption is greatly dependent on its executing workload. During a program execution, performance and power consumption can vary widely according to its program behavior. Such workload characteristics can be exploited for power management. When a processor is idle waiting for its memory accesses, power consumption can be reduced by scaling down processor frequency and voltage without a significant performance loss. If we are able to find the slack points in a program region dynamically, power management decisions can be made with the help of an accessing model that estimates their effects on performance and power savings.

In this paper, a runtime performance projection model for dynamic power management is proposed. The model is used to estimate the performance impact of different p-states (voltage-frequency pairs). Workload behavior such as CPI (average cycles per instruction) and the number of memory accesses are monitored dynamically for a program region using hardware performance monitoring counters (PMCs). Performance for the next region is estimated using the proposed performance projection model with the information collected from PMCs. Performance of all processor p-states is estimated and the lowest frequency is selected within the specified performance constraints. The selected frequency is set with a low-overhead DVFS-based p-state change mechanism for the next program region.

We build a linear regression model that relates processor performance to two architectural parameters on a real Pentium-M processor. The proposed performance estimation model is a first-order linear equation that predicts performance impact on CPI by a given p-state using monitored program activities such as CPI and the number of memory accesses. We evaluate the performance degradation and energy saving using the proposed projection model for SPEC CPU2000 benchmark on a Pentium M platform. We measure the execution time and energy consumption under 4 pre-determined constraints – 10%, 20%, 40%, 80%, that specify maximum performance degradation allowed. The results show that our dynamic management scheme could save energy consumption by 3%, 18%, 38% and 48% with a performance degradation of 3%, 19%, 45% and 79%, under 10%,20%,40% and 80% constraints, respectively.

2 Related Work

There have been many studies that investigate power and energy models for power management using DVFS. Among them, Contreras et al. develop a linear power estimation model to estimate run-time CPU and memory power consumption of the Intel PXA255 processor using hardware performance counters. They derive coefficients of the equation by minimizing the difference between estimated power consumption and actual measured power consumption using linear algebra [2].

Wu et al. propose an analytic DVFS decision model to determine new p-states in a dynamic compilation environment. They develop an equation to select a p-state using CPU execution slack due to asynchronous memory accesses [15]. Isci et al. propose a runtime phase predictor, called GPHT predictor. It monitors workload behavior for dynamic power management on Pentium-M processor [7]. After every 100M microoperations, they predict a new p-state based on Wu's DVFS decision model and apply a new DVFS setting.

Rajamani et al. developed two models based on performance counter events such as decoded instructions per cycle and data cache miss to decide a p-state for power management [11]. Their performance estimation model applies performance projection across all p-states and the p-state with the lowest frequency that meets the specified requirement is selected for next interval.

The power management scheme of Rajamani et al. is the closest to ours. However, their performance model uses both linear and non-linear equations, and they divide the workload into CPU-bound and memory-bound. Their sampling interval is 10ms. Our model is a linear model derived from a regression analysis on different performance parameters. We use a fixed instruction sampling interval that is less sensitive to clock frequency change. Previous research has shown that program phases can alter the timing and values of observed metrics [7]. Therefore, using a fixed instruction interval can eliminate the effect of timing variations.

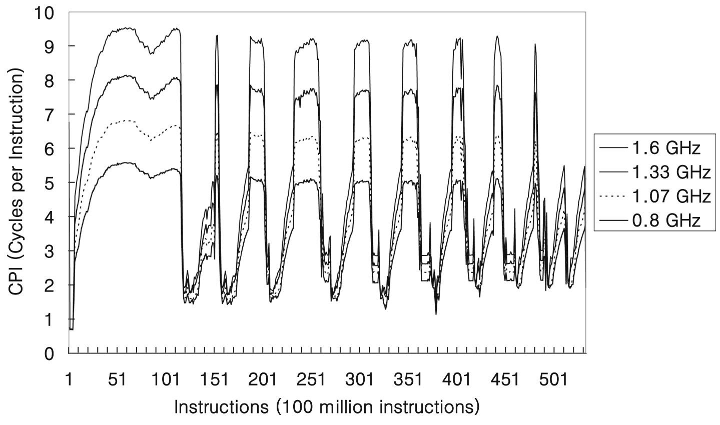
3 Runtime Performance Projection Model

3.1 Workload Behavior

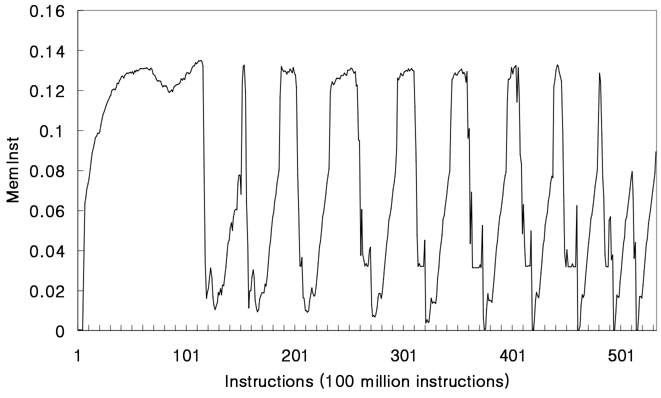
Many studies on program behavior show that programs exhibit repetitive execution phases, and their future behavior could be predicted by their monitored past behavior [6]. Figure 1 shows an example of program phase change for the SPEC CPU2000 benchmark mcf. All results are obtained using PMCs on Pentium-M processor (Model 730) in laptop computer. The Pentium M processor we experimented has 4 different DVFS-based ACPI-defined p-states (voltage-frequency pairs). Table 1 shows the four p-states of Pentium-M 730 model used in this study.

Table 1. Frequency and voltage pairs of Pentium-M (Model 730)

Frequency	Supply Voltage
1.6 GHz	1.308 V
1.33 GHz	1.212 V
1.07 GHz	1.1 V
0.8 GHz	0.988 V



(a) Performance Trace



(b) Memory access trace

Fig. 1. Performance and memory access traces for the SPEC2000 benchmark *mcf*

In Figure 1 (a), the y-axis is cycles per instruction (CPI) measured for every 100M instructions during the program execution. Figure 1 (b) shows the degree of memory accesses, MemInst. It is defined as the ratio between the memory bus transactions to the number of instructions retired. From the figure, the program has many repetitive phases, and its performance has a strong correlation to memory accesses. Figure 2 shows normalized performance and energy consumption across four p-states for mcf. Energy consumption is estimated using power equation described in section 5. All values are normalized toward those obtained using 1.6 GHz. For mcf, the performance has a strong impact on energy consumption across all p-states.

Considering the above workload behavior, we use a linear equation to model program performance for each p-state. Its coefficients will be obtained by a regression model detailed in the next section.

$$CPI_{f'} = \beta_0 + \beta_1 CPI_f + \beta_2 MemInst_f \qquad (1)$$

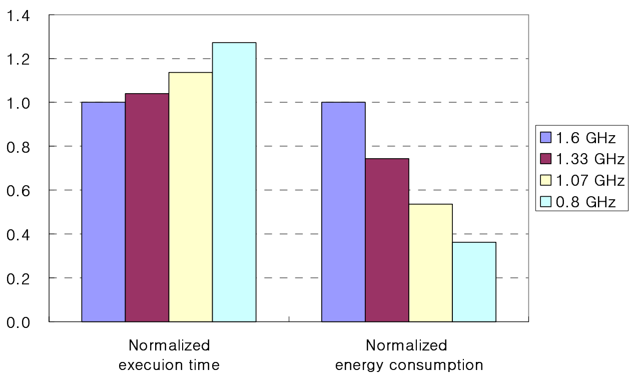


Fig. 2. Normalized performance and energy consumption across four p-states for *mcf*

In the equation (1), f and f' denote the two different frequencies of two different p-states. $CPI_{f'}$ is the estimated CPI at the p-state of f', and CPI_f is the measured CPI at the p-state of f. β_0, β_1, β_2 are regression coefficients. These coefficients indicate the relative significance of the corresponding terms obtained by the regression analysis.

3.2 Linear Regression Model

A regression model is a compact mathematical representation of the relationship between the response variable and the independent variables in a given design space [13]. Linear regression models are widely used to obtain predictions of the response variable at arbitrary points in the design space. Linear regression is represented as follows:

$$y_i = \beta_0 + \sum_{j=1}^{k} \beta_j x_{ji} + e_i \tag{2}$$

where y_i is the ith observation of the response variable which depends on the independent variables $x_1, x_2, \ldots, x_k$. The β_j are the coefficient of the variable x_j and an estimation of the value can provide the useful relationship. The e_i is the error term representing the deviation of the ith observation value (i.e., y_i) from the estimated value by the given linear equation.

A very elegant method for estimating β_j is the method of least squares. This method of estimation, which leads to estimates of certain optimal properties, is based on the appealing idea of choosing β_j to minimize the squares of the error term. That is, determining β_j that minimizes MSE (Mean Squares Error) is one of our goals.

$$MSE = \sum_{i=1}^{n} (y_i - \beta_0 - \sum_{j=1}^{k} \beta_j x_{ji})^2 \tag{3}$$

If the relationship equation could be derived, very useful information could also be obtained by using the equation. In this paper, CPI under a certain p-state is predicted by applying the values of CPI and MemInst under a different p-state using equation (1).

Another problem considered in the regression analysis is that of statistical testing for the null hypothesis $H_0 : \beta_j = 0$ (j=1,2,…, k). The F-test is a standard statistical method for testing the regression model, in which the total variation (SST) is decomposed into two terms: the variations due to linear regression (SSR) and the regression error (SSE). The definitions of SST, SSR, and SSE are

$$SST = \sum_{i=1}^{n} (y_i - \bar{y})^2 , \quad SSR = \sum_{i=1}^{n} (\hat{y}_i - \bar{y})^2 , \quad SSE = SST - SSR \tag{4}$$

where $\bar{y}$ is the mean of the observed values of response variables and $\hat{y}_i$ are the predicted values by the regression equation. The sample coefficient of determination R^2 can be calculated by

$$R^2 = \frac{SSR}{SST} , \ (0 \le R^2 \le 1) \tag{5}$$

R^2 provides the multiple correlation statistic, so the larger the value of R^2 and the smaller the value SSE the better the fit of the predicted value to the observations. F-statistic F_0 for the hypothesis test is defined as

$$F_0 = \frac{SSR}{k} \Big/ \frac{SSE}{n-k-1} \tag{6}$$

If the F-statistic value is lager than the given significance level which is referred to the F-distribution, the null hypothesis $H_0 : \beta_j = 0$ is rejected, in other words, the regression equation has significant meanings.

3.3 Linear Performance Model

We derive coefficients, β_j, for equation (1) by the above regression analysis. The experimental data were obtained from the execution of SPEC CPU2000 benchmark

Table 2. Estimated regression coefficients for each p-state

(a) Response variable $CPI_{1.6GHZ}$ case

f	β_0	β_1	β_2	R^2	F_0
1.3GHz	-0.16	1.01	9.85	0.99	2864261
1.07GHz	-0.02	1.00	20.39	0.98	1272807
0.8 GHz	-0.06	1.02	29.08	0.98	1736878

(b) Response variable $CPI_{1.3GHZ}$ case

f	β_0	β_1	β_2	R^2	F_0
1.3 GHz	0.04	0.96	-8.16	0.99	2417377
1.07GHz	-0.01	0.99	10.30	0.98	1978693
0.8GHz	-0.04	1.02	18.92	0.99	3343031

(c) Response variable $CPI_{1.07GHZ}$ case

f	β_0	β_1	β_2	R^2	F_0
1.6GHz	0.08	0.94	-17.31	0.97	910968
1.3GHz	0.04	0.98	-9.53	0.98	1769944
0.8GHz	-0.02	1.01	8.88	0.98	1707008

(d) Response Variable $CPI_{0.8GHZ}$ case

f	β_0	β_1	β_2	R^2	F_0
1.6 GHz	0.11	0.92	-25.39	0.96	835010
1.3 GHz	0.06	0.96	-17.91	0.99	2272772
1.07 GHz	0.04	0.97	-8.35	0.98	1536522

using the four p-states on Pentium-M processor. The values of each variable are sampled at every 100M instructions interval. And then, we derived the relationships between the CPI values of two different p-states including MemInst. For each pair (f,f'), the coefficients β_j are determined through the linear regression analysis. Table 2 shows the results of the regression analysis. The coefficients are obtained for the performance prediction of each p-state in equation (1). For example, Table 2 (a) shows the coefficients to estimate $CPI_{1.6GHZ}$ (f' = 1.6 GHz) from CPI_f and $MemInst_f$ when f=1.33 GHz, 1.07 GHz, 0.8 GHz in equation (1). Table 2, also, shows the values of R^2 and F_0. For all cases they are sufficiently large, so we can conclude that the null hypothesis $H_0 : \beta_j = 0$ is rejected, that is, the regression equation has significant meanings. Figure 3 shows the plotting of pairs between the observed values and the predicted values. It shows a very good fit of the model to real observed experimental data.

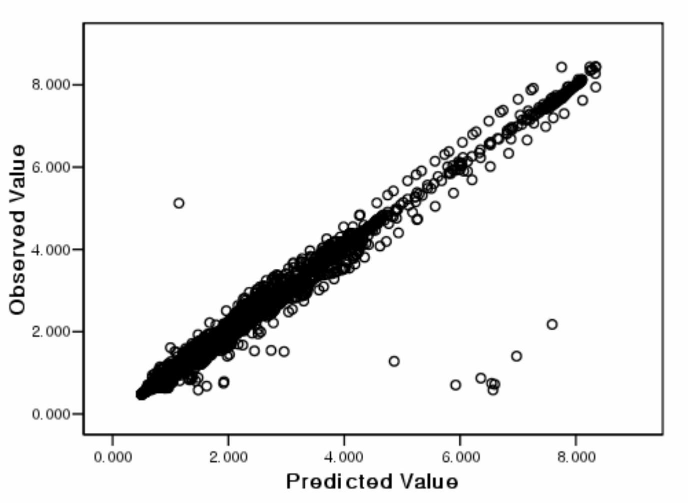


Fig. 3. A plotting of pairs (observed value, predicted value)

4 Dynamic Power Management

Using the linear regression model for performance estimation, we designed a dynamic power management framework on a Pentium-M processor (Model 730) using an off-the-shelf laptop computer (Toshiba Satellite A80) running Linux kernel 2.6-19. Workload behavior is monitored with PMCs dynamically for a program region of every 100M instructions. To eliminate the effect of timing variations across a p-state change during monitoring, we monitor workload behavior at a fixed interval of 100M instructions. After 100M instructions a performance monitoring interrupt (PMI) handler is invoked. The PMI handler is implemented as a loadable kernel module on Linux kernel. The PMI monitors application execution through two PMCs for retired instructions (INST_RETIRED event), memory bus transaction (BUS_TRAN_MEM event) and a time stamp counter (TSC) for clocks on the Pentium-M processor.

Figure 4 shows our dynamic power management framework. From the monitored values, CPI (the ratio of clock cycles to instructions retired) and MemInst (the ratio of memory bus transactions to instructions retired) are calculated. Performance impact is then estimated using our projection model, equation (1), with the calculated parameters to determine a p-state for the next interval. CPIs of all the p-states except current p-state are estimated. After getting CPIs, the expected run times for all p-states are calculated. We choose the p-state with the lowest frequency within the specified performance constraint by comparing their estimated run times. Then, the new p-state with the selected frequency is applied to next interval. Before exiting the PMI handler, the PMCs and a TSC are reinitialized and the counters are restarted.

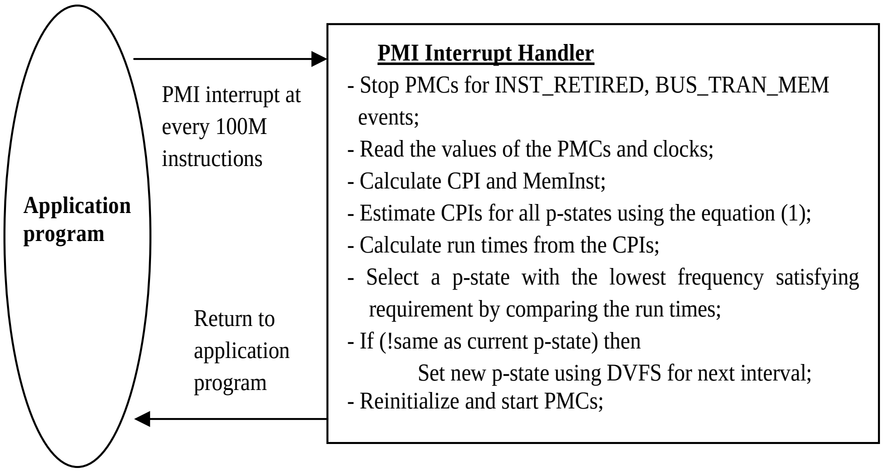

Application program
PMI interrupt at every 100M instructions
Return to application program
PMI Interrupt Handler
- Stop PMCs for INST_RETIRED, BUS_TRAN_MEM events;
- Read the values of the PMCs and clocks;
- Calculate CPI and MemInst;
- Estimate CPIs for all p-states using the equation (1);
- Calculate run times from the CPIs;
- Select a p-state with the lowest frequency satisfying requirement by comparing the run times;
- If (!same as current p-state) then
Set new p-state using DVFS for next interval;
- Reinitialize and start PMCs;

Fig. 4. Our dynamic power management framework with the runtime performance projection model

5 Evaluation Results

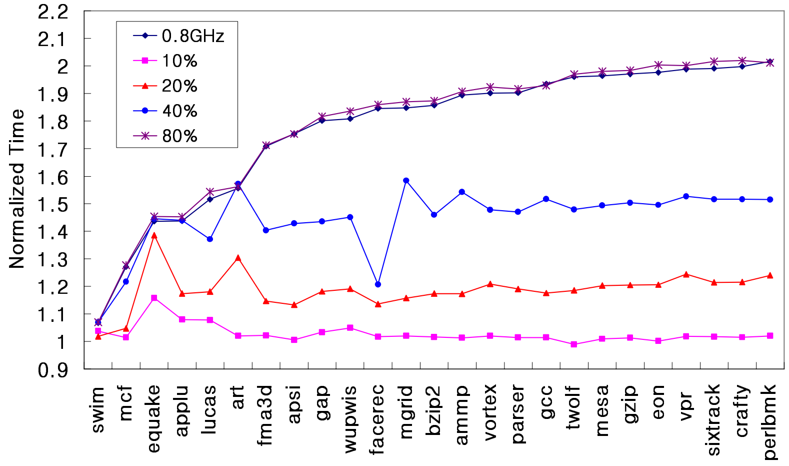
For evaluation, we use SPEC CPU2000 benchmark (178.galgel benchmark is excluded because of compile error). We measured execution time and computed energy consumption for each benchmark program. We could not measure actual energy used but calculated the energy consumption instead using the following equation for CMOS circuit.

$$E = PT = CV^2 fT \tag{7}$$

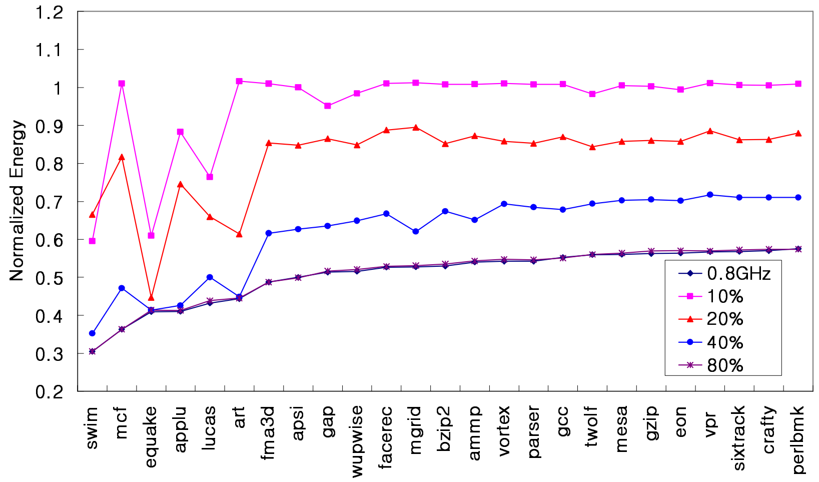
where P is the dynamic power, C is the switched capacitance, V is the supply voltage, f is the clock frequency and T is the total execution time of the program. Let's assume the program is executed on the processor with m p-states and DVFS is applied for dynamic power management. We define

$$E = \sum_{i=1}^{m} E_i = \sum_{i=1}^{m} (C_i f_i V_i^2 T_i) \approx C \sum_{i=1}^{m} f_i V_i^2 T_i \tag{8}$$

where E_i is the total energy consumption using i^{th} p-state, and C_i, f_i, V_i and T_i are switched capacitance, clock frequency, supply voltage and execution time for the i^{th} p-state. We assume all the switched capacitances are the same. By measuring program execution for each p-state, we can get their relative energy consumptions.



(a) Normalized execution time



(b) Normalized energy consumption

Fig. 5. Execution time and energy consumption normalized to 1.6GHz result for SPEC CPU2000 benchmark

Figure 5 shows the results of execution time and energy consumption normalized to the values at 1.6 GHz which is the maximum frequency allowed. It also shows the results at 0.8 GHz, the minimum-frequency among all p-states, and our power management under the performance constraints of 10%, 20%,40% and 80%, respectively. Here, 10% performance constraint means the performance degradation should be less than 10% comparing to the maximum performance at 1.6GHz. *Swim, mcf, equake, applu, lucas, art, fma3d* on the left are memory-bound applications (MemInst are above 0.01). They have the least performance loss and most energy saving using dynamic power management. The benchmarks on the right are CPU-bound applications. They have the least energy saving with most performance degradation. In Figure 5 (a), on average, the values of normalized time are 1.03, 1.19, 1.45 and 1.79 for 10%, 20%,40% and 80% performance constraints, respectively. There is only one violation at the 40% performance constraint. Almost all benchmarks in Figure 5 (a), except at 20% in which it's results are close to the constraint, are well within their required tolerance. In the 10% case, our performance model estimates the performance too conservatively. For the 40% and 80% cases, memory-bound applications satisfy the requirements well but CPU-bound applications violate the constraints. We think this discrepancy is due to the error caused by four discrete p-states and the error from the CPI term in equation (1). But, in general, our performance projection model estimates performance well. The execution time for the 80% case is 1% greater than the minimum-frequency 0.8GHz because of the overhead of the PMI handler and DVFS overhead. Figure 5 (b) shows the energy consumption normalized to 1.6 GHz. The smaller value means more energy saving. Memory-bound applications show much energy saving with less performance degradation. On average, normalized energy consumption are 0.97,0.82,0.62,0.52 for 10%,20%, 40%,80% constraint, respectively.

Figure 6 shows average execution time and energy consumption normalized to 1.6GHz including the GPHT predictor. GPHT is a dynamic power management framework with a phase predictor that has a similar structure as a branch predictor [7].

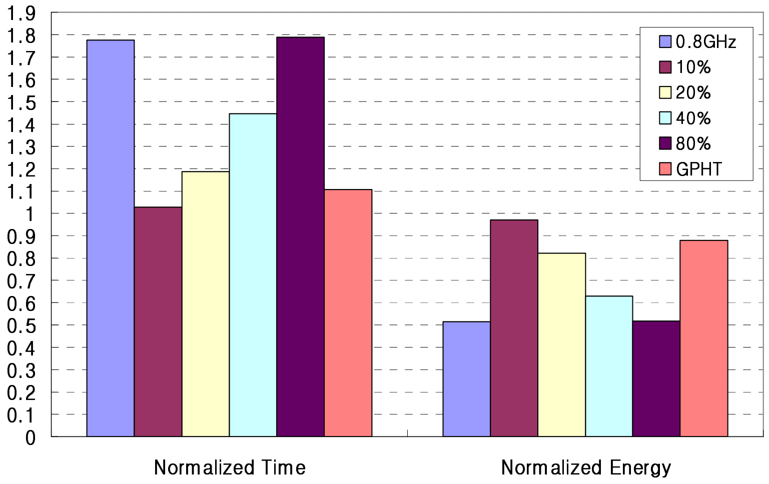


Fig. 6. Average execution time and energy consumption normalized to 1.6GHz including GPHT

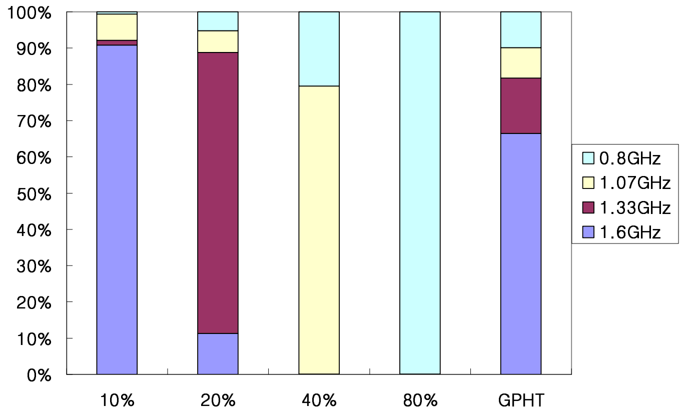


Fig. 7. Average breakdown of p-states for SPEC CPU2000 benchmark

We implemented it using the same value of Mem/micro-op (the ratio of memory bus transactions to micro-operations retired) to classify phases. It has no explicit performance requirement. The result shows that average normalized time is 1.11 and energy consumption is 0.88. GPHT shows better results comparing to the case of our 10% performance requirement. However, its energy saving is limited for less aggressive power management. Our power management for the 20% performance requirement saves much more energy. Figure 7 shows the average breakdown of p-states used during the execution for SPEC CPU2000 benchmark.

6 Conclusion

This paper presents a runtime performance projection model for dynamic power management. This model is used to predict performance impact of a p-state (voltage-frequency pairs). Using linear regression analysis, a first-order linear equation is built to estimate performance impact from monitored activity information such as CPI (cycles per instruction) and the number of memory accesses. We develop a dynamic power management framework using the performance projection model. It monitors and estimates workload behavior through hardware performance monitors sampled at every 100M instructions on Pentium-M processor. At each sample it estimates the performance of all allowable p-states. The lowest frequency that still meets a pre-determined level of tolerance in performance degradation is selected for next execution interval. We experiment our framework with SPEC CPU2000 benchmark. The result shows that our dynamic management saves energy consumption by 3%, 18%, 38% and 48% with a performance degradation of 3%, 19%, 45% and 79% under the pre-determined performance degradation tolerance of 10%,20%,40% and 80%, respectively. When we compare it with the GPHT power management scheme [7] that has 12% energy saving with 11% performance loss, our management framework can provide more energy saving.

References

1. Choi, K., Cheng, W., Pedram, M.: Frame-based Dynamic Voltage and Frequency Scaling for an MPEG Player. Journal of Low Power Electronics, American Scientific Publishers 1(1), 27–43 (2005)
2. Contreras, G., Martonosi, M.: Power Prediction for Intel XScale Processors Using Performance Monitoring Unit Events. In: International Symposium on Low Power Electronics and Design (ISLPED'05) (August 2005)
3. Intel Corp.: IA-32 Intel Architecture Software Developer's Manua: vol. 3 System Programming Guide (2005)
4. Intel Corp.: Enhanced Intel SpeedStep ® Technology and Demand-Based Switching on Linux, http://www.intel.com/cd/ids/developer/asmo-na/eng/195910.htm?prn=Y
5. Isci, C., Martonosi, M., Buyuktosunoglu, A.: Long-term Workload Phases: Duration Predictions and Applications to DVFS. IEEE MICRO. 25 (2005)
6. Isci, C., Martonosi, M.: Phase Characterization for Power: Evaluating Control-Flow-Based and Event-Counter-Based Techniques. In: Proceedings of 12th International Symposium on High-Performance Computer Architecture (HPCA-12) (Feburary 2006)
7. Isci, C., Contreras, G., Martonosi, M.: Live, Runtime Phase Monitoring and Prediction on Real Systems with Application to Dynamic Power Management. In: Proceedings of the 39th International Symposium on Microarchitecture (MICRO-39) (December 2006)
8. Nienhuser, D.: Power Management Guide, http://www.gentoo.org/doc/en/power-management-guide.xml
9. Poellabauer, C., Zhang, T., Pande, S., Schwan, K.: An Efficient Frequency Scaling Approach for Energy-Aware Embedded Real-Time Systems. In: Beigl, M., Lukowicz, P. (eds.) ARCS 2005. LNCS, vol. 3432, Springer, Heidelberg (2005)
10. Rajamani, K., Hanson, H., Rubio, J., Ghiasi, S., Rawson, F.: Online Power and Performance Estimation for Dynamic Power Management. IBM Technical Report RC24007, IBM Research (July 2006)
11. Rajamani, K., Hanson, H., Rubio, J., Ghiasi, S., Rawson, F.: Application-Aware Power Management. In: IEEE International Symposium on Workload Characterization (IISWC-2006), October 2006, IEEE Computer Society Press, Los Alamitos (2006)
12. Sazeides, Y., Kumar, R., Tullsen, D., Constantinou, T.: The Danger of Interval-Based Power Efficiency Metrics: When Worst Is Best. IEEE Computer Architecture Letters 4 (January 2005)
13. Seber, G., Lee, A.: Linear Regression Analysis. Wiley-Interscience, Chichester (2003)
14. Sprunt, B.: Pentium 4 Performance-Monitoring Features. IEEE MICRO 22(4) (2002)
15. Wu, Q., Reddi, V., Lee, J., Connors, D., Brooks, D., Martonosi, M., Clark, D.: Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance. In: Proceedings of the 38th International Symposium on Microarchitecture (MICRO-38), November 2005 (2005)
16. Zhu, Z., Zhang, X.: Look-ahead Architecture Adaptation to Reduce Processor Power Consumption. IEEE MICRO 25(4) (2005)
17. Advanced Configuration and Power Interface Specification 3.0, http://www.acpi.info

A Power-Aware Alternative for the Perceptron Branch Predictor

Kaveh Aasaraai and Amirali Baniasadi

University of Victoria, Victoria BC, V8P 3Y9, Canada
{aasaraai,amirali}@ece.uvic.ca

Abstract. The perceptron predictor is a highly accurate branch predictor. Unfortunately this high accuracy comes with high complexity. The high complexity is the result of the large number of computations required to speculate each branch outcome.

In this work we aim at reducing the computational complexity for the perceptron predictor. We show that by eliminating unnecessary data from computations, we can reduce both predictor's power dissipation and delay. We show that by applying our technique, predictor's dynamic and static power dissipation can be reduced by up to 52% and 44% respectively. Meantime we improve performance by up to 16% as we make faster prediction possible.

1 Introduction

The perceptron branch predictor is highly accurate. The high accuracy is the result of exploiting long history lengths [1] and is achieved at the expense of high complexity.

Perceptron relies on exploiting behavior correlations among branch instructions. To collect and store as much information as possible, perceptron uses several counters per branch instruction. Such counters use multiple bits and record how each branch correlates to previously encountered branch instructions. The predictor uses the counters and performs many steps before making the prediction. These steps include reading the counters and the outcome history of previously encountered branches and calculating the vector product of the two.

In this study we introduce power optimizations for perceptron. We show that while the conventional scheme provides high prediction accuracy, it is not efficient from the energy point of view. This is mainly due to the observation that not all the computations performed by perceptron are necessary. In particular, computations performed on counter lower bits are often unnecessary as they do not impact the prediction outcome. We exploit this phenomenon and reduce power dissipation by excluding less important bits from the computations.

We rely on the above observation and suggest eliminating the unnecessary bits from the computation process. We propose an efficient scheme to reduce the number of computations and suggest possible circuit and system level implementations.

© Springer-Verlag Berlin Heidelberg 2007

Power optimization techniques often trade performance for power. In this work, however, we not only reduce power but also improve processor performance. Performance improvement is possible since eliminating unnecessary computations results in faster and yet highly accurate prediction. We reduce the dynamic and static power dissipation associated with predictor computations by 52% and 44% respectively while improving performance up to 16%.

The rest of the paper is organized as follows. In Section 2 we discuss perceptron background. In Section 3 we discuss the motivation. In Sections 4 we introduce our optimization. In Section 5 we explain our simulations methodology and report results. In Section 6 we discuss related work. In Section 7 we offer concluding remarks.

2 Background

The perceptron branch predictor [1] uses multiple weights to store correlations among branch instructions. For each branch instruction, perceptron uses a weight vector which stores the correlation between the branch and previously encountered branch instructions.

As presented in Figure 1, the perceptron predictor takes the following steps to make a prediction. First, the predictor loads the weight vector corresponding to the current branch instruction. Second, each weight is multiplied by the corresponding outcome history from the history vector. Third, an adder tree computes the sum of all the counters. Fourth, the predictor makes prediction based on the sum's sign. For positive summations, the predictor assumes a *taken* branch otherwise the predictor assumes a *not taken* branch.

The outcome history is essentially a bit array, in which "0"s and "1"s represent not taken and taken outcomes respectively. However, in the multiplication

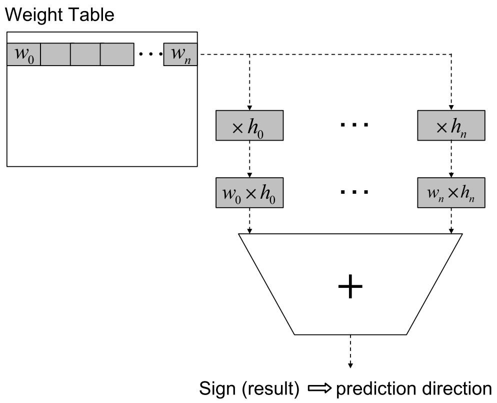


Fig. 1. The perceptron branch predictor using weight vector and history vector. The dot product of the two vectors is used to make the prediction.

process, “0”s are treated as “-1”, meaning that the corresponding weight must be negated.

At the update time, the behavioral correlation among branch instructions is recorded. The predictor updates the weights vector using the actual outcome of the branch instruction. Each weight is incremented if branch’s outcome conforms to the corresponding outcome history. Otherwise the weight is decremented.

3 Motivation

As presented in Figure 1, the perceptron predictor uses two vectors per branch instruction. For every direction prediction, the predictor computes the dot product of the two vectors.

The complexity of the computations involved in the dot product calculation makes it a slow and energy hungry one. This process requires an adder tree, with a size and complexity proportional to the size of the vectors and weights’ widths. The wider the weights are, the more complex the summation process will be. Note that the perceptron predictor proposed in [1] uses 8-bit counters to achieve the best accuracy. Furthermore, in order to achieve high accuracy, long history lengths, resulting in long vectors, are required [1]. This also substantially increases adder tree’s size and complexity.

In this study we show that the conventional perceptron predictor is not efficient from the energy point of view. This is the result of the fact that not all counter bits are equally important in making accurate predictions. Accordingly, higher order bits of the weights play a more important role in deciding the final outcome compared to lower order bits. In Figure 2 we present an example to provide better understanding.

History / Weight Values

h_i	-1	-1	1	1	-1	1	-1	-1
w_i	25	3	-5	10	2	0	-9	7
$w_i \times h_i$	-25	-3	-5	10	-2	0	9	-7
Binary	$\underline{100}111$	$\underline{111}101$	$\underline{111}011$	$\underline{001}010$	$\underline{111}110$	$\underline{000}000$	$\underline{001}001$	$\underline{111}001$

Result (All Bits) = -25 + -3 + -5 + 10 + -2 + 0 + 9 + -7 = -23 → Not Taken
Result (High Bits) = -4 + -1 + -1 + 1 + -1 + 0 + 1 + -1 = -6 → Not Taken

Fig. 2. The first calculation uses all bits and predicts the branch outcome as “not taken”. The second one uses only higher bits (underlined) and results in the same direction.

To investigate this further, in Figure 3 we report how often excluding the lower n bits of each counter impacts prediction outcome. As reported, on average, 0.3%, 1.0%, 4.0%, 13.7%, and 25.8% of time eliminating the lower one to five bits results in a different outcome respectively. This difference is worst (45%) when the lower five bits are excluded for *bzip2* (see Section 5 for methodology).

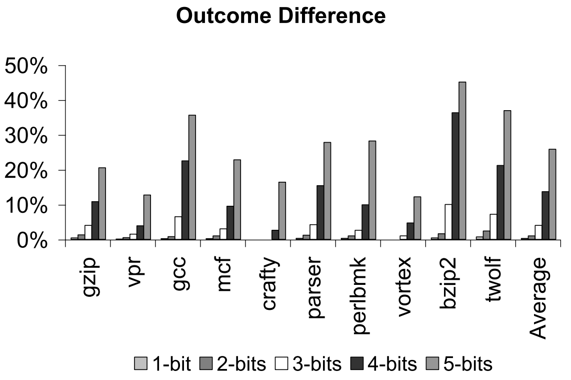


Fig. 3. How often removing the lower n bits from the computations results in a different outcome compared to the scenario where all bits are considered. Bars from left to right report for scenarios where one, two, three, four or five lower bits are excluded.

We conclude from Figure 3 that eliminating lower order bits (referred to as LOBs) of the weights from the prediction process and using only higher order bits (referred to as HOBs) would not significantly affect predictor's accuracy. We use this observation and reduce predictor's latency and power dissipation.

4 LOB Elimination

Considering the data presented in Section 3, we suggest eliminating the LOBs of the weights from the lookup and summation process. We modify the adder tree to bypass the LOBs of the weights, and perform the summation only over HOBs.

Excluding LOBs from the summation process reduces the size and complexity of the adder tree required. Therefore, a smaller and faster adder can be used. This will result in a faster and more power efficient lookup process.

As we show in this work, LOBs have very little impact on the prediction outcome at the lookup stage. However, it is important to maintain all bits, including LOBs, at the update stage. This is necessary to assure recording as much correlation information as possible. Therefore, we do not exclude LOBs at the update stage and increment/decrement weights taking into account all counter bits.

In Figure 4 we present the modified prediction scheme. The adder tree does not load or use all counter bits. Instead, the adder tree bypasses the LOBs of the weights, and performs the summation process only on the HOBs. Eliminating LOBs reduces power but can, in principle, impact accuracy and performance.

4.1 Accuracy vs. Delay

By eliminating LOBs from the lookup process, we reduce the prediction latency at the cost of accuracy loss. However, a previous study on branch prediction

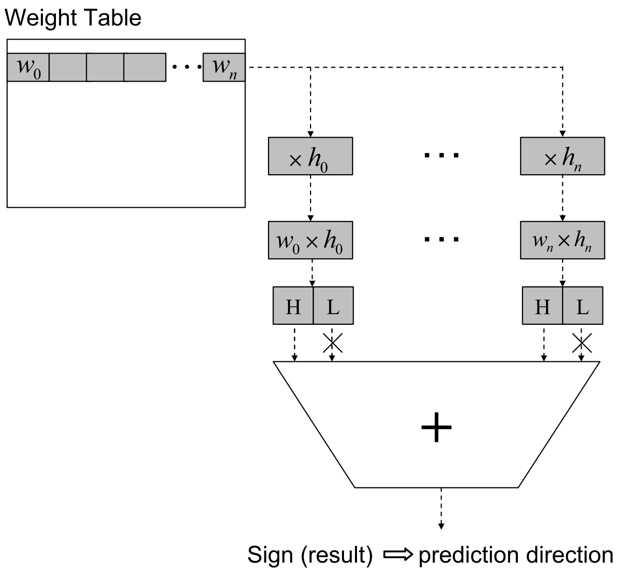


Fig. 4. The optimized adder tree bypasses LOBs of the elements and performs the addition only on the HOBs. Eliminating LOBs results in a smaller and faster adder tree.

delay shows that a relatively accurate single-cycle latency predictor outperforms a 100% accurate predictor with two cycles of latency [2].

To investigate whether the same general trade-off is true for perceptron, we study if the prediction speedup obtained by eliminating LOBs is worth the accuracy cost. In Section 5 we show that for the benchmarks used in this work the performance improvements achieved by faster prediction outweigh the cost associated with the extra mispredictions.

4.2 Power

By eliminating LOBs from the lookup process, we reduce both the dynamic and the static power dissipated by the predictor. First, fewer bits are involved in the computations, reducing the overall activity and dynamic power. Second, as we reduce the adder tree's size, we exploit fewer gates, reducing the overall static power.

As we eliminate LOBs from the computations necessary at the prediction time, reading all bit lines of the weight vector is no longer necessary. One straightforward mechanism to implement this is to decouple LOBs and HOBs. To this end, we store LOBs and HOBs in two separate tables.

As presented in Figure 5, at the prediction time, the predictor accesses only the tables storing HOBs, saving the power dissipated for accessing LOBs in the conventional perceptron predictor. Note that while we save the energy spent on wordline, bitline and sense amplifiers, we do not reduce the decoder power dissipation as we do not reduce the number of table entries.

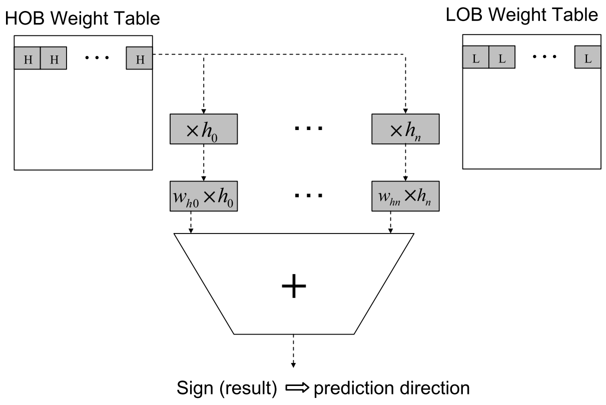


Fig. 5. Predictor table is divided to HOB and LOB tables. Only the HOB table is accessed at the prediction time.

5 Methodology and Results

For our simulations, we modify the SimpleScalar tool set [3] to include the conventional perceptron branch predictor and our proposed optimization. We use Simpoint [4] to identify representative 500 million instruction regions of the benchmarks. We use a subset of SPEC2K-INT benchmarks for our simulations.

Table 1 reports the baseline processor used in our study. For predictor configuration, we use the 64Kb budget global perceptron predictor proposed in [1].

For predictor power and timing reports, we use Synopsys Design Compiler synthesis tool assuming the 180 nm technology. We use the high effort optimization option of the Design Compiler, and optimize the circuit for delay. We simulated both the conventional and the optimized perceptron predictors.

Since we assume that table read time remains intact, for timing reports, we only measure the time the adder tree requires.

For our simulations, we assume the processor has a 1GHz frequency.

For simplicity, we use the notation of PER-n to refer to a perceptron predictor modified to eliminate the lower n bits from the computations.

Table 1. Processor Microarchitectural Configurations

Fetch/Decode/Commit	6
BTB	512
L1 I-Cache	32 KB, 32B blk, 2 way
L1 D-Cache	32 KB, 32B blk, 4 way
L2 Unified-Cache	512 KB, 64B blk, 2 way
L2 Hit Latency	6
L2 Miss Latency	100
Predictor Budget	64Kbits

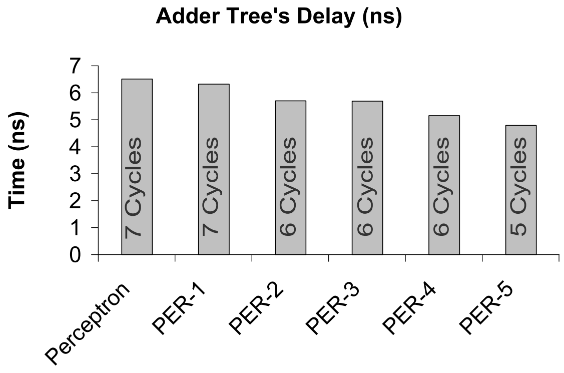


Fig. 6. Time/cycle required to compute the dot product

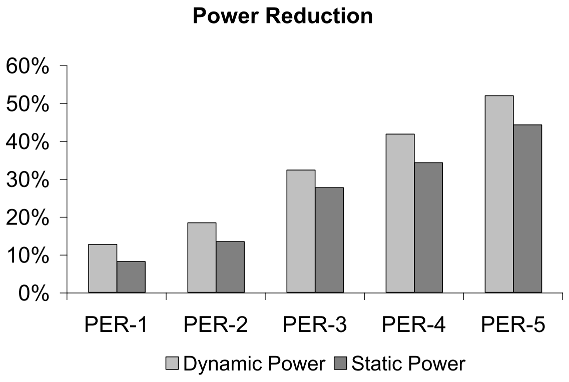


Fig. 7. Power reduction for the adder tree compared to the conventional perceptron. Results are shown for PER-1, PER-2, PER-3, PER-4 and PER-5.

5.1 Timing

Figure 6 reports time (in nanoseconds) and the number of cycles required to compute the predictor computation result. We report results for the original perceptron predictor and five optimized versions. As reported, the original predictor takes 7 clock cycles to compute the result. By eliminating one bit from the computation process no clock cycle is saved. However, removing two, three or four bits saves one clock cycle and removing five bits saves two clock cycles. We use these timings in our simulations to evaluate the optimized predictors.

5.2 Power Dissipation

Figure 7 reports the reduction in both static and dynamic power dissipation for the predictor's adder tree. Results are obtained by gate level synthesis of the circuit. Eliminating one to five bits saves from 13% to 52% of the dynamic power

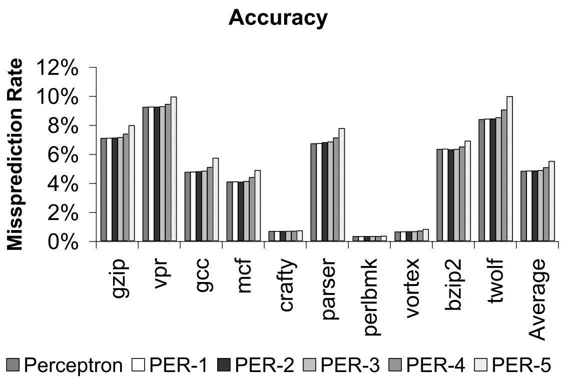


Fig. 8. Prediction accuracy for the conventional perceptron predictor and five optimized versions, PER-1, PER-2, PER-3, PER-4 and PER-5. The accuracy loss is negligible except for PER-5.

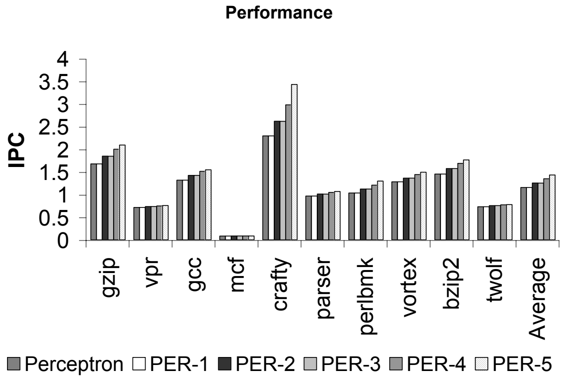


Fig. 9. Performance improvement compared to a processor using the conventional perceptron predictor. Results are shown for processors using PER-1, PER-2, PER-3, PER-4 and PER-5 predictors.

and 8% to 44% of the static power dissipation. This is the result of exploiting smaller adders.

5.3 Prediction Accuracy

As we use fewer bits to make predictions, we can potentially harm accuracy. To investigate this further in Figure 8 we compare prediction accuracy for six different predictors: The original perceptron predictor, PER-1, PER-2, PER-3, PER-4 and PER-5. As reported, average misprediction is 4.80% 4.81% 4.82% 4.85% 5.04% 5.48% for perceptron, PER-1, PER-2, PER-3, PER-4 and PER-5 respectively.

5.4 Performance

Figure 9 reports processor's overall performance compared to a processor using the original perceptron predictor. We report for five different processors using PER-1, PER-2, PER-3, PER-4 and PER-5 branch predictors. As reported, average IPCs are 1.15 1.15 1.25 1.25 1.35 1.43 for Perceptron, PER-1, PER-2, PER-3, PER-4 and PER-5 respectively.

Although the optimized perceptron predictor achieves slightly lower accuracy compared to the original one, the overall processor performance is higher. As explained earlier, this is the result of achieving faster prediction by eliminating LOBs.

6 Related Work

Vintan and Iridon [5] suggested Learning vector quantization (LVQ), a neural method for branch prediction. LVQ prediction is about as accurate as a table-based branch predictor. However, LVQ comes with implementation difficulties.

Aasaraai and Baniasadi [6] used the same technique used in this work on the O-GEHL branch predictor. They showed that power savings are possible by eliminating lower order bits from computations involved in the O-GEHL branch predictor. They also use disabling technique in [7] to improve the power efficiency of the perceptron branch predictor. They reduced perceptron power dissipation by utilizing as much resources as needed according to the branch behavior, effectively reducing overall number of computations.

Loh and Jimenez [8] introduced two optimization techniques for perceptron. They proposed a modulo path-history mechanism to decouple the branch outcome history length from the path length. They also suggested bias-based filtering exploiting the fact that neural predictors can easily track strongly biased branches whose frequencies are high. Therefore, the number of accesses to the predictor tables is reduced due to the fact that only bias weight is used for prediction.

Parikh et al. explored how branch predictor impacts processor power dissipation. They introduced banking to reduce the active portion of the predictor. They also introduced prediction probe detector (PPD) to identify when a cache line has no branches so that a lookup in the predictor buffer/BTB can be avoided [9].

Baniasadi and Moshovos introduced Branch Predictor Prediction (BPP) [10]. They stored information regarding the sub-predictors accessed by the most recent branch instructions executed and avoided accessing underlying structures. They also introduced Selective Predictor Access (SEPAS) [11] which selectively accessed a small filter to avoid unnecessary lookups or updates to the branch predictor.

Huang et al. used profiling to reduce branch predictor's power dissipation [12]. They disabled tables that do not improve accuracy and reduced BTB size for applications with low number of static branches.

Our work is different from all the above studies as it eliminates unnecessary and redundant computations for the perceptron predictor to reduce power. Unlike many previously suggested techniques, our optimizations do not come with any timing or power overhead as we do not perform any extra computation or use any additional storage.

7 Conclusion

In this work we presented an alternative power-aware perceptron branch predictor. We showed that perceptron uses unnecessary information at the prediction time to perform branch direction prediction. We also showed that eliminating such unnecessary information from the prediction procedure does not impose substantial accuracy loss. We reduced the amount of information used at the prediction time, and showed that it is possible to simplfiy the predictor structure, reducing both static and dynamic power dissipation of the predictor.

Moreover, we showed that by avoiding such computations it is possible to achieve faster branch prediction. Consequently, we improved the overall processor performance despite the slightly lower prediction accuracy.

References

1. Jimenez, D.A., Lin, C.: Neural methods for dynamic branch prediction. ACM Transactions on Computer Systems, 369–397 (2002)
2. Jimenez, D.A., Keckler, S.W., Lin, C.: The impact of delay on the design of branch predictors. In: Proceedings of the 33rd International Symposium on Microarchitecture (MICRO-33), pp. 66–77 (2000)
3. Burger, D., Austin, T.M.: The simplescalar tool set version 2.0. Technical report, Technical Report 1342, Computer Sciences Department, University of Wisconsin (June (1997)
4. Sherwood, T., Perelman, E., Hamerly, G., Calder, B.: Automatically characterizing large scale program behavior. In: Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (2002)
5. Vintan, L.N., Iridon, M.: Neural methods for dynamic branch prediction. In: International Joint Conference on Neural Networks, pp. 868–873 (1999)
6. Aasaraai, K., Baniasadi, A., Atoofian, E.: Computational and storage power optimizations for the o-gehl branch predictor. In: ACM International Conference on Computing Frontiers, ACM Press, New York (2007)
7. Aasaraai, K., Baniasadi, A.: Low-power perceptron branch predictor. Journal of Low Power Electronics 2(3), 333–341 (2006)
8. Loh, G.H., Jimenez, D.A.: Reducing the power and complexity of path-based neural branch prediction. In: Proceedings of the 5th Workshop on Complexity Effective Design (WCED), pp. 1–8 (2005)
9. Parikh, D., Skadron, K., Zhang, Y., Barcella, M., Stan, M.: Power issues related to branch prediction. In: Proceedings of the Eighth International Symposium on High-Performance Computer Architecture. p. 233 (2002)

10. Baniasadi, A., Moshovos, A.: Branch predictor prediction: A power-aware branch predictor for high-performance processors. In: Proceedings of 20th International Conference on Computer Design (ICCD 2002), pp. 458–461 (2002)
11. Baniasadi, A., Moshovos, A.: Sepas: A highly accurate energy-efficient branch predictor. In: Proceedings of the 2004 International Symposium on Low Power Electronics and Design. pp. 38–43 (2004)
12. Huang, M.C., Chaver, D., Pinuel, L., Prieto, M., Tirado, F.: Customizing the branch predictor to reduce complexity and energy consumption. In: Proceedings of 33rd International Symposium on Microarchitecture, pp. 12–25 (2003)

Power Consumption and Performance Analysis of 3D NoCs

Akbar Sharifi and Hamid Sarbazi-Azad

HPCAN Lab, Computer Engineering Department,
Sharif University of Technology, Tehran, Iran
a_sharifi@ce.sharif.edu, azad@sharif.edu

Abstract. Nowadays networks-on-chip are emerging as a hot topic in IC designs with high integration. Much research has been done in this field of study recently, e.g. in routing algorithms, switching methods, VLSI Layout, and effects of resource allocation on system performance. On the other hand, three-dimensional integrated circuits allow a time-warp for Moore's Law. By vertically stacking two or more silicon wafers, connected with a high-density, high-speed interconnect, it is now possible to combine multiple active device layers within a single IC. In this paper, we examine performance and power consumption in a three dimensional network-on-chip structure under different types of traffic loads, routing algorithms, and switching methods. To the best of our knowledge, this is the first work dealing with 3D NoCs implemented in a 3D VLSI model.

Keywords: 3D VLSI, 3D NoCs, Performance evaluation, Power consumption.

1 Introduction

Nowadays there are many challenges in designing a complicated integrated circuit. System-on-chip (SoC) is a novel idea that has been proposed to decrease such complexity in designing an IC. This kind of approach has some limitations. One of the major problems associated with future SoC designs arises from non-scalable global wire delays [6]. These limitations have been mentioned in many researches [1], [2], [6], [7], such as limitation in the number of IP cores that can be connected to the shared bus, arbitration for accessing the shared bus, reliability, and so on. To overcome these limitations, network-on-chip (NoC) is introduced in recent years and much research has been conducted in this field of study. The classification of these researches has been discussed in [5]. Another new technology that has been proposed is three dimensional VLSI that exploits the vertical dimension to alleviate the interconnect related problems and to facilitate heterogeneous integration of technologies to realize a SoC design [10]. In this paper we claim that by combining the ideas in these two types of technology, a new kind of architecture for NoC is imaginable and general characteristics of this new architecture under different circumstances have been investigated. In [18], new insights on network topology design for 3D NoCs is provided and issues related to processor placement across 3D layers and data management in L2 caches is addressed. In three dimensional designs with the aid of small

© Springer-Verlag Berlin Heidelberg 2007

links in length between adjacent layers, there is an improvement in performance of the network. The performance and also the power consumption of the circuit are related to the traffic pattern, routing algorithm, switching method, and so on. In this paper, an analysis of the performance and power consumption of three dimensional mesh topology under different circumstances is presented. Two outstanding features of this kind of topology are fast vertical communication and reduction in chip area size.

This paper is organized as follows. The next two sections present the background in NoC, 3D VLSI, and performance metrics in NoC. Section 4 explains the simulation environment that the results have obtained in. The simulation results and experimental evaluation of our approach are presented in section 5. We conclude the paper in section 6.

2 The Basics of Network-on-Chip

Figure 1 shows the basic components of a typical mesh-connected NoC. The ideas and analysis in network-on-chips are very similar to those in interconnect networks in computer networks.

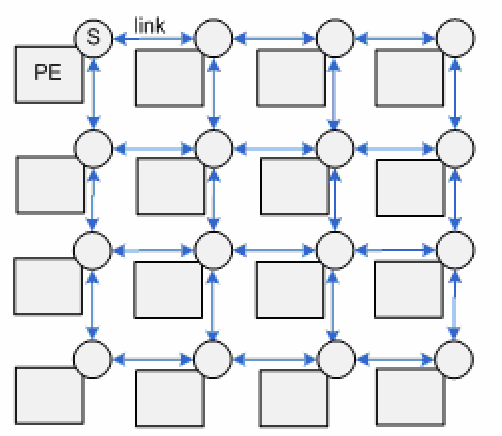


Fig. 1. A 4x4 Mesh Topology

There are four basic components in every NoC that should be considered: 1) Processing elements that are the IP cores connected by the network; 2) Routers and switches that route the packets till received at destination; 3) Network adapters that are the interface between PEs and switches; 4) Links that connect two adjacent switches. In analysis of the network, the behavior of these components should be considered carefully.

One of the benefits of network architecture is that layers in OSI model can be applied in design and analysis. In each layer of the model, important research topics have been discussed briefly in [5]. For instance, in [8], an architecture-level methodology for modeling, analysis, and design of NoC is presented and also tested through two NoC designs.

A different set of constraints exists when adapting these architectures to the SoC design paradigm. High throughput and low latency are the desirable characteristics of a multiprocessor system. Instead of aiming strictly for speed, designers increasingly need to consider energy consumption constraints, especially in the SoC domain [6]. So, in order to compare different NoC architectures, there are some important metrics that should be considered such as latency, energy consumption, and throughput [6]. In this paper, these metrics are investigated in the 3D mesh architecture. This comparison can be obtained in different abstraction levels of simulation environment. In [25], a VHDL based cycle accurate register transfer level model for evaluating the latency, throughput, dynamic and leakage power consumption of NoCs is presented. This evaluation can be analytic; for each architecture in [9], analytical expressions for area, power dissipation, and operating frequency as well as asymptotic limits of these functions are derived. The analysis quantifies the intuitive NoC scalability advantages. Modeling in software is an easy and fast way to evaluate and compare different architectures. We use this method in our work. Power is estimated by a model proposed in [19], called Orion, and latency can be evaluated in the main simulator for interconnection networks that has been developed on the basis of that reported in [22].

3 The 3D VLSI Technology

There are various vertical interconnect technologies that have been explored, including wire bonded, microbump, contactless (capacitive or inductive), and through-via vertical interconnect [17]. Presently, there are several possible fabrication technologies that can be used to realize multiple layers of active-area (single crystal Si or recrystallized poly-Si) separated by interlayer dielectrics (ILDs) for 3D circuit processing [10]. A brief description of these alternatives is given in [10].

Generally, there are some main advantages using the third dimension in VLSI design and these advantages can be very useful in NoC architectures. The benefits of 3D ICs include: 1) higher packing density due to the addition of a third dimension to the conventional two-dimensional layout, 2) higher performance due to reduced average interconnect length, and 3) lower interconnect power consumption due to the reduction in total wiring length [18]. Furthermore, the 3D chip design technology can be exploited to build SoCs by placing circuits with different voltage and performance requirements in different layers [10]. The first benefit is true for conventional circuits and also for NoC architectures. For example, if 64 IP cores in a NoC architecture are organized in a 3D network instead of 2D organization, the chip area reduces almost four times and we will have more integration in design. In conventional integrated circuits, the length of global wires is very important in latency and power consumption especially in emerging deep sub-micron technologies. In NoC architectures, even though wires are invariable in size, links between vertical layers can be very short in comparison with the links in each layer in second dimension. The shorter the links are, the less power they consume. The last benefit that was mentioned is very amazing in SoC designs that is applicable in NoC architectures. The digital and analog components in the mixed-signal systems can be placed on different Si layers thereby achieving better noise-performance due to lower electromagnetic interference between such circuit blocks [10].

There are currently various 3D technologies being explored in industry and academia, but the two most promising ones are Wafer-Bonding [14], [18] and Multi-Layer Buried Structures (MLBS) [17], [18]. The details of these processes are discussed in [10]. There are currently two primary wafer orientation schemes, Face-To-Face and Face-To-Back, as mentioned in [16]. While the former provides the greatest layer-to-layer via density, it is suitable for two-layer organizations, since additional layers would have to employ back-to-back placement using larger and longer vias. Face-To-Back, on the other hand, provides uniform scalability to an arbitrary number of layers, despite a reduced inter-layer via density [18]. As mentioned in [17], [18], wafer bonding requires fewer changes in the manufacturing process and is more popular in industry than MLBS technology. Therefore, the integration approach we adopt in this study is the wafer-bonding technology [17].

Previously in many studies the performance of three dimensional designs has been investigated and examined [10], [11], [14], [15], [16]. Most of the performance evaluations are based on wire-length distributions. It means that a stochastic 3D interconnect model is presented and the impact of 3D integration on circuit performance and power consumption is investigated. In this study, our attention is focused on a network architecture point of view of three dimensional technologies.

4 Simulation Environment

Modeling of NoCs is to achieve two main goals: 1) Exploration of design space, and 2) Evaluation of trade-offs between different parameters like power, latency, design time and so on [5]. In many studies, different parameters of NoCs are modeled and investigated [4], [6], [8], [23], [24]. For instance, in [8], a hierarchal modeling for on-chip communication is presented. The model consists of two main parts: 1) On-chip communication architecture (OCA), and 2) Processing elements.

The simulator, in our work, consists of two main parts: 1) A power model for evaluating power consumption in components of the network in each layer (called Orion), 2) An interconnection network simulator that is developed based on POPNET simulator presented in [22]. Orion is a power-performance interconnection network simulator that is capable of providing detailed power characteristics, in addition to performance characteristics, to enable rapid power performance tradeoffs at the architecture level [19].

The model computes the power consumption in different components of the network according to the events that occur during simulation. So, according to the model, the network is decomposed into basic components and power models are applied for each component. As mentioned in [19], the total energy each flit consumes at a specified node and its outgoing link is given by:

$$E_{flit} = E_{wrt} + E_{arb} + E_{read} + E_{xb} + E_{link} \tag{1}$$

It consists of five components: 1) the power that is consumed in writing into buffers; 2) The power of arbiter; 3) The power that is consumed in reading from buffers;

4) The power of the internal crossbar; 5) The power that is consumed in the links. The general switch model is illustrated in figure [19].

In [20], the models for each component are discussed in detail. In [21], an architectural leakage power modeling methodology is also proposed.

The POPNET simulator that is based on Orion is presented in [22] only for a two dimensional mesh topology. We have customized the simulator to support other topologies and other routing algorithms. So, the simulator takes the configuration parameters and will do the simulation. The power is obtained and reported for each layer and each component of the network.

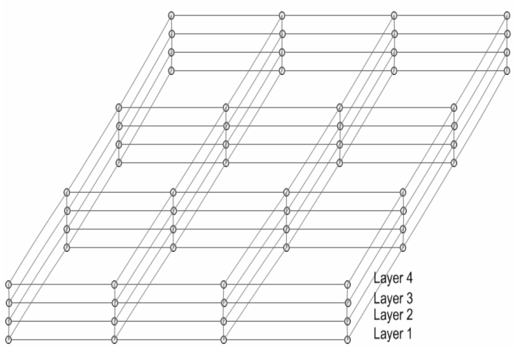


Fig. 2. A 3D mesh structure (4x4x4 mesh)

5 Simulation Results

We applied our simulation to a three dimensional mesh structure with four active layers as shown in figure 2. We have examined the effect of different parameters (like: different switching methods, routing algorithms, length of messages, number of virtual channels, traffic patterns) on the performance and power consumption (on each layer) of the network. The length of each flit has been assumed 32 bits and the length of each message will be a factor of the number of flits. In most of the simulations, the length of messages is assumed to be 32 flits and the size of buffers is 2 flits except for virtual-cut-through switching which uses a full message size buffer of 32 flits. Flits represent logical units of information, which correspond to physical quantities, that is, the number of bits that can be transferred in parallel in a single cycle [3]. Injection rate is the probability that a node sends a packet during a cycle, and the traffic according to the injection rate is generated on the basis of Poisson distribution.

It is seen from figure 3 that, if the length of messages is doubled, the traffic of the network will be almost doubled too. So at a same injection rate before network saturation the average latency of the network with longer messages dominates the one with shorter messages. Also the network will be saturated at a lower injection rate. At the saturation point the latency will increase exponentially, due to the limitations in network resources. The power that dissipates in the network also depends on the messages' length to a great extend. If the message length increases, consequently at a

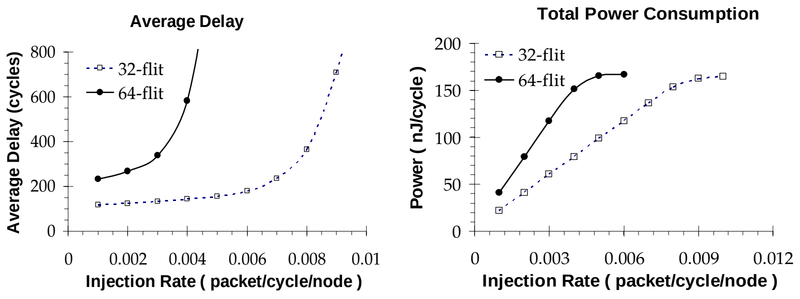


Fig. 3. Total power consumption and performance comparison for two different message lengths (32-flit and 64-flit). Different parameters of simulation and network are: (switching method: wormhole, routing algorithm: dimension-order routing, length of input buffer: 2 flits, number of virtual channels per physical channel: 2, traffic pattern: uniform).

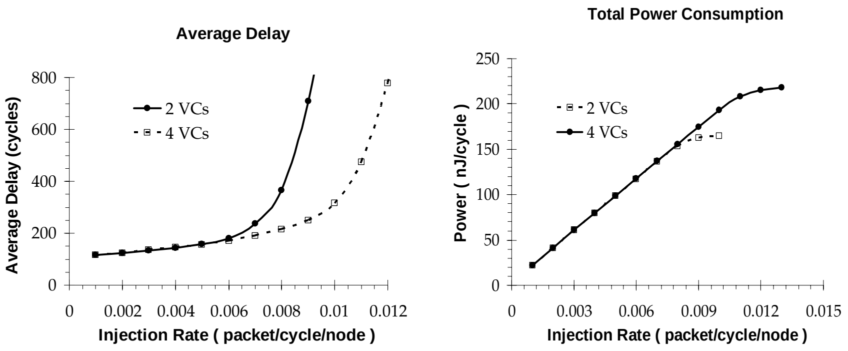


Fig. 4. Total power consumption and performance comparison for two different numbers of virtual channels per physical channel (2 and 4). Other parameters are the same as figure 3.

same injection rate there are more flits in the network communicated and so they will dissipates more power. The power dissipation will increase linearly according to the injection rate. Although the network with longer messages dissipates more power, near the saturation point the power consumption will be the same and remains constant for the two networks. This stability also can be seen in all following results in this paper. Beyond saturation, no additional messages can be injected successfully into the system and, consequently, no additional energy is dissipated. The value of power dissipation is the same for two conditions beyond saturation point because at the saturation point the network is overloaded with flits of data and there is no difference between the network with 32-flit messages and 64-flit messages.

Figure 4 shows the impact of the number of virtual channels. The main goal of adding virtual channels in interconnection networks is to improve performance. As shown in figure 4, the network with 4 virtual channels can handle the traffic with larger injection rate in comparison with the network with 2 virtual channels per physical channel. The expense that we pay is more power that dissipates in the virtual channels. At the lower injection rates, there is no magnificent difference between networks with 2 and 4 virtual channels, but when the network approaches saturation point the impact of number of virtual channels is visible. With more virtual channels, there are more memory elements in the network for flits and so the accepted traffic will increase. The network with 4 virtual channels will approach saturation point approximately 30% later than the network with 2 virtual channels but it consumes about 30% more power than the other one.

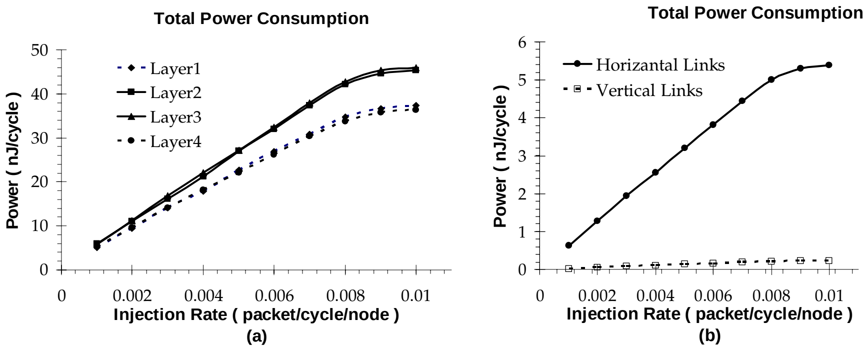


Fig. 5. (a) Total power dissipation in each of four active layers. (b) Total power dissipation in vertical and horizontal links. Different parameters of simulation and network are: (switching method: wormhole, routing algorithm: dimension-order routing, length of input buffer: 2 flits, number of virtual channels per physical channel: 2, traffic pattern: uniform, message length: 32 flits).

In Figure 5.a, we show the power dissipation in each layer of the network in the case of uniform traffic. As mentioned before, the power remains constant beyond the saturation point. The power that dissipated in layer 1 and 4 is approximately equal. Also, the power in layers 2 and 3 is the same. Layers 1 and 4 are symmetric in the network architecture and also are layers 2 and 3. So, in the uniform traffic, they will handle the same amount of messages in the fixed period of time. Layers 2 and 3 consumes more power than layers 1 and 4 because layers 2 and 3 are in the middle of the structure and they take the messages from up and down layers, but layers 1 and 4 are adjacent only to one layer and so they should handle small amount of messages and hence they will consume less power (as shown in figure 5.a).

In three dimensional structures, the vertical links according to their shortness consumes much less power than horizontal links as shown in figure 5.b. In deep submicron technologies the power dissipation in links becomes dominant factor and so the shorter links with less capacitance and resistance are favorable.

Figure 6 shows the power that dissipates in each layer in the case of hotspot traffic. The hotspot is located in layer 2 with a hit probability of 0.16 (it means that other nodes will send the packets to the hotspot node with the probability of 0.16). The location of hotspot is very important. In figure 5.a, the power that dissipates in layer 2 is similar to the power consumption of layer 3, but as it is shown in figure 6, the power that dissipates in layer 2 is larger than any other layers. The reason is that the dimension-order routing algorithm is in the order of z-y-x. So, the traffic in layer 2 will increase because the packets that destined for the hotspot at first will move through z dimension. This behavior is important in three dimensional designs; because if the hotspot is located in lower layers the heat transfer becomes difficult because the heat should move through upper layers to reach the sink. With the presence of hotspots in the network, the network will approach the saturation point more quickly.

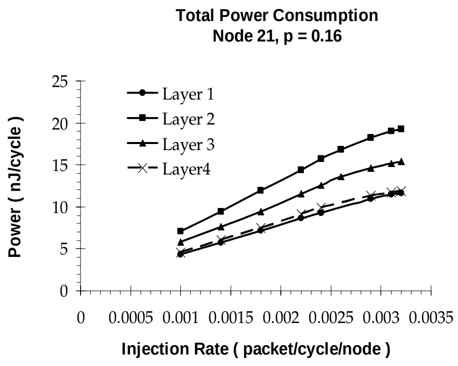


Fig. 6. The power dissipation in each layer in the case of hot-spot traffic

If the probability of sending messages of other nodes to the hotspot increases, the network will saturates more quickly as shown in figure 7.a. Figure 7.a shows the average delay of network in the case of hotspot that is located in layer 1 with two probabilities 0.2 and 0.1.

In the previous simulations, the routing algorithm that has been used is dimension-order routing algorithm (in the order of z-y-x). Dimension order routing is one of the most popular deterministic routing algorithms. Here, a message is routed along

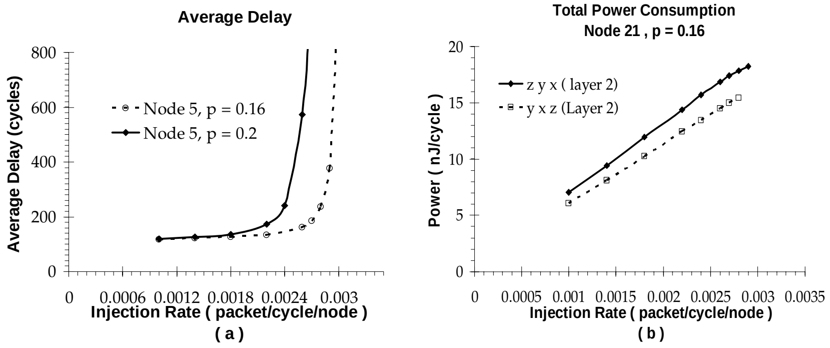


Fig. 7. (a) Average delay of the network in the case of hot spot traffic with two different probabilities p=0.1 and p=0.2. (b) The comparison of two approaches in dimension order routing algorithm in the presence of hotspot in layer 2. Power dissipation of the layer 2 that the hot spot is located in is shown.

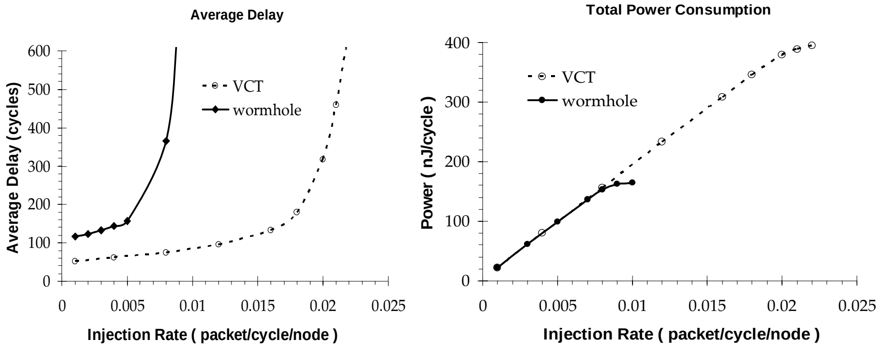


Fig. 8. Average delay and power dissipation in the case of wormhole and virtual-cut-through switching

decreasing dimensions with a decrease occurring only when zero hops remain in all the higher dimensions. In the case of hotspot located in layer2, the order of dimensions in the algorithm has been changed (in the order of y-x-z). As it was mentioned, the power that dissipates in the layer that hotspot is placed is higher than other layers. The power dissipation in the hotspot layer of the network in two cases is shown in figure 7.b. The power dissipation in layer 2 is reduced because the traffic in the layer has been decreased.

In virtual-cut-through switching method the size of buffers in each node is equal to the size of packets. So every node can save a complete packet in the case of blocking. By using much more buffers in each node, the network can handle a larger traffic load that will result in lately saturation. As the network can tolerate much more messages, the power that dissipates in the components of the network continues to increase (as shown in figure 8).

6 Conclusion

In this paper we have claimed that by combining the ideas in 3D VLSI and NOC technologies, a new kind of architecture for NoC is imaginable and the main goal of this paper is to examine general characteristics and parameters of a three dimensional network-on-chip architecture. It is possible to evaluate other topologies in three-dimension and compare the performance and power behavior of the network with two dimensional implementations in our simulation platform. Fast communications in vertical links can improve performance and power dissipation to a great extent. Of course, the power dissipation in each layer especially in lower layers is very important and should be considered carefully. Modeling the heat transfer in 3D designs also can be mentioned based on power dissipation in each active layer. Still there are many topologies and parameters in three dimensional network-on-chips that should be explored.

References

1. Jantsch, A., Tenhunen, H.: Network on Chip. Kluwer Academic Publishers, Dordrecht (2003)
2. Wiklund, D.: An On-Chip Network Architecture for Hard Real Time Systems. MSc thesis, Linköpings University (2003)
3. Duato, J., Yalamanchili, S., Ni, L.M.: Interconnection Networks. Morgan Kaufman, San Francisco (2003)
4. Nielsen, K.H.: Evaluation of Real-time Performance Models in Wormhole-routed On-chip Networks. MSc thesis, Stockholm (2005)
5. Bjerregaard, T., Mahadevan, S.: A Survey of Research and Practices of Network-on-Chip. ACM Computing Surveys 38(1), 1–51 (2006)
6. Pande, P.P., Grecu, C., Jones, M., Ivanov, R., Saleh, R.: Performance Evaluation and Design Trade-Offs for Network-on-Chip Interconnection Architectures. IEEE Transactions on Computers 54(8) 1025-1040 (2005)
7. Grecu, C., Pande, P.P., Ivanov, R., Saleh, R.: Timing Analysis of Network on Chip Architectures for MP-Soc Platforms. Microelectronics 36, 833–845 (2005)
8. Xu, J., Wolf, W., Henkel, J., Chakradhar, S.: A Methodology for Design, Modeling, Analysis of Networks-on-Chip. In: ISCAS, pp. 1778–1781 (2005)
9. Bolotin, E., Cidon, I., Ginosar, R., Kolodfdny, A.: Cost Considerations in Network on Chip. Integration, the VLSI journal 38, 38–42 (2005)
10. Banerjee, K., Souri, S.J., Kapur, P., Saraswat, K.C.: 3D ICs: A Novel Chip Design for Improving Deep-Submicrometer Interconnect Performance and Systems-on-Chip Integration. Proceedings of the IEEE 89(5), 602–633 (2001)
11. Zhang, R., Roy, K., Koh, C.K., Janes, D.B.: Stochastic Wire-Length and Delay Distributions of 3-Dimensional Circuits. In: ICCAD, pp. 208–213 (2000)
12. Cong, J., Jagannathan, A., Ma, Y., Reinman, G., Wei, J., Zhang, Y.: An Automated Design Flow for 3D Microarchitecture Evaluation. In: ASPDAC, pp. 384–389 (2006)
13. Puttaswamy, K., Loh, G.H.: Implementing Caches in a 3-D technology for High Performance Processors. In: ICCD, pp. 525–532 (2005)
14. Das, S., Chandrakasan, A., Reif, R.: Three Dimensional Integrated Circuits: Performance, Design, Methodology and CAD tools. In: ISVLSI, pp. 13–18 (2003)
15. Souri, S.J., Banerjee, K., Mehrotra, A., Saraswat, K.C.: Multiple Si Layer ICs: Motivation, Performance Analysis and Design Implications. In: DAC, pp. 213–220 (2000)
16. Zhang, R., Roy, K., Koh, C.K., Janes, D.B.: Power Trends and Performance Characterization of 3-Dimensional Integration for Future Technology Generation. In: ISQED, pp. 217–222 (2001)
17. Xie, Y., Loh, G.H., Black, B., Bernstein, K.: Design Space Exploration for 3D Architectures. ACM Journal on Emerging Technologies in Computing Systems 2, 65–103 (2006)
18. Li, F., Nicopoulis, C., Richardson, T., Xie, Y., Krishnan, V., Kandemir, M.: Design and Management of 3D Chip Multiprocessors Using Network-in-Memory. In: ISCA, pp. 130–141 (2006)
19. Wang, H.S., Zhu, X., Peh, L.S., Malik, S.: Orion: A Power-Performance Simulator for Interconnection Networks. In: MICRO, pp. 294–305 (2002)
20. Wang, H.: A Detailed Architectural Level Power Model for Router Buffers, Crossbars and Arbiters. Technical Report, Princeton University (2003)
21. Chen, X., Peh, L.S.: Leakage Power Modeling and Optimization in Interconnection Networks. In: ISLPED, pp. 90–95 (2003)
22. Shang, L.: PoPNet Simulator (2007), http://www.ee.princeton.edu/~lshang/popnet.html

23. Kreutz, M., Marcon, C., Carro, L., Calazans, N., Susin, A.A.: Energy and Latency Evaluation of NOC Topologies. In: ISCAS, pp. 5866–5869 (2005)
24. Wolkotte, P.T., Smit, G.J.M., Kavaldjiev, N., Becker, J.E.: Energy Model of Networks-on-Chip and a Bus. In: SoC, pp. 82–85 (2005)
25. Banerjee, N., Vellanki, P., Chatha, K.S.: A Power and Performance Model for Network-on-Chip Architectures. In: DATE, pp. 1250–1255 (2004)

A Design Methodology for Performance-Resource Optimization of a Generalized 2D Convolution Architecture with Quadrant Symmetric Kernels

Ming Z. Zhang and Vijayan K. Asari

Computational Intelligence and Machine Vision Laboratory
Department of Electrical and Computer Engineering
Old Dominion University, Norfolk, VA 23529, USA
{mzhan002,vasari}@odu.edu

Abstract. We present a design technique to meet application driven constraints for performance-resource optimization of a generalized 2D convolution with quadrant symmetric kernels. A fully pipelined multiplierless digital architecture for computing modularized 2D convolution utilizing the quadrant symmetry of the kernels is proposed in this paper. Pixels in the four quadrants of the kernel region with respect to an image pixel are considered simultaneously with distributed queues for computing the partial results of the convolution sum in time-sliced fashion. The new architecture performs computations in log-domain by utilizing low complexity high performance $\log_2$ and inverse-$\log_2$ estimation modules. An effective data handling strategy is developed to minimize routing of data path in conjunction with the logarithmic modules to eliminate the necessity of the multipliers in the architecture. The proposed architecture is capable of performing convolution operations for 45.5 (1024×1024) frames or 47.73 million outputs per second in minimum resource configuration with 8×8 kernels in a Xilinx's Virtex XC2V2000-4ff896 field programmable gate array (FPGA) at maximum clock frequency of 190.92 MHz. Analysis shows that the performance and resource utilization between the fully parallel and fully resource constrained architectures are proportional to f and 1/f, respectively where f is the application driven reusability of the main computing components. In addition to resource reduction from optimization scheme, evaluation in Xilinx's core generator also showed that the proposed design results in 60% reduction in hardware when compared to the design using pipelined multipliers.

Keywords: 2D convolution, log-domain computation, multiplier-less architecture, quadrant symmetric kernels, modularized optimization, FPGA based architecture.

1 Introduction

Two-dimensional convolution is one of the most frequently used operations in image and video processing applications. Kernel size is usually limited to a small

© Springer-Verlag Berlin Heidelberg 2007

bounded range when general processors are used. It is therefore necessary to find optimal designs to reduce hardware resources and power consumption while supporting high speed operations for real-time applications. Convolution is a mathematical operator which takes two functions, f and g, and produces a third function, z, that in a sense represents the amount of overlap between f and a reversed and translated version of g. The definition of 2D convolution $O = W * I$ in general can be expressed as

$$O(m,n) = \sum_{j_1=-a_1}^{a_1} \sum_{j_2=-a_2}^{a_2} W(j_1,j_2) \times I(m-j_1, n-j_2) \tag{1}$$

where a_i =(J_i-1)/2 for i = 1, 2 and W is the kernel function. The complexity for 2D convolution (filter) operation in image processing applications is of the order O($M \times N \times J_1 \times J_2$) where $M \times N$ corresponds to the x-y dimension of I/O images and $J_1 \times J_2$ is the size of the kernel. For instance, in a video processing application, if the frame size is 1024× 1024 and the kernel size is 10×10, more than 3 billion operations per second are required to support real-time processing rate of 30 frames per second. Many researches have studied and presented various methods to implement the hardware architectures to perform convolution operation for real time applications in the last two decades as described in [1] and [2]. One of the most difficult challenges in designing the high speed convolution architecture with a large kernel is to effectively utilize the hardware resources and limited number of processing elements (PEs) to support real time processing [3][4][5][6][7][8]. Algorithms such as [1][2][9][10][11][12] optimize the hardware resources by generating the kernel coefficients dependent architectures. Such architectures may be too specific and require the supports of reconfigurability and external system to generate reconfiguration bit streams for uploading to FPGAs in the event of changing the coefficients. Hence they are more suitable for static kernel coefficients. In general, 2D convolution operations can be partitioned to a number of 1D convolutions [13] without specializing for separable kernels [14]. Specifically, the partitioned 1D kernels are first convolved simultaneously on all columns of the input image and the partial results are accumulated in a row-ordered fashion or vise versa. We presented log-domain computation to significantly lower the complexity of the 2D convolution operation with quadrant symmetric architecture with support of programmable kernel coefficients to suit different transfer functions. In this paper, we propose a modularized 2D convolution and utilize distributed queue architecture for performance-resource optimization where excessive performance can be utilized to minimize resource requirement.

2 Concept of the Modularized 2D Convolution with Quadrant Symmetric Property

From the definition of 2D convolution in (1), it can be partitioned to reflect its symmetry. With slight modification to (1), equation (2) defines the convolution operation where the center pixel of the kernel is overlapped with center pixel of the image under the window of consideration. The rearrangement of the equation is done by reducing the summation to half and by adding the terms to be multiplied by the same kernel coefficients. For odd kernel size in equation (3), corresponding locations in all four quadrants are added before multiplication. The last term is outside the summation since the kernel coefficients on the axes may be different from those reflected about the axes. For even kernel size, the folding method is shown in equation (4). This folding scheme can be achieved within the design with respect to particular architectural style and reduce processing power to focus on a quarter of the kernel.

$$O(m,n) = \sum_{j_1=0}^{J_1} \sum_{j_2=0}^{J_2} W(j_1,j_2) \cdot I\left(m + j_1 - \frac{J_1}{2} + 1, n + j_2 - \frac{J_2}{2} + 1\right) \tag{2}$$

$$\begin{aligned} O(m,n) = &\sum_{j_1=0}^{\frac{J_1}{2}-1} \sum_{j_2=0}^{\frac{J_2}{2}-1} W(j_1,j_2) \cdot I\left(m \pm j_1 + \tfrac{J_1}{2}, n \pm j_2 + \tfrac{J_2}{2}\right) \\ &+ W\left(\tfrac{J_1}{2}, \tfrac{J_2}{2}\right) \cdot I(m,n) \end{aligned} \tag{3}$$

$$\begin{aligned} O(m,n) = &\sum_{j_1=0}^{\frac{J_1-1}{2}} \sum_{j_2=0}^{\frac{J_2-1}{2}} W(j_1,j_2) \times \\ &\begin{bmatrix} I\left(m + j_1 - \frac{J_1}{2} + 1, n + j_2 - \frac{J_2}{2} + 1\right) \\ +I\left(m - j_1 + \frac{J_1}{2}, n + j_2 - \frac{J_2}{2} + 1\right) \\ +I\left(m + j_1 - \frac{J_1}{2} + 1, n - j_2 + \frac{J_2}{2}\right) \\ +I\left(m - j_1 + \frac{J_1}{2}, n - j_2 + \frac{J_2}{2}\right) \end{bmatrix} \end{aligned} \tag{4}$$

The $\log_2$ and inverse-$\log_2$ modules presented in [15] can further be employed to replace the essential multiplications in (3) and (4) with additions. The logical logarithmic operators in (5) require very low hardware complexity. We generalized the operators to handle input with fractions and derived very compact implementation without compromising hardware resource and performance compared to conventional (unrolled pipelining) architectures which operate on integers only [16], [17]. Eq. (5) states that the $\log_2$ scale of V can be calculated by concatenating the index I_V of leading 1's in V with the fractions (remaining bits after I_V^{th} bit). The reversed process holds true as well, except the leading 1's and fractions, L_f, are shifted to the left by L_i (integer of L) bits as shown in (5).

$$\log_2(V) \cong \{I_V\} + \{(V - I_V) \gg I_V\} \Leftrightarrow \log_2^{-1}(L) \cong \{1 \ll L_i\} + \{L_f \ll L_i\} \tag{5}$$

For even kernels, (4) can be modularized to balance the performance and hardware resource based on the constraint imposed by the application. For example of a high-end FPGA design (usually has greater throughput rate than needed by the application) deployed to low-end system, the excessive bandwidth can be distributed for reuse to minimize the resource. Vise versa, a low-end design can sustain very high throughput by increasing internal parallelism with more resource. The partitioning scheme can be defined by (6), where $I_4(.)$ is the folded pixel value, and i is the i^{th} partition for i in $0..\lceil J_2/(2f)\rceil - 1$, and f is the desired reusability driven by application constraint. The design of modularized architecture is discussed in section 3.

$$\begin{aligned} O(m,n) &= \sum_{j_1=0}^{\frac{J_1-1}{2}} \sum_{j_2=0}^{\frac{J_2-1}{2}} W(j_1,j_2)\cdot I_4(.) \\ &= \sum_{i=0}^{\lceil J_2/(2f)\rceil -1} \left\{ \sum_{j_2=i\times f}^{(i+1)\times f-1} \sum_{j_1=0}^{\frac{J_1-1}{2}} W(j_1,j_2)\cdot I_4(.) \right\},\ j_2 \subseteq 0...\tfrac{J_2-1}{2} \end{aligned} \tag{6}$$

3 Architecture of Modularized 2D Convolution

3.1 Dataflow of Modularized Architecture

In this section, even kernel size is assumed for the discussion. The design also applies to odd kernel size with minor difference. It is also assumed the data comes in stream form (i.e. pixel by pixel fetched in real-time). Equation (4) is translated into dataflow of the convolution architecture as illustrated in Fig. 1. Incoming pixels pass through a series of line buffers (LBs) and fold at vertical position. This folding corresponds to the equation (4) without modification. The results from vertical folding are sent to a set of delay registers. The horizontal folding takes account of the inherent delays of systolic architecture rather than direct translation of (4). This is compensated by rerouting the folding points accordingly as shown in Fig. 1. The processing bandwidth is shared by f horizontally folded lines as reflected by the partition for each i in (6). The partial results from horizontal folding are successively accumulated to the right side and merged to form complete output.

3.2 Overview of Modularized 2D Convolution Architecture

Fig. 2 shows the block diagram of the quadrant symmetric convolution architecture with distributed storage queues. The LBs consist of J_2-1 line delays and create massive internal parallelism for concurrent processing. The folding is determined at nodes $LB(i)+LB(J_2-i)$ where i ranges from 0 to (J_2-1)/2. The resulting nodes are registered and fed into multiplexers (MUXes) for time-sliced processing where each selected input occupies computational modules once every f cycles. The horizontal folding, as illustrated in Fig. 1, is done accordingly at the nodes $MQ(0)+DQ(2k+1)$ (where MQ is the registered output of MUX and DQ is the distributed storage queue for vertically folded data) rather than straight from the equations due to the pipelining involved in horizontal folding. The registered results from horizontal folding are sent to shared PE arrays (PEAs) for multiplication with the kernel coefficients and successive accumulation. The multipliers are reduced by three quarters by simply performing the folding procedure and by another f factor for sharing of processing bandwidth.

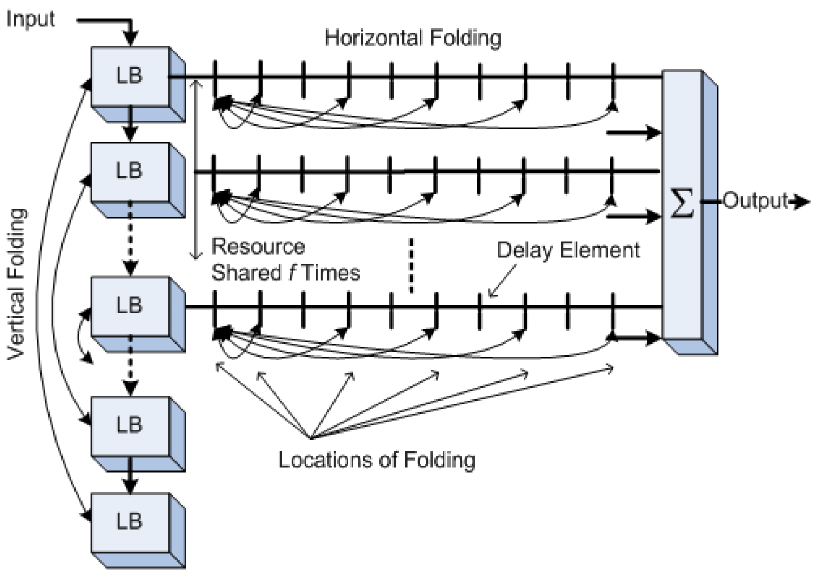


Fig. 1. Dataflow version of folding illustrates the compensated delays induced in the systolic architecture. The nodes involved with vertical folding at symmetric locations are added together. The horizontal folding takes place in every other nodes.

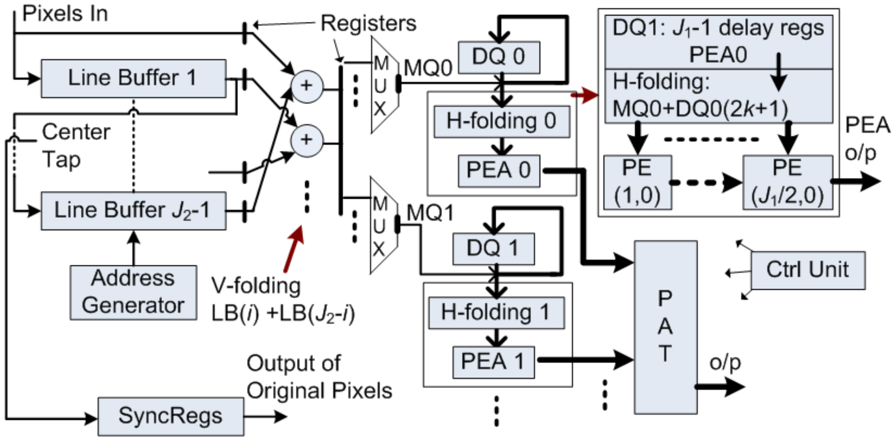


Fig. 2. Block diagram for overall architecture of the modularized 2D convolution with quadrant symmetric property in the kernel

The overall output is computed by merging results of the shared PEAs through pipelined adder tree (PAT). The original pixel values can be obtained, along with convolved data, through synchronized registers (SyncRegs).

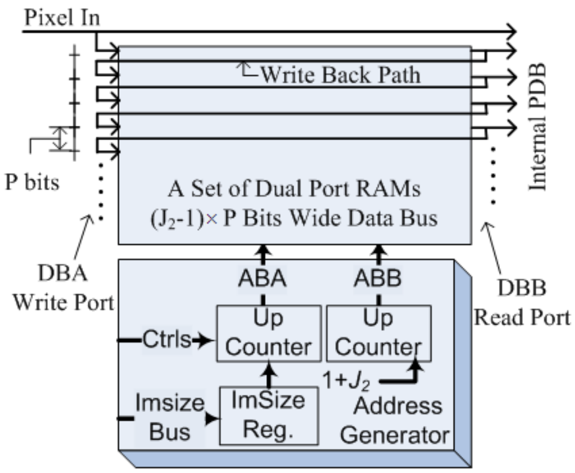


Fig. 3. Design of Data Buffer Unit with DPRAMs

3.3 Data Buffer Unit

The data buffer unit (DBU), which generates the internal parallelism is implemented with the dual port RAMs (DPRAMs) as shown in Fig. 3. One set of DPRAMs is utilized to form LBs and store just enough lines of image to create massive internal parallelism for concurrent processing. The pixels are fetched to the data buffer in raster-scan fashion which requires unity bandwidth suitable for real time streaming applications in video processing. The DPRAM based implementation has advantage of significantly simplifying the address generator compared to commonly known first-in-first-out (FIFO) based approach. Tracking of items during transient stage is eliminated as opposed to LBs implemented by FIFOs. The address generator with the DPRAMs based implementation makes scalability of DBU consistent and simple. It consists of two counters to automatically keep track of the memory locations to insert and read the data to internal parallel data bus (PDB). Data bus A (DBA) of (J_2-1) $\times P$ bits wide, which is formed with just enough number of DPRAMs in parallel, is used to insert pixel values through write-back paths to the memory location designated by address bus A (ABA). P is 8 bits for 8-bit pixel resolution. The data bus B (DBB) is used to read the pixel values onto PDB and write to the write-back paths. Only one address generator is necessary in buffering scheme. The DBU is only active once every f cycles. The vertical folding and modularizing scheme is discussed next.

3.4 Modularized Processing

The vertical folding is accomplished by pre-adding the nodes, $LB(i)+LB(J_2-i)$, on the PDB where i ranges from 0 to (J_2-1)/2 for even size kernels. The aligned results from vertical folding form a group of V-folded bus (VB) with the width of $(P+1)\times f$ bits. Each VB is connected to an array of $P+1$ MUXes of f-to-1 type in such a way that the binary select lines of the MUXes incrementally activate appropriate data from PDB for time-sliced processing. Hence, each active ordered bus of $P+1$ bits wide will periodically occupy the PEA on one of those f time slots. The routing scheme shown in Fig. 2 and Fig. 4 (left) from VB to MQ can be generalized as

$$VB_i\,\{v\} \Rightarrow MQ_i \begin{Bmatrix} v, & v=(P+1)f-1 \\ \{v\times(P+1)\}\,\%\,\{(P+1)f-1\}, & otherwise \end{Bmatrix} \tag{7}$$

for all i in (6), v= 0..$(P+1)\times f-1$, where v is the index to the VB which is mapped to the inputs of MUX array, and % denotes the modulo operation. For all select lines of MUXes, only one common counter suffices the time slicing necessary according to the partitioned convolution equation in (6).

3.5 Queuing of Active VB Data

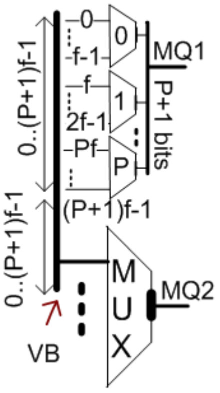
The time-sliced data on MQ_i output is combined with the circular queue DQ_i, implemented with f registers of $(J_1-1)\times(P+1)$bits wide as illustrated in Fig. 4 (right). The merged signals $MDQ_i[0..J_1-1]=[MQ_i, DQ_i[f]]$ are connected to horizontal folding scheme described by

$$\mathrm{HB}_i\,(k)=\begin{cases} \mathrm{MDQ}_i[0]+\mathrm{MDQ}_i[2k], & \text{for even } J_1,\ \forall k \\ \mathrm{MDQ}_i[0]+\mathrm{MDQ}_i[2k+1], & \text{for odd } J_1, \mathrm{k}\neq 0 \\ \mathrm{MDQ}_i[0], & \text{for odd } J_1, \mathrm{k}{=}0 \end{cases} \tag{8}$$

to form horizontally folded bus (HB). The k ranges from 0 to $\lceil J_1/2\rceil-1$ for both even and odd kernels (i.e. J_1 is even number). The last $P+1$ bits of MDQ_i are dropped out with the remaining bits clocked back into the $DQ_i[0]$ register, mimicking the shifting operation of $P+1$ bits shift registers. Hence, the functional equivalence of fully parallel approach in [18] can be achieved in time-sliced fashion without introducing additional hardware components. The architecture of processing elements is discussed in the next section.

3.6 Processing Elements in PEAs

The design of each PE utilizes the log-domain computation to eliminate the need of hardware multipliers [15], [18]. The data from H-fold register is converted to $\log_2$ scale as shown in Fig. 5 (left) and added with the output of $\log_2$ scaled kernel coefficients queue (LKC) implemented with register set which holds the f desired spatial samples of the transfer functions according to the partitioned in (6). The result from last stage is converted back to linear scale with range check (RC). If the overflow or underflow occurs, the holding register of this pipeline stage is set or clear, respectively. Setting and clearing contribute the max and min values representable to $P+2$ bits (resolution of the $HB_i(k)$ data is 10 bits



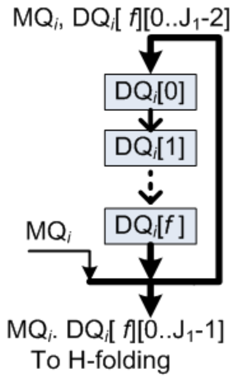
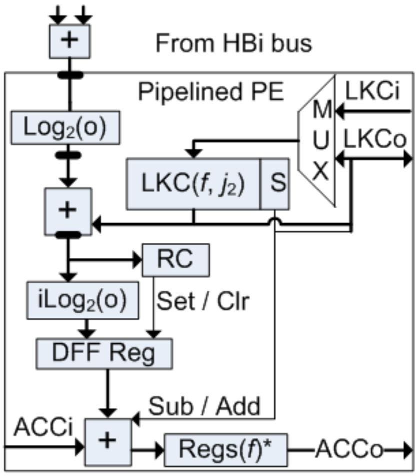

MQ$_i$, DQ$_i$[f][0..J$_1$-2]
DQ$_i$[0]
DQ$_i$[1]
DQ$_i$[f]
MQ$_i$
MQ$_i$. DQ$_i$[f][0..J$_1$-1]
To H-folding

Fig. 4. Left: routing scheme for vertically folded data, Right: feedback path of DQ_i



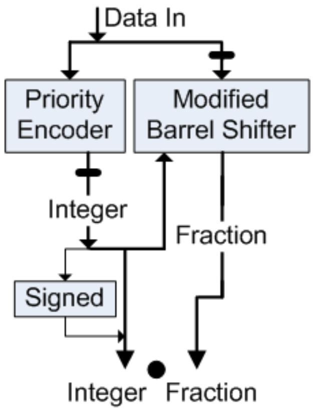


Fig. 5. Left: design of PE, Right: pipelined $\log_2$ module

for 8-bit image) register. The output of this stage is successively accumulated along the accumulation line (ACCi) and queued by f registers at the output (ACCo) to synchronize the computed partial results correctly to the assigned time-slice. 'ACCi' is set to 0's for the leftmost PE in each PEA. For the rightmost PE, the output buffer at 'ACCo' is replaced with cyclic adder for cyclic accumulation of the results produced by the PEAs. Cyclic adder is similar to conventional accumulator used in multiply-accumulate (MAC) unit with periodic reset signal. Cyclic adder sums up the results produced by each f-cycle interval. The output of *LKC* is looped back to its input through 2-to-1 MUX. Thus, the coefficients stored and rotated in *LKC* can be synchronized to appropriate time-slice. The other input on the MUX is connected to the coefficient input bus, 'LKCi' from its neighbor PE. With the coefficient output, 'LKCo', an inter-chained bus can be organized for streamed initialization of kernel coefficients for different convolution masks. The PE can easily be modified to support negative inputs from horizontal folding. This is done by separating out the sign bit and exclusively OR-ed the registered sign bit with the sign from output of *LKC* queue. Hence if either one is negative, a subtraction is carried out alone the accumulation line. Otherwise, addition is performed. The $\log_2$ architecture shown in Fig. 5 (right) is very similar to [15], except full precision is used and registers are introduced to approximately double the performance. The maximum logic delay is reduced to single component and makes no sense to pipeline further.

4 Simulation and Error Analysis

4.1 Simulation

Both low-pass and high-pass transfer functions are applied to convolve with 8-bit grayscale images (336×336) in the simulation. The $\log_2$ and inverse-$\log_2$ modules are optimized based on the resolution of the test images shown in Fig. 6a and 6b. Fig. 6a is derived from Fig. 6b by adding Gaussian noise with$\mu = 0$and$\sigma^2 = 0.02$. Each test image is fetched into the architecture pixel by pixel in raster-scan fashion. After transient state, the outputs become available and are collected for error analysis. The resulting images produced by the Matlab software are shown in Fig. 6c and 6d for noise filtering and edge detection kernels, respectively. The corresponding output images from hardware simulation are shown in Fig. 6e and 6f. Both images are visually identical to Fig. 6c and 6d. Error Analysis is given next to determine quantitatively the degree of difference.

4.2 Error Analysis

Typical histograms of the error between software algorithm and hardware simulation are shown in Fig. 7a and 7b for the test images. The error produced in noise filtering illustrated in Fig. 7a has average error of 2.19 pixel intensity with peak of 22.23. The average error for applying the edge detection kernel in Fig. 7b is 4.17. Simulation with large set of images shows majority of the errors in this design is less than 5 pixel intensities. The peak errors are generally larger for high-pass transfer functions since the kernel coefficients typically have broader range of sample values. Images with evenly distributed histograms also have tendency to increase the peak error while contribute less to average error. The error measure includes the fact that the hardware simulation is bounded to approximation error and specific number of bits representable in the architecture where the software algorithm is free from these constraints. The performance-resource relationship is characterized in section 5.

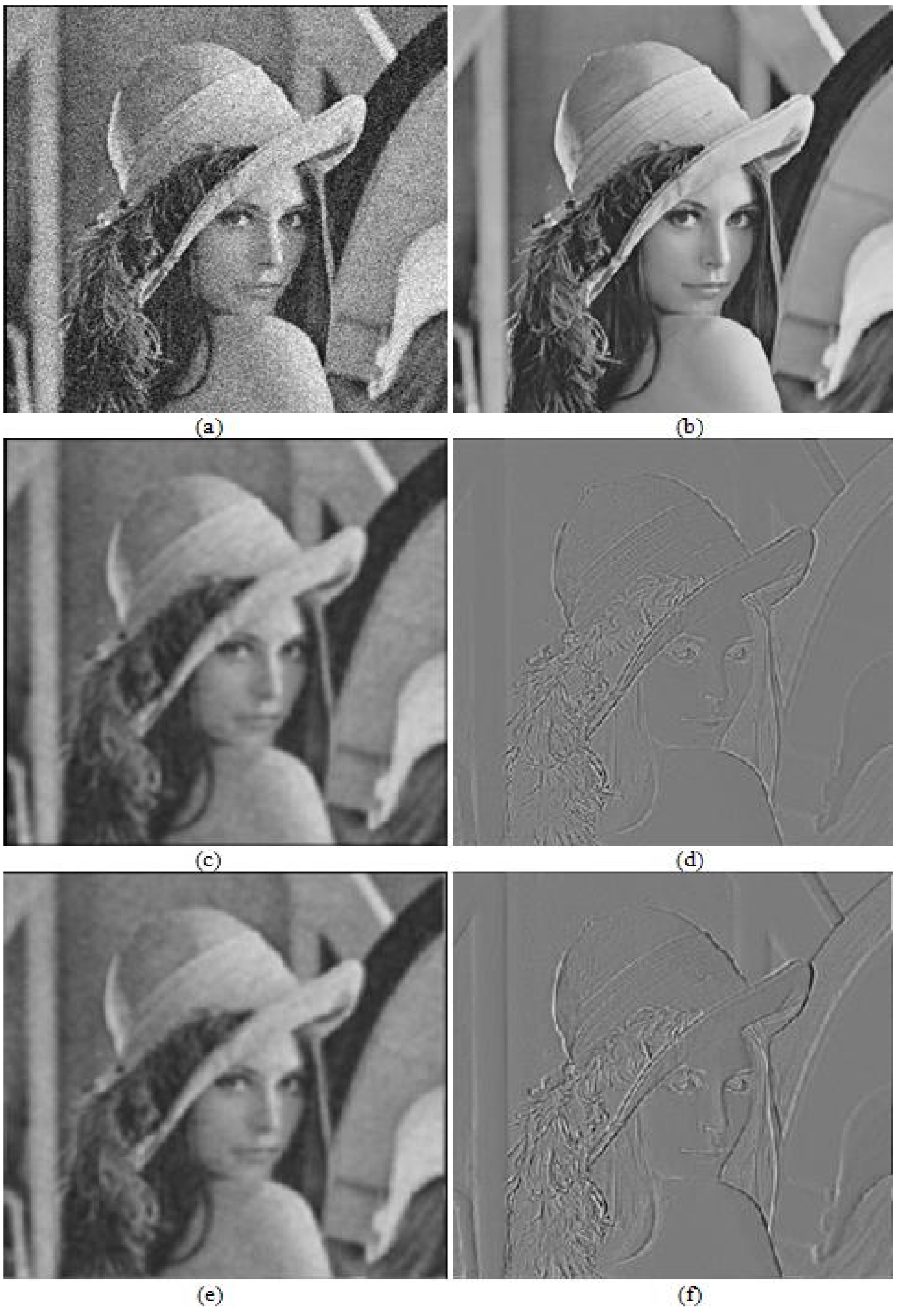
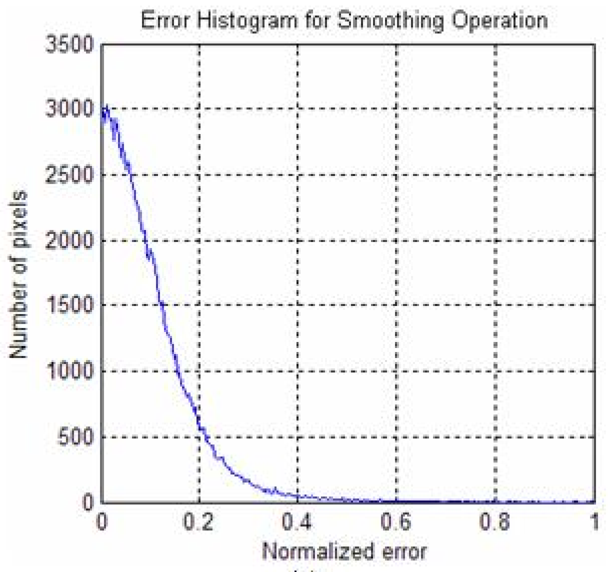
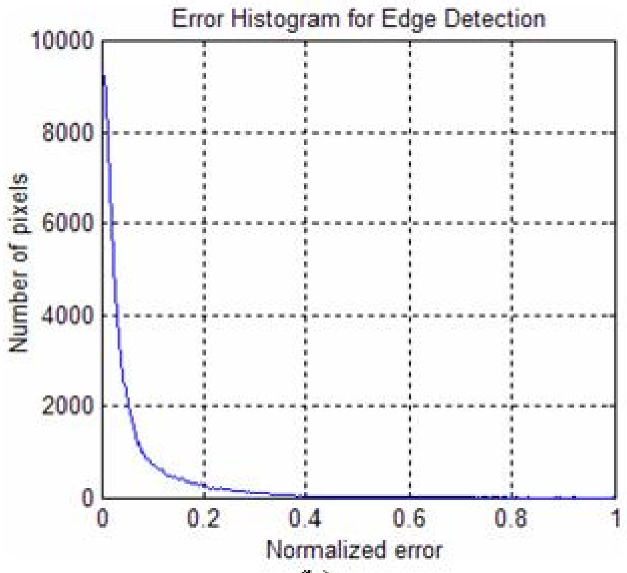


Fig. 6. (a) and (b) are test images for convolution operation with low-pass and high-pass transfer functions, respectively. (c) and (d) are the output images produced by Matlab software. (e) and (f) are the results from hardware simulation.



(a)



(b)

Fig. 7. Histograms of error for (a) noise filtering and (b) edge detection kernels. The corresponding average errors are 2.19 and 4.17 pixel intensities with the peak error at 22.23 and 70.97. The reason such large peak errors exist is that the range of kernel coefficients is broader with well distributed histogram on the test images.

5 Performance-Resource Optimization

5.1 Hardware Utilization of Minimum PEA Configuration

The hardware resource utilization is characterized based on the Xilinx's Virtex II XC2V2000-4ff896 FPGA on the multimedia platform and the Integrated Software Environment (ISE). The particular FPGA chip we target has 10,752 logic slices, 21,504 flip-flops (FFs), 21,504 lookup tables (4-input LUTs), 56 block RAMs (BRAMs), and 56 embedded 18-bit signed multipliers in hardware; however, we do not utilize the built-in multipliers. The resource allocation for various sizes of the kernels with minimum PEA is shown in Table 1. For 16×16 kernels, the computational power is 8 log-domain additions instead of 256 multiplications with fully parallel approach. The maximum windows can be utilized on target FPGA is mainly constraint by the storage capacity rather than the LUTs. In addition to trading the performance for resource, the design utilizes approximately 60% less hardware resource when compared to the quadrant symmetric architecture which implements fully pipelined multipliers.

5.2 Performance Evaluation of Minimum PEA Configuration

The critical timing analysis of Xilinx's ISE shows that the 190.922MOPS of the PEA is the most optimal throughput achievable with the maximum clock frequency of 190.922 MHz. The overall output of convolution architecture for various sizes is shown in the last column of Table 1. It should be clear that the performance decreases with large kernels. The propagation delay of log-based architecture is comparable to Xilinx's enhanced multiplier-based architecture which is 0.21ns faster. Further evaluation of pipelining the critical path suggests that increasing the level of pipelining does not gain significant throughput rate. Given 1024×1024 image frame, it can process over 45.5 frames per second without frame buffering with 8×8 kernel. Characterization of optimization scheme is discussed next.

5.3 Characterization of Optimization Scheme

In this section, we characterize the relationship between the performance and the resource allocation of the architecture in various configurations. We first analyze

Table 1. Hardware resource utilization for various sizes of the kernels with its corresponding performance indicates the overall effectiveness of the architecture

Kernel Size	Logic Slices	Slice FFs	LUTs	BRAMs	Perf (MOPS)
4×4	3%	2%	2%	2	95.46
8×8	6%	5%	3%	4	47.73
10×10	9%	7%	4%	4	38.18
12×12	12%	10%	4%	5	31.82
13×13	15%	13%	5%	6	27.28
15×15	19%	16%	6%	7	23.87
16×16	19%	17%	6%	7	23.87

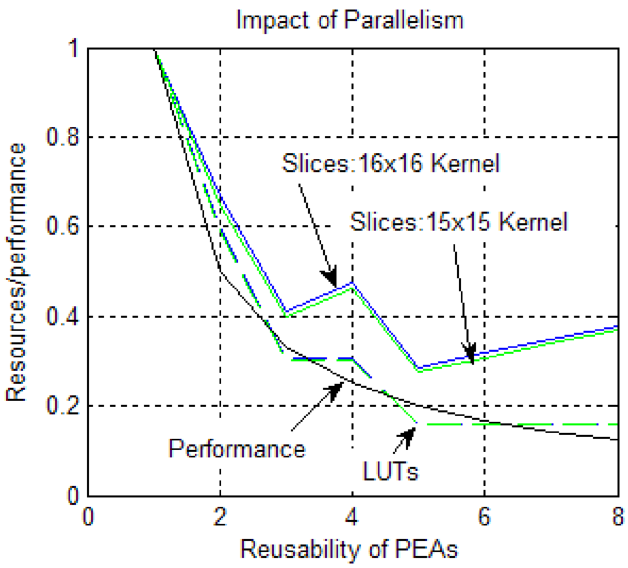


Fig. 8. The impact of reuse of PEAs over the performance and resource utilization of the architecture

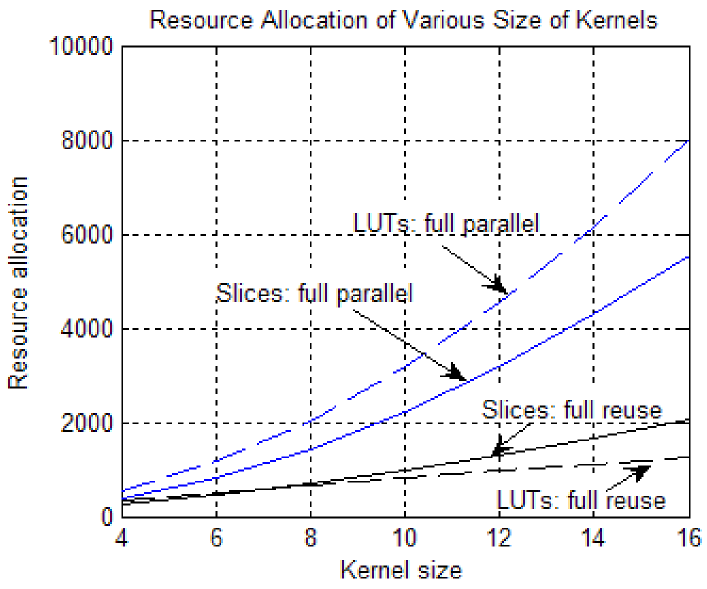


Fig. 9. The proportionality between the performance and resource allocation for various sizes of kernels

the impact of the parallelism on performance and resource requirement with fixed J_1 and J_2 attributes, as graphed in Fig. 8. The x-axis indicates the frequency of reuse of PEAs (i.e. the f attribute in(6)) while the y-axis is the resource and performance quantities normalized by the design parameters configured to the fullest parallelism. Fig. 8 suggests that the resource utilization for even and odd sizes of kernels has minor difference. A more interesting point should be noted is that the performance is directly proportional to the utilization of the LUTs with varying f attribute. The Slices also tend to increase with larger f since storage capacity must be increased for the DQ queues, LKC queues, and output queues of the PEs. While the performance-resource has exponential relationship to the kernel size, its trading is nearly linear with f as depicted in Fig. 9. For example, the ratio for the LUTs of fully parallel and reused configurations is about 8 (f =8) while performance of fully reused architecture is reduced by $1/f$.

6 Conclusion

A technique for optimization of 2D convolution architecture with quadrant symmetric kernels is being presented in this paper to balance the performance and hardware resource based on the constraints imposed by the application. The design regularizes the reusability of the core components determined by application specification. We also demonstrated the generalized procedure for realization of the low complexity architecture with effective data path routing scheme and the assistance of logarithmic estimation modules. For the implementation highly limited to the resource, we showed that the architecture is able to sustain 45.5 1024×1024 frames per second (fps), which is more than real-time criteria of 30 fps, with minimum resource of 6% Slices and 3% LUTs for 8×8 kernels in a Xilinx's Virtex XC2V2000-4ff896 FPGA at maximum clock frequency of 190.92 MHz. We also determined that the optimization of the performance and resource has a relationship of $1/f$, where the f is evaluated by the application. With larger f, both normalized performance and resource are reduced to $1/f$.

References

1. Breitzman, A.F.: Automatic Derivation and Implementation of Fast Convolution Algorithms. PhD Dissertation, Drexel University (2003)
2. Jamro, E.: Parameterised Automated Generation of Convolvers Implemented in FPGAs. PhD Dissertation, University of Mining and Mentallurgy (2001)
3. Chang, H.M., Sunwoo, M.H.: An Efficient Programmable 2-D Convolver Chip. In: Proc. of the 1998 IEEE Intl. Symp. on Circuits and Systems, ISCAS, vol. 2, pp. 429–432. IEEE Computer Society Press, Los Alamitos (1988)
4. Hecht, V., Ronner, K., Pirsch, P.: An Advanced Programmable 2-D Convolution for Real Time Image Processing. In: Proc. of IEEE Intl. Symp. on Circuits and Systems, ISCAS, pp. 1897–1900. IEEE Computer Society Press, Los Alamitos (1991)
5. Kim, J.H., Alexander, W.E.: A Multiprocessor Architecture for 2-D Digital Filters. IEEE Trans. On Computer C-36, 876–884 (1987)
6. Nelson, A.E.: Implementation of Image Processing Algorithms on FPGA Hardware. MS Thesis, Vanderbilt University (2000)
7. Lee, S.Y., Aggarwal, J.K.: Parallel 2-D Convolution on a Mesh Connected Array Processor. IEEE Trans. On Pattern Analysis and Machine Intelligence PAMI-9, 590–594 (1987)

8. Bosi, B., Bois, G.: Reconfigurable Pipelined 2-D Convolvers for Fast Digital Signal Processing. IEEE Trans. On Very Large Scale Integration (VLSI) Systems 7(3), 299–308 (1999)
9. Wiatr, K., Jamro, E.: Constant Coefficient Multiplication in FPGA Structures. In: IEEE Proc. of the 26th Euriomicro Conference, Maastricht, The Netherlands, vol. 1, pp. 252–259 (2000)
10. Samueli, H.: An Improved Search Algorithm for the Design of Multiplier-less FIR Filters with Powers-of-Two Coefficients. IEE Trans. Circuits systems, 1044–1047 (1989)
11. Ye, Z., Chang, C.H.: Local Search Method for FIR Filter Coefficients Synthesis. In: Proc. 2nd IEEE Int. Workshop on Electronic Design, Test and Applications (DELTA-2004), Perth, Australia, pp. 255–260. IEEE Computer Society Press, Los Alamitos (2004)
12. Yli-Kaakinen, J., Saramäki, T.: A Systematic Algorithm for the Design of Multiplierless FIR Filters. In: Proc. IEEE Intl. Symp. Circuits Syst., Sydney, Australia, pp. 185–188. IEEE Computer Society Press, Los Alamitos (2001)
13. Kung, H.T., Ruane, L.M., Yen, D.W.L.: A Two-Level Pipelined Systolic Array for Multidimensional Convolution. Image and Vision Computing 1(1), 30–36 (1983)
14. Lakshmanan, V.: A Separable Filter for Directional Smoothing. IEEE Transaction on Geoscience and Remote Sensing Letters 1(3), 192–195 (2004)
15. Zhang, M.Z., Ngo, H.T, Asari, V.K.: Design of an Efficient Multiplier-Less Architecture for Multi-dimensional Convolution. In: Srikanthan, T., Xue, J., Chang, C.-H. (eds.) ACSAC 2005. LNCS, vol. 3740, pp. 65–78. Springer, Heidelberg (2005)
16. Mitchell, J.N.: Computer Multiplication and Division Using Binary Logarithms. IRE Transactions on Electronic Computers, 512–517 (1962)
17. SanGregory, S.L., Brothers, C., Gallagher, D., Siferd, R.: A Fast Low-Power Logarithm Approximation with CMOS VLSI Implementation. In: Proc. of the IEEE Midwest Symposium on Circuits and Systems, vol. 1, pp. 388–391. IEEE Computer Society Press, Los Alamitos (1999)
18. Zhang, M.Z., Asari, K.V.: An Efficient Multiplier-less Architecture for 2-D Convolution with Quadrant Symmetric Kernels. Integration, the VLSI Journal (in print)

Bipartition Architecture for Low Power JPEG Huffman Decoder

Shanq-Jang Ruan and Wei-Te Lin

Low Power Systems Lab, Department of Electronic Engineering
National Taiwan University of Science and Technology,
No.43, Sec.4, Keelung Rd., Taipei, 106, Taiwan, R.O.C.
sjruan@mail.ntust.edu.tw

Abstract. JPEG codec in portable device has become a popular technique nowadays. Because the portable device is battery powered, reducing power dissipation is practical. In this paper, a low power design technique for implementing JPEG Huffman decoder is presented, in which the Huffman table is divided into two partitions. As the main contribution, we propose a low power Huffman decoder with bipartition lookup table to reduce the power consumption in JPEG. Experimental results of the gate level simulation show that the proposed method can reduce the power consumption by 15% on average compared to general Huffman decoder.

Keywords: Huffman, Decoding, Low power, Bipartition.

1 Introduction

JPEG image compression [1] is one of the most commonly used standards for communication and storage applications. It is a block-based system whose structure consists of three major steps: 2-D discrete cosine transformation (DCT), quantization, and entropy coding.

The entropy coding for JPEG is a combination of run-length coding (RLC) and Huffman coding [2]. The basic idea of the Huffman code is to use variable-length code to compress input symbols based on the probability of the symbol's occurrence. The most frequently used symbols are represented with shorter codes, while less frequently occurring symbols have a longer representation. According to the probability of the symbol's occurrence, we can use the bipartition architecture [3], [4] to divide the lookup table.

JPEG compression in portable device has become a popular technique nowadays, such as laptop computer, personal digital assistant (PDA) and cellular phone. Because these applications are battery powered, reducing power dissipation is practical. Most papers only discuss the memory usage efficiency and decoding speed on Huffman decoder, such as [5], [6], [7] etc. In this paper, we propose a bipartition lookup table for JPEG Huffman decoder which is based on the concept of the bipartition architecture and the parallel VLC decoding [7],

© Springer-Verlag Berlin Heidelberg 2007

[8], [9] together with canonical codes [10], [11] in the design of Huffman decoder to reduce power consumption.

The rest of the paper is organized as follows. In Section 2, we briefly introduce the background of our method. The proposed technique and the design of VLSI architecture using this proposed method are presented in Section 3. We present experimental results in Section 4. Section 5 is conclusion.

2 Background

2.1 Bipartition Architecture

Consider the circuit of Fig. 1. A pipelined stage is a combinational logic block separated by distinct registers. The bipartition architecture [3], [4] is shown in Fig. 2. The combinational logic portion in Fig. 1 is partitioned into two groups. One that includes some states of high activity is small called $Group_1$ and the other that includes the remainder of low activity is big called $Group_2$. The two groups work in turns. The GCB is a precomputation block for only one block is working at the same time.

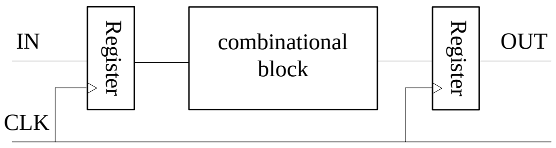


Fig. 1. A combinational pipelined circuit

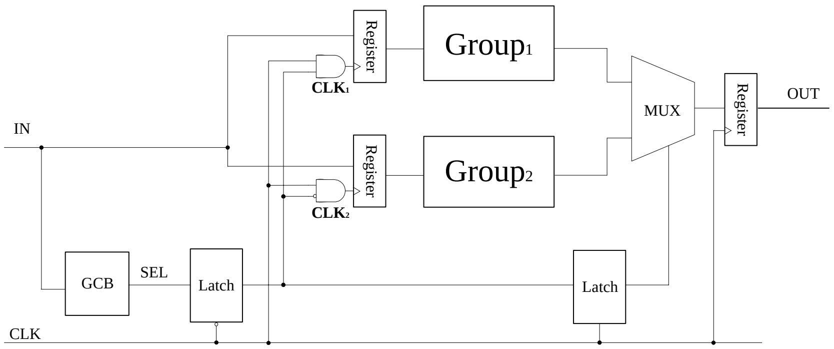


Fig. 2. Bipartition architecture

2.2 Parallel VLC Decoding

The parallel VLC decoder architecture provides the constant-output-rate VLC decoding. It mainly consists of two units: a barrel shifter for the alignment of

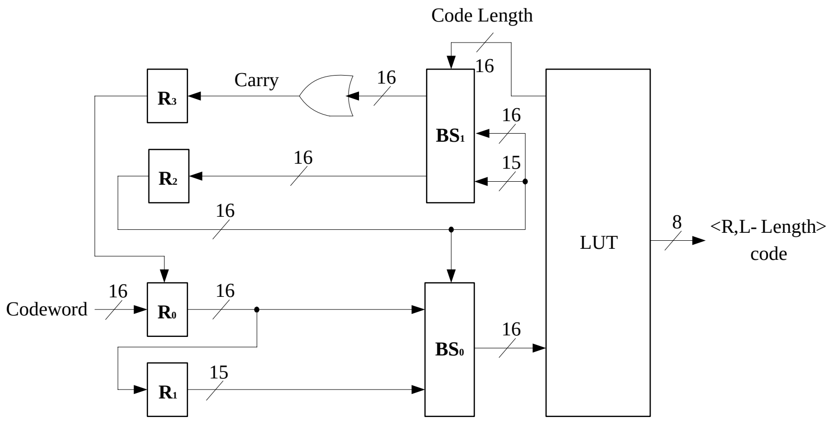


Fig. 3. Parallel VLC decoding [8]

incoming codeword and a lookup table for generation of output symbols and codeword length.

A block diagram of a parallel VLC decoder is shown in Fig. 3 [8]. The functions of its major components are described as follows. The LUT matches a codeword and outputs the corresponding symbol and codeword length. The codeword length is accumulated by the barrel shifter BS_1 and the register R_2. The barrel shifter BS_0 then shifts the next codeword according to this accumulated codeword length. The input data are stored in the registers R_0 and R_1. R_2 represents the number of decoded bits in R_1. It controls the barrel shifter BS_0.

3 Proposed Scheme

In this Section, a new method to rearrange and partition the Huffman table for the JPEG Huffman decoder is proposed. There are two merits in this new method. First, it makes no change at the encoding end because the Huffman table used in the decoding end is the original one. Secondly, the power consumption will be reduced due to the bipartition lookup table technique.

3.1 Bipartition Lookup Table Technique

The property of Huffman codes [1] that belongs to variable length code (VLC) [9] is that the shorter codewords are assigned to symbols which are frequent and the longer codewords are assigned to symbols which are rare. There are 125 codewords with the longest length in the 162 codewords on the Huffman table for JPEG standard as shown in Table 1. These codewords will cause wasting power with the look-up table method. These codewords consists of first 9 bits of the number of 1s and the remaining other 7 bits. Here we partition these codewords which the first 9 bits are the number of 1s into LUT1. On the other hand, the other codewords with shorter length are partitioned into LUT0.

Table 1. Huffman table for JPEG

Codeword	Symbol	CL
00	0/1	2
01	0/2	2
100	0/3	3
1010	EOB	4
1011	0/4	4
1100	1/1	4
11010	0/5	5
11011	2/1	5
111000	0/6	6
111001	1/2	6
⋮	⋮	⋮
11111111010	D/1	11
111111110110	E/1	12
111111110111	F/0	12
111111111000000	8/2	15
1111111110000010	0/9	16
1111111110000011	0/A	16
⋮	⋮	⋮
1111111111111101	F/9	16
1111111111111110	F/A	16

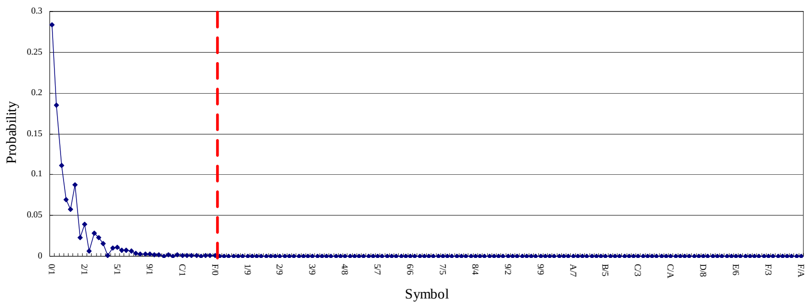


Fig. 4. Probabilities of Huffman table for JPEG

Fig. 4 shows the probabilities of symbols in Huffman table for JPEG. As we can see from this figure, the most frequently symbols are in the left side of the broken line and the less frequently symbols are in the right side. Therefore, we divide the lookup table into two partitions.

The flowchart of the proposed method is shown in Fig. 5. If "End of File" is reached, the Huffman decoding process is completed; otherwise, the Huffman

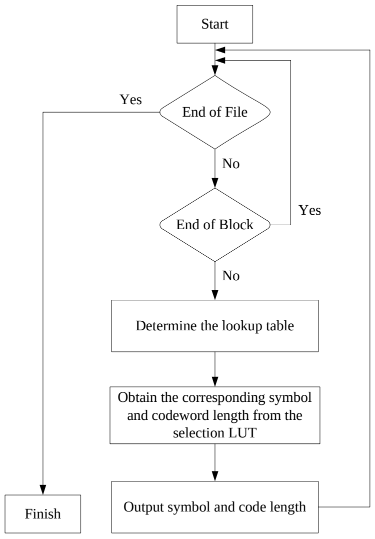

Start
End of File
Yes
No
End of Block
Yes
No
Determine the lookup table
Obtain the corresponding symbol and codeword length from the selection LUT
Output symbol and code length
Finish

Fig. 5. Flowchart of proposed method

decoder using the proposed method will proceed following Steps 1–3, which are discussed in detail below.

Step 1: Determine the lookup table.
Given a coded Huffman codeword, the first 9 bits are read to determine directly the LUT select bit through the LUT selector unit. If the first 9 bits are all 1s, the LUT select bit will be 1 and the lookup table LUT1 will be activated. Otherwise, the LUT select bit will be 0 and the lookup table LUT0 will be activated.

Step 2: Obtain the corresponding symbol and codeword length.
The LUT consists of two parts, the first part is the run/length symbol and the second part is codeword length. The corresponding symbol and codeword length will be obtained in selected LUT after step 1.

Step 3: Output symbol and codeword length.
Finally, obtain the correct symbol and codeword length from the selected LUT. The symbol is outputted to the next stage of the JPEG decoding and the codeword length is received by the codeword shifter block to shift to the next codeword.

3.2 VLSI Architecture for Bipartition Lookup Table Technique

The proposed method described in Section 3.1 can be mapped efficiently to a VLSI architecture. There are three major blocks in this architecture, namely, the codeword shifter block, the bipartition decoding block, and the symbol output block, as shown in Fig. 6. First, the codeword shifter block is designed to accumulate codeword length and shift to the next codeword according to the accumulated codeword length. Next, the bipartition decoding block receives the parallel data and then yields a LUT select bit to select the correct lookup table to extract the symbol and codeword length as mentioned in Steps 1 and Step 2. The third block, the Symbol output block, determines the output symbol and codeword length from the selected lookup table. The detailed descriptions of these three architectures are given in Sections 3.2.1–3.2.3.

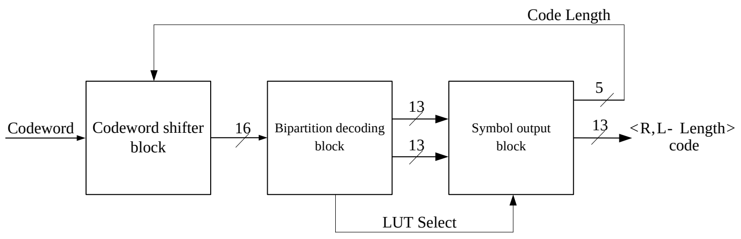


Fig. 6. Huffman decoder using proposed method

3.2.1 Codeword Shifter Block

The codeword shifter block consists of codeword register, bits barrel shifter and accumulator as shown in Fig. 7. The input data are stored in codeword register. The codeword length is accumulated by accumulator. The new accumulated codeword length controls the barrel shifter to shift to the correct position for the next decoding cycle.

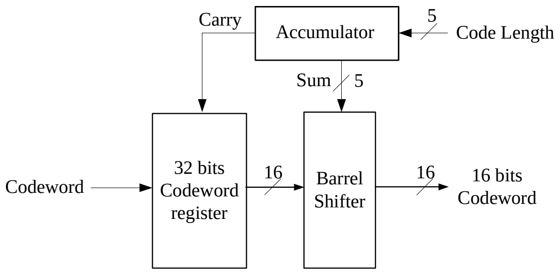


Fig. 7. Codeword shifter block

3.2.2 Bipartition Decoding Block

The bipartition decoding block is the key block of our method. First, this block receives the 16-bit parallel data from the codeword shifter block and decides the LUT select bit. The LUT select bit determines which lookup table, LUT0 or LUT1, will be used to extract the symbol and codeword length. If the LUT select bit is 0, the lookup table LUT0 will be selected. Otherwise, the lookup table LUT1 will be selected when the LUT select bit is 1. A schematic diagram representing the bipartition decoding block is shown in Fig. 8.

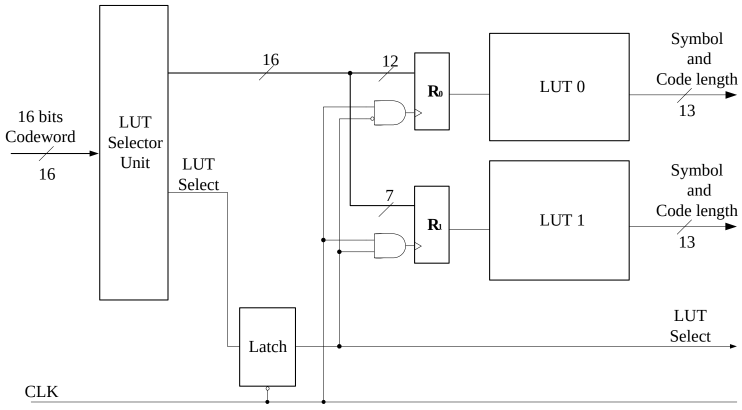


Fig. 8. Bipartition decoding block

3.2.3 Symbol Output Block

The Symbol output block consists of a multiplexer, latch and 13 bits output register as shown in Fig. 9. In this block, the LUT select bit controls the multiplexer to select the correct output. Once the output is determined, the accumulator in the codeword shifter block will obtain the codeword length from the output register.

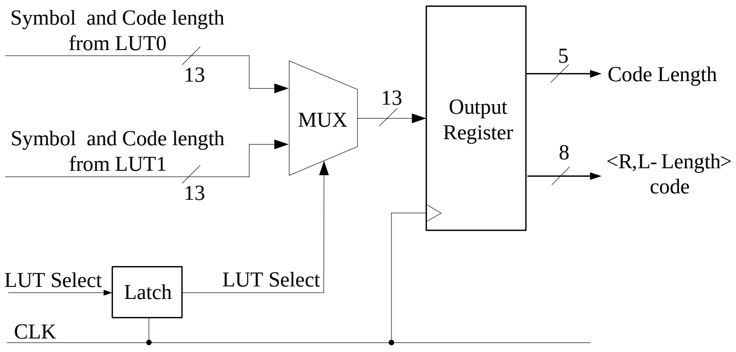


Fig. 9. Symbol output block

4 Experiments

4.1 Experimental Setup

To compare the effectiveness of our proposed method and the general parallel Huffman decoder [5], [6], we synthesize these two methods using Synopsys Design compiler, with a 0.13um, 1.3-V technology library from TSMC and the power consumption is obtained by Synopsys PrimePower. The test images are shown in Fig. 10 and the codewords belong to which LUT, LUT0 or LUT1, are shown in Table 2.

Fig. 10. Test images

Table 2. Huffman table for JPEG

Image	Total coldewords	Codewords belong to the LUT0	Codewords belong to the LUT1
Barbara	61580	61316 (99.57%)	264 (0.43%)
Goldhill	61299	61224 (99.88%)	75 (0.12%)
Jet	44228	44023 (99.5%)	205 (0.5%)
Lena	44526	44482 (99.9%)	44 (0.1%)
Mandril	95279	95137 (99.85%)	142 (0.15%)
Peppers	46767	46656 (99.76%)	111 (0.24%)

4.2 Experimental Results

Fig. 11 shows experimental results that compare our bipartition technique with the general method. The first bar for each test image represents the result of power consumption using general parallel Huffman decoder. The second bar shows the result of power consumption using bipartition parallel Huffman de-

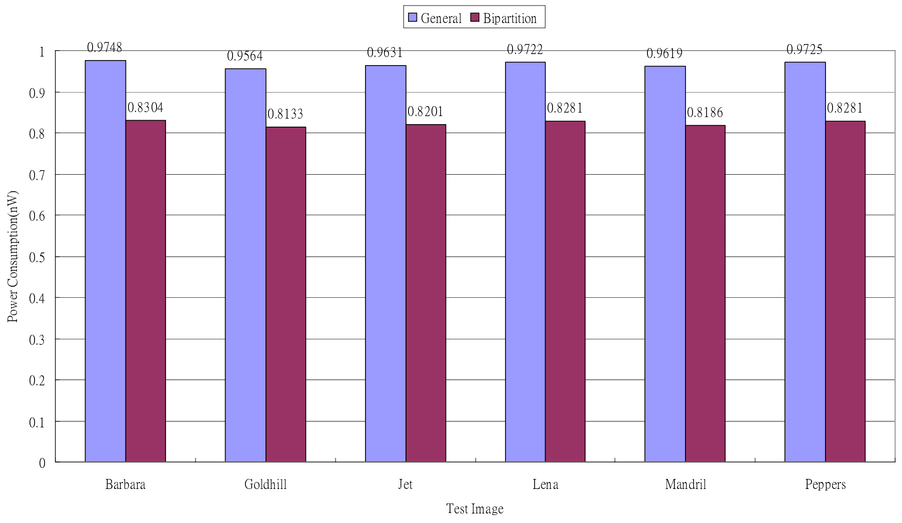


Fig. 11. Power consumption of the general method and bipartition technique

coder. Our bipartition technique and the general method both were executed with the same predetermined test image.

As we can see from the results, our method reduces the power consumption by 15% on average.

5 Conclusion

In this paper, we have proposed a power-efficient method to reduce the decoding power consumption for Huffman decoder of JPEG. The proposed method reduces decoding power at the bipartition lookup table technique. Our scheme reduces the power consumption of Huffman decoder of JPEG by 15% on average compared with the general method. Our experimental results show that bipartition LUT help improve the power reduction of JPEG Huffman decoder effectively

References

1. Bhaskaran, V., Konstantinides, K.: Image and video compression standards algorithm and architectures. Kluwer Academic Publishers, Dordrecht (1999)
2. Huffman, D.A.: A method for the construction of minimum-redundancy codes. In: Proc. Institute of Radio Engineers(IRE), September 1952, vol. 40, pp. 1098–1101 (1952)
3. Ruan, S.J., Shang, R.J., Lai, F., Chen., S.J., Huang, X.J.: A bipartition-codec architecture to reduce power in pipelined circuits. In: Proc. IEEE/ACM Int. Conf. Computer-Aided Design, November 1999, vol. 20, pp. 84–89. ACM Press, New York (1999)
4. Ruan, S.J., Shang, R.J., Lai, F., Chen., S.J., Huang, X.J.: A bipartition-codec architecture to reduce power in pipelined circuits. IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems 20(2), 343–348 (2001)
5. Choi, S.B., Lee, M.H.: High speed pattern matching for a fast huffman decoder. IEEE Trans. Consum. Electron 41, 97–103 (1995)
6. Jiang, J.H., Chang, C.C., Chen, T.S.: An efficient huffman decoding method based on pattern partition and look-up table. In: Fifth Asia-Pacific Conf. on Communications and Fourth Optoelectronics and Communications, vol. 2, pp. 904–907 (1999)
7. Chang, Y.W., Truong, T.K., Chang, Y.: Direct mapping architecture for jpeg huffman decoder. IEE Proc. Communications 153, 333–340 (2006)
8. Lei, S.M., Sun, M.T.: An entropy coding system for digital hdtv applications. IEEE Trans. Circuits and Systems for Video Technology 1, 147–155 (1991)
9. Chang, S.F., Messerschmitt, D.: Designing high-throughput vlc decoder part i - concurrent vlsi architectures. IEEE Trans. Circuits and Systems for Video Technology 2, 187–196 (1992)
10. Moffat, A., Turpin, A.: On the implementation of minimum redundancy prefix codes. IEEE Trans. Communications 45, 1200–1207 (1997)
11. Nekritch, Y.: Decoding of canonical huffman codes with look-up tables. In: Proc. Data Compression Conf., vol. 566 (March 2000)

A SWP Specification for Sequential Image Processing Algorithms*

Wensheng Tang, Shaogang Wang, Dan Wu, and Wangqiu Kuang

Computer School, National University of Defense Technology
Changsha, Hunan, 410073 China
13327311970@hn165.com

Abstract. To retarget sequential image processing algorithm written in sequential languages (e.g. C) to processors with multimedia extensions and multimedia-specific embedded microprocessors is an open issue. Image processing algorithms are inherently with sub-word parallelism (SWP) But current compilers are not able to exploit it by locating SWP within a basic block. This paper proposes a program representation and pattern-matching approach for generating an explicit SWP specification from sequential source code. The representation is based on an extension of the multidimensional synchronous dataflow (MDSDF) model of computation. For both the compiler and source programs should not be user-modified, we extend compiler's functionality by adding a specialized pattern-library. After data-flow and control-flow analysis with pattern matching, the generated SWFG (Sub-Word Flow Graph) can be used as an intermediate representation for the next step of compiler for SWP instruction selection and code generation.

Keywords: image-processing, SWP, pattern matching, SWFGcode selection.

1 Introduction

Nowadays, multimedia applications are widely used. Characteristics of multimedia applications bring a special micro-processing in processor [1]. As a consequence, the high processing power which used to be needed only for research or military projects is now required at home by millions of users. Instead of designing and developing SIMD arrays with expensive, specialized coprocessors, many microprocessor vendors add multimedia extensions to processors, designed to complement the "regula" instructions and improve the performances in today's multimedia applications. Besides, many DSP designers begin to develop embedded specific microprocessors for processing multimedia applications. Image-processing programs are inherently parallel and small-data. More memories included and number of calculation units and/or their functionality increases, raises complexity in processor manufacturing. An effective method to exploit available functionality for media applications is sub-word parallelism (SWP).

* Sponsored by National Natural Science Foundation of China under Grant NO.60173040.

© Springer-Verlag Berlin Heidelberg 2007

1.1 Common Issues in SWP Compilation

SWP It allows the parallel calculation of 2, 4, or 8 similar operations in one calculation unit (SWP unit). Fig. 1 (a) shows an example of 32-bit SWP processing.

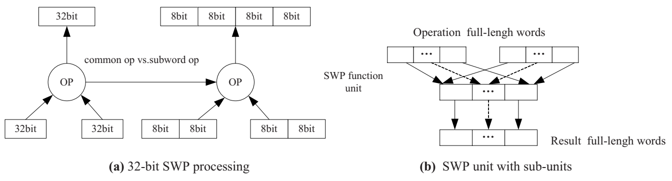


Fig. 1. SWP processing and SWP unit with sub-units

Packing/unpacking. As shown in Fig. 1 (b), the operands (sub-words, data) in SWP function unit have to be packed from two full-length operation words. After calculation in the SWP function unit in parallel, the results are unpacked in a full-length word.

SWP instruction utilization. Current approaches suffer from shortcomings like lack of portability and high cost of software development. Thus, VLIW-based media architecture is believed to be fit for exploiting SWP, with compilers ability of parallelism exploitation and dynamically schedule, besides its flexible datapath.

1.2 Existing Compilation Techniques

The current problem is that, on one hand we have modern multimedia execution hardware, but on the other hand, the software and the general compilers are not able to automatically exploit this inherent parallelism and can not make full use of the multimedia instruction set.

Some researches are explored to find a way for compiling techniques. Some of them are techniques used for SIMD processors to generate optimized code. One such technique, known as vectorization [2,3], can indeed give interesting results when implemented for MMX [2] or VIS [4,3]. The main motivation behind vectorization is that in computation intensive applications such as multimedia applications where some processing is applied to large data sets containing small elements-loops are the most critical part of the code and should present a large amount of parallelism. Thus, one solution to optimize the whole application is to detect these loops that can be parallelized, and transform them into a vector operation, operating on infinite length vectors. This operation is then transformed in a loop using the SWP operations.

Another well-known compiler effort came from Larsen who has implemented a superword parallelism module in SUIF [5]. They adopt a heuristic approach which can produce good results in a very reasonable time. An advantage of

SLP over vectorization is that it does not require large amount of parallelism to be profitable. SLP can be seen as a restricted form of ILP. Indeed, executing the same instruction on all subwords of a packed register is about the same as executing a few instructions at the same time, each performing the operation on one of the subwords. However, the problem is that there may be several different packing possibilities within the same basic block. S. Larsen proposed in [5] an optimal algorithm to extract SLP. Unfortunately, his approach is far too complex to be computable, which is why he also gives a heuristic.

An interesting approach to parallelization of code is that taken by M. Boekhold, I. Karkowski and H. Corporaal in [6]. Instead of having a compiler with multiple passes, they propose to have a single transformation engine, which can replace code matching a specific pattern by another code fragment, provided some use-provided conditions are respected. In this way, the parallelization can be extracted quite simply, by providing the correct transformation specifications. PAP [7] and PARAMAT [8] have employed program recognition to extract parallelism from loop kernels. The PAP system, targeted at distributed memory architectures, creates an annotated program dependence graph [9] representing the loop, and then performs pattern-matching to identify groups of structures s concepts. Sets of concepts can then be aggregated into higher-level concepts and, eventually, transformed into parallelized code. PARAMAT is similar, but uses abstract syntax trees as the basic program representation. Both PAP and PARAMAT attempt to identify sequential algorithms in the code and replace them with well-known parallel implementations of those algorithms.

1.3 Our Approach

Our research goal is to retarget sequential image processing algorithm written in sequential languages (e.g. C) to processor with multimedia extensions and multimedia-specific embedded microprocessors, without modifying source programs. To implement it, we design a compiler framework consisting of some passes. In this compiler framework, a key part is specialized knowledge-based pattern library and an intermediate representation- SWFG (Sub-Word Flow Graph). After pattern matching and recognition, parallelism candidates are explicitly represented amd SWP instruction selection can be realized.

Approach described in this paper is based on loop analysis and dependence analysis. But differs from current methods, our approach tends to present explicit parallelism, be available for embedded microprocessor instead of SIMD arrays and provide a flexible way to retarget sequential programs to parallel execution. SWFG will be described in section 2 and the experiment design in section 3.

2 Sub-Word Flow Graph

2.1 Bitwise Data Flow Analysis

Data flow analysis helps us to determine inner loop. We use loop normalization to ensure that the iteration space of the loops is regular and the process of dependence verification is simple. For a C compiler the lower bounds are set to 0 and the step is set to 1, making amount of all loops iterative executed. And the old induced variables are replaced by the affine function with new induced variables. The index expressions and the lower bounds are modified accordingly. A small loop with lower bound 2 is shown as follows.

We use lattice as a formal ordering of the internal data structure in data flow analysis, which traditionally can not yield accurate results on arithmetic operations or support precise arithmetic analysis.

We select lattice of propagating data-ranges as the structure in data flow analysis. In this structure, a data range is a single connected subrange of the integers from a lower bound to an upper bound. Thus a data-range keeps track of a variable's lower and upper bounds. Because only a single range is used to represent all possible values for a variable, this representation does not permit the elimination of low-order bits. However, it does allow us to operate on arithmetic expressions precisely.

The propagation of data range lattice is shown in Fig. 2. Lattice represents the life ranges of values that can be assigned to a variable and is lifted from a bottom element. Definitions and computations of the value in this lattice are listed in Table 1. Take the code of motion estimation kernel of the MPEG-4 decoder algorithm as an example, after analysis, we find true dependence and output dependence in this code segment. The data-dependence graph of this loop is illustrated in Fig. 3 left. After loop-distribution, the strong connected component is generated, as shown in Fig. 3 (right).

A strong connected component of a dependence graph is a maximal set of vertices in which there is a directed path between every pair of vertices in the set. A non-singleton SCC of a data dependence graph represents a maximum set of statements that are involved in a dependence cycle. We can fine out that

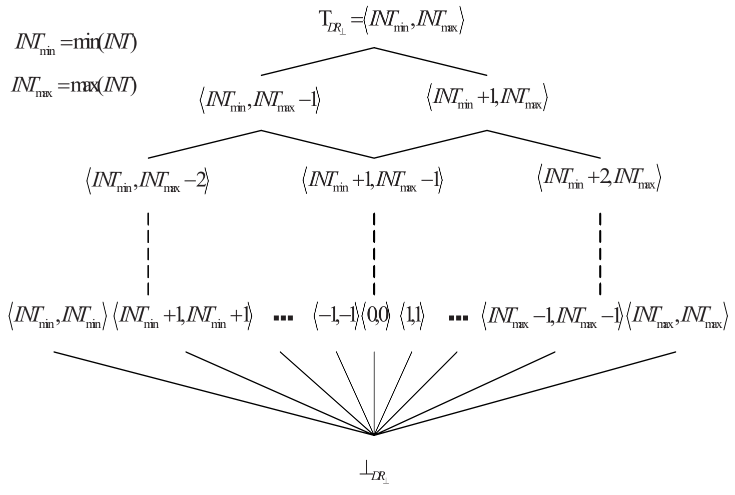

$INT_{min} = min(INT)$
$INT_{max} = max(INT)$
$T_{DR_\perp} = \langle INT_{min}, INT_{max} \rangle$
$\langle INT_{min}, INT_{max}-1 \rangle$
$\langle INT_{min}+1, INT_{max} \rangle$
$\langle INT_{min}, INT_{max}-2 \rangle$
$\langle INT_{min}+1, INT_{max}-1 \rangle$
$\langle INT_{min}+2, INT_{max} \rangle$
$\langle INT_{min}, INT_{min} \rangle$
$\langle INT_{min}+1, INT_{min}+1 \rangle$
$\langle -1,-1 \rangle$
$\langle 0,0 \rangle$
$\langle 1,1 \rangle$
$\langle INT_{max}-1, INT_{max}-1 \rangle$
$\langle INT_{max}, INT_{max} \rangle$
$\perp_{DR_\perp}$

Fig. 2. Lattice representing the life ranges of values that can be assigned to a variable

Table 1. Lattice definition and value computation

$\perp_{DR_\perp}$	The value of subword's life range that have not been initialized
$\mathrm{T}_{DR_\perp}$	The value that can not be statically determined. It is the upper bound of subwords-set's life range.
$\cup$	Life range union. The union over the single connected subrange of the integers where $\langle a_l, a_h \rangle \cap \langle b_l, b_h \rangle = \langle \min(a_l, b_l), \max(a_h, b_h) \rangle$
$\cap$	Life range intersection. The set of all integers in both subranges $\langle a_l, a_h \rangle \cap \langle b_l, b_h \rangle = \langle \max(a_l, b_l), \min(a_h, b_h) \rangle$

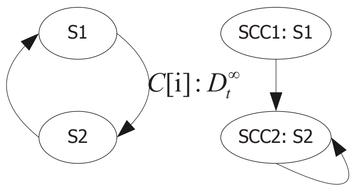


Fig. 3. Data-dependence graph and strong connected components

statement S1 forms a singleton SCC (without a self-arc). Hence, S1 can be executed in subword way. Statement S2 is self-dependent during the loop. To eliminate this dependence, we can do loop-unrolling on S2.

The loop is first unrolled the correct number of times depending on the type of the operands. If register is 32-bit, the loop is unrolled 2 times for short int operands. Then the loop body is inspected and acyclic instruction scheduling is performed in order to have all instances of the same loop instruction grouped together. Hence, S2 can also be executed in subword way.

2.2 SWFG Representation

Our SWFG representation is based on the multidimensional synchronous dataflow (*MDSDF*) model of computation (MOC) [10]. In SDF models, each node is a process that produces (consumes) tokens on its output (input) arcs at integer rates. These integer rates of production and consumption allow the execution rates of each process to be statically determined, leading to a static scheduling of all processes. SDF models are valuable in modeling signal processing applications and generating efficient code for programmable embedded DSP processors [11]. *MDSDF* augments SDF with multielement, multidimensional tokens. These tokens provide a useful representation for image processing algorithms, which often operate on regular subregions of an image, such as rows, columns, and tiled blocks.

Our SWFG representation extends the *MDSDF* representation in two ways. The first is that edge annotations include the bounds information, e.g., (1:8:1, 0:8:2) specifies a two-dimensional token with eight elements in the first dimension, beginning at position one, and having a stride of one. Similarly, the second dimension has eight elements, begins at position zero, and has a stride of two between each element. This preserves information from the original source concerning loop iteration ranges and array boundary conditions. The second is the notation of operation area, which explicitly describes the parallel operations on separated units of different sub-word.

2.3 Initial Representation of Source Program

To facilitate recognition, the C source is parsed and translated into an intermediate representation, which is a flow graph with the following node types:

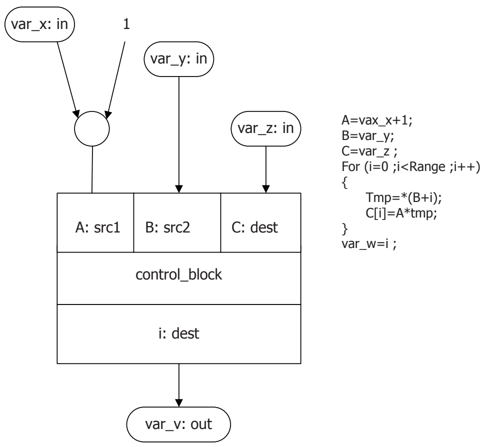
Basic computational operators: addition, multiplication, logical operations, comparison operations, etc. These are illustrated by circles in the figures in this paper.

Memory operators: load and store, with appropriate variants depending on whether the access is through a pointer, a singly subscripted array, or a doubly subscripted array. These are indicated by rectangles in our figures.

Control blocks: abstractions over embedded loops and conditionals. In our graph, control conditions are encapsulated in control blocks. These are indicated by adding named ports on rectangles in our figures.

Interface: input and output nodes, representing data flow into or out of the flow graph. These are illustrated by rounded rectangles in this paper.

An example is shown in Fig. 4.



```
A=vax_x+1;
B=var_y;
C=var_z ;
For (i=0 ;i<Range ;i++)
{
     Tmp=*(B+i);
     C[i]=A*tmp;
}
var_w=i ;
```

Fig. 4. source codes and its flow graph representation

2.4 Program Code

There are three main patterns in the pattern library, as described in Table 2. Formats like "for (i=0; i≤R; i++)" belong to COUNT PATTER. Reading and writing multiple SWP units must pay attention to subscripts- singly subscripted or doubly subscripted. An example is as follows.

```
for(i=B1;i<R1;i+=S1)
 for(j=B2;j<R2;j+=S2){
   ...=swp[i];/*SWP-read with single-subscripted */
   ...=swp[i][j] /*SWP-read with double-subscripted*/ }
```

Table 2. Pattern library

PATTERN	**DESCRIPTION**
COUNT PATTERN	In the named ports of control blocks, illustrating loops and control conditions
SWP-READ	In processing SWP computations, reading multiple SWP units
SWP-WRITE	In processing SWP computations, writing multiple SWP units

2.5 Dimension Notations

For edges in SWFG, there are notations on each dimension and processing attribute.

base::(Begin1:Range1:Step1:Lo/Hi,Begin2:Range2:Step2:Lo/Hi)

base: base address variable for sub-word area, which can be noted for pointer and array variables in source code.

Begin: the first element position of the loop control condition

Range: number of elements in a sub-word area

Step: stride in increasing or decreasing

Lo/Hi: processing format on sub-word area, low-part or high-part. Lo/Hi notation is specially important for loop-unrolling and assurance of alignment. An example is shown as follows in Table 3. We can pack these statements and process in parallel.

Table 3. Example of loop unrolling

for(i=1;i≤64; i=i+1) A[i+4] = A[i] + B[i];	After loop-unrolling 4 times	for (i=1; i≤64; i=i+4) { A[i+4] = A[i+0] + B[i+0]; A[i+5] = A[i+1] + B[i+1]; A[i+6] = A[i+2] + B[i+2]; A[i+7] = A[i+3] + B[i+3];}

3 Constructing a Compiling Framework

3.1 Algorithm for Pattern Matching

Pattern-recognition is applied to the initial program representation to identify patterns (as subgraphs) implementing memory accesses to SWP regions. Once identified, the subgraphs are replaced by the single high-level node, abstracting away the implementation details. Our pattern matching algorithm belongs to graph-based pattern recognition. The algorithm is illustrated in Table 4.

Table 4. Algorithm for pattern-recognition

STEP1	Find innermost for loop
	statement ::= expr {expr} for-statement | expr {expr} /*const-list is number of SWP units,4 or 8 preferably*/
STEP2	Creating list of SWP type variables
	array-type-variable ::= short | int | float identifier '['const-list']' /*Ivalue : :=identifier| *expr| lvalue| primary [expr] */
STEP3	Checking syntax of for statement

	for-statement ::=
	'for' ' (lvalue = constl;
	lvalue rel-op const2;
	(++ | - - | lvalue |
	lvalue (++ | - -) ') ' statement;
	/*Ivalue : := identifier| *expr| lvalue| primary [expr]*/

STEP4	Analyzing list of expressions
	expr ::= primary | (+ | - | ! | " | * | &) expr | expr ? expr: expr /*primary ::= identifier| constant | (expr) | primary [expr]*/
STEP5	Pattern-matching

	Matching the patterns described in Table 1 based on their structural characteristics.
	if ((identifier-in-expr == SWP-type-variable) and (patterns-in-the-Table 1))
	make-list-of-parallel statement-candidates;
	for-all-parallel statement?candidates_in-the-list-insert-macros_into-flowgraph;

3.2 Experimental Results and Conclusions

After exploiting subword parallelism, we have implemented code selectors in compiling framework using graph-based code selection techniques improved from Leupers's [12]. The experiments are composed of two parts. One is to test the performance of SWFG-based instruction selection. The other is to compare our SWFG-based approach with Leupers's code selection techniques based on intermediate dataflow graph. The result of the algorithm is list of parallel components which can be represented in flow-graph and executed in parallel by compiler. SWFG of loop unrolling in Table 3 is shown in Fig. 5.

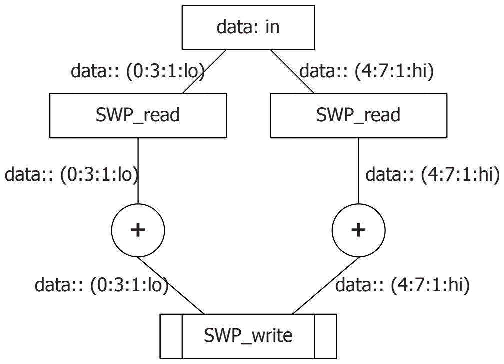


Fig. 5. A SWFG example for loop-unrolling

Framework Design. Our compiler framework is designed on sub-word TTA (Transport Triggered Architecture)[13] data path. It offers concurrent execution through a simple and flexible execution model. TTA depends much on compiler for the premise that the compiler is capable of effectively analyzing an application statically in order to extract available parallelism and target the available hardware resources.

Front-end of the compiling framework is built upon GCC 3.4.0, by planting its compiler gcc, assembler gas and linker gld, with BSD libraries libc and libm. By adding auto-vectorization in gcc, the vectorizable serial programs are generated ready for parallization. Back-end is built by utilizing compiling optimizations provided by SDTA, including software bypassing, dead result move elimination, operand sharing, operand sharing, socket sharing and scheduling freedom. It needs to read sequential codes from front-end and the architecture description from user. Then, subword parallel codes are realized

Prototype of the compiling framework is shown in Fig. 6. After identifying SWP components, subword instructions are selected out. Benefited from the

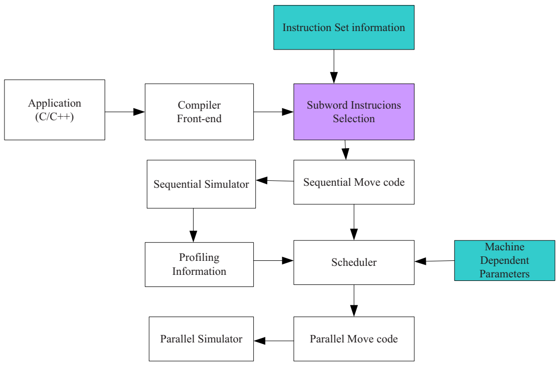


Fig. 6. Prototype of the compiling framework for Subword TTA Datapath

subword TTA datapath, they can be mapped to machine-dependent instructions easily. The generated codes afterwards can operate on corresponding registers and sub-registers, so that existing instruction scheduling and register allocation techniques can still be used.

Performance Test. The experiment is carried on with a group of standard media applications using MP4 Instruments sets. It is not the purpose of our experimentation to compare the code quality gain achieved by a certain SWP instruction set. Instead, we mainly wanted to show that code selection with SWP instructions frequently does result in higher code quality, and more important, that exploitation of SWP instructions in compilers is possible without compiler intrinsic and assembly libraries.

Table 5. Experimental results: Code selection with SWP instructions

source	data_type	Unroll	Without SWP	With SWP	CPU
vector add	short	1	8	4	0.7
IIR filter	short	0	22	22	5.1
convolution	short	1	8	8	0.9
FIR filter	short	1	15	9	0.9
N complex updates	short	1	20	20	4.7
image composition	short	1	14	7	3.2

The third column in Table 5 is unrolling factor. Unrolling is helpful in producing more parallelism and is necessary for SWP instructions exploitation. The fourth and fifth column give the number of generated machine instructions for the loop body separately with and without exploitation of subword parallelism. The sixth column gives the CPU seconds required when using SWP instructions. From the results we can conclude that, using vector-add can bring the most decrease of instruction. And using SWP instructions with loop unrolling once do not generate increase of instructions. For some applications, e.g. IIR and convolution, SWP instructions are not applicable. For FIR filter and image compositing, the code quality gains are significant, due to the more powerful SWP capabilities of MP4, e.g. special instructions for FIR computation. As shown for the vector add and FIR filter, the use of SIMD instructions for char data results in instruction count reduction, of 75.

Even though we use ILP for a part of code selection, the run time consumed by our approach is moderate if the graphs to be compiled are not too large. This is a consequence of the fact that most decisions concerning code selection are already made during the DFG covering phase, which only takes polynomial time in the DFG size. The largest example (FIR filter on char data), whose SWFG comprises 95 nodes, takes 26.5 CPU seconds. We believe that this is acceptable for embedded applications and systems-on-a-chip, where code quality is of much higher concern than compilation speed.

Comparative Test. Leupers has proposed an advanced dataflow graph-based code selection techniques. To compare his with our SWFG-based code selection, we compare the final SWP nodes pairs and generated program performance, as shown in Table 6. The lib functions are chosen from sources in section 3.2, with data type of 8 or 16 bit and loop unrolling of 1 or 3 times. From experiment above, it is obvious to find that DFG-based code selection can only recognize bld_move with general SWP instruction (ADD2). The reason is that DFG-based code selection techniques are unable to recognize sophisticate SWP instrucions and to operate on subregisters in different locations. Expect general SWP instructions such as MPY2 and SUB2, our SWFG-based techniques can select more kinds of SWP instrucions, e.g. DOTP2 in fir_sym, MIN2 in mad_16*16 and SADD2 in pix_sat_cn.

Table 6. Comparation of two code selection techniques

Lib Functions	unrolling	DFG-based code selection		SWFG-based code selection	
		node	cycle	node	cycle
blk_move	1	3	278	3	278
iir	1	0	2785	7	1184
fir_sym	1	0	1040	5	496
mad_16*16	3	0	12852	3	2685
pix_sat_cn	3	0	557	3	112

Conclusions and Future Work. In this paper, we attempt to experiment the effect of automatic code selection based on SWFG. An improved code selector using graph-based code selection techniques in our compiling framework is implemented. After testing the performance and comparing with that of DFG-based method, we can conclude that the SWFG-based approach can provide fairly good result and the generated SWFG can be used as an intermediate representation for the next step of compiler for SWP instruction selection and code generation. In the next step, more efforts will be focused on complex loop nests, alignment and loop unrolling factors.

Acknowledgement

Authors would like to thank all reviewers for their helpful comments.

References

1. Diefendorff, K., Dubey, P.K.: How multimedia workloads will change processor design. Computer 30(9), 43–45 (1997)
2. Sreraman, N., Govindarajan, R.: A vectorizing compiler for multimedia extensions. Int. J. Parallel Program 28(4), 363–400 (2000)
3. Krall, A., Lelait, S.: Compilation techniques for multimedia processors. Int. J. Parallel Program 28(4), 347–361 (2000)

4. Cheong, G., Lam, M.: An optimizer for multimedia instruction sets (1997)
5. Larsen, S., Amarasinghe, S.: Exploiting superword level parallelism with multimedia instruction sets. In: PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation, pp. 145–156. ACM Press, New York (2000)
6. Boekhold, M., Karkowski, I., Corporaal, H.: Transforming and parallelizing ANSI C programs using pattern recognition. In: Sloot, P.M.A., Hoekstra, A.G., Bubak, M., Hertzberger, B. (eds.) High-Performance Computing and Networking. LNCS, vol. 1593, p. 673. Springer, Heidelberg (1999)
7. Martino, B.D., Iannello, G.: Pap recognizer: A tool for automatic recognition of parallelizable patterns. In: WPC '96: Proceedings of the 4th International Workshop on Program Comprehension (WPC '96), Washington, DC, USA, p. 164. IEEE Computer Society Press, Los Alamitos (1996)
8. Martino, B.D., Kesler, C.W.: Two program comprehension tools for automatic parallelization. IEEE Concurrency 8(1), 37–47 (2000)
9. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. 9(3), 319–349 (1987)
10. Murthy, P., Lee, E.: Multidimensional synchronous dataflow. IEEE Transactions on Signal Processing 50(7), 2064+ (2002)
11. Bhattacharyya, S.S., Murthy, P.K., Lee, E.A.: Synthesis of embedded software from synchronous dataflow specifications. J. VLSI Signal Process. Syst. 21(2), 151–166 (1999)
12. Leupers, R.: Code selection for media processors with simd instructions. In: DATE '00: Proceedings of the conference on Design, automation and test in Europe, pp. 4–8. ACM Press, New York (2000)
13. Corporaal, H.: Transport Triggered Architectures. Design and Evaluation. PhD thesis (1995)

A Stream System-on-Chip Architecture for High Speed Target Recognition Based on Biologic Vision

Nan Wu, Qianming Yang, Mei Wen, Yi He, Changqing Xun, and Chunyuan Zhang

National Laboratory for Parallel & Distributed Processing Chang Sha,
Hu Nan, P. R. of China, 410073
nanwu@nudt.edu.cn

Abstract. For target recognition based on biologic vision, an application-specific stream SOC: MASA-MI is described in this paper. MASA-MI consists of several heterogeneous cores, and a stream accelerator core is used to accelerate matching image which consumes the most time in target recognition. We implemented it on Altera EP2S60 FPGA. Result shows the 166MHz MASA-MI provides a peak performance of 585 fps. MASA-MI's performance is an order of magnitude higher than those of today's DSPs such as the Texas Instruments TMS320DM642 (600MHz). On the other hand, the cost is far less than special purpose processors.

1 Introduction

Automatic target recognition is generally applied in image recognition of vehicle, UAV, and missile seeker. Since the image acquired from different angles and positions, great distortion of the same target exists in different images. Target recognition algorithm based on biologic vision can accurately recognize the target [1][2]. However the algorithm requires tens to hundreds of billions of computations per second. Along with the demand for higher speed carrier and larger image resolution, the computing requirement is increasing. To achieve these computation rates, special-purpose architectures tailored to the application are used [3][4][10]. Such processors require significant design effort and are thus expensive. Today's DSPs such as the Texas Instruments TMS320DM642 [5] are relative cheap, but are still one to two orders of magnitude worse than special-purpose processors [6]. For example, the 600MHz TMS320DM642 DSP provides a peak performance of about 50fps that is much lower than the requirement of high speed carrier. So there is a large gap between the cost and performance of special-purpose and DSP on target recognition.

FPGA (Field Programmable Gate Array) is a high-performance and economic method of IC implementation because of its short development period and flexibility. The logic-capacity of a single state-of-the-art FPGA chip such as Altera StratixII FPGA is up to tens million gates [7]. There are abundant DSP and RAM resources and advanced bus structure in modem FPGA.

For high speed target recognition based on biologic vision, this paper designs and implements MASA-MI stream processor on a StratixII EP2S60 FPGA chip. It is an application-specific stream SoC, which consists of heterogeneous cores. A stream accelerator is used to accelerate matching image, which consumes the most time in target recognition. Result shows that the 166MHz MASA-MI provides a peak

© Springer-Verlag Berlin Heidelberg 2007

performance of 585fps. MASA-MI's performance is an order of magnitude higher than those of today's DSPs such as the Texas Instruments TMS320DM642 (600MHz). On the other hand, the cost is far less than special purpose processors.

This paper mainly discuss the target recognition algorithm based on biologic vision and its hardware implementation, the other related contents such as filter, quantification and image segmentation are not discussed in this paper. The detail discussion of them can be found in [15][16].The remainder of this paper is organized as follows. Section 2 indicates the theory of target recognition based on biologic vision and numerical computation method. Section 3 describes MASA-MI architecture. Section 4 shows the implementation result. The last section summarizes the conclusions drawn in this paper.

2 Theroy and Algorithmn

2.1 Theory of Target Recognition Based on Biologic Vision

First step of *Target Recognition algorithm based on Biologic Vision* (TRBV) is using *hypercolumns vector* to represent image [1]. It chooses 2D Gabor function as reception field function of unicellular and tune orientation of Gabor function to determine a group of orientations. Then an n-Dimension hypercolumns vector can be derived by doing inner product of partial image and n Gabor templates respectively [8][9]. Gabor function is a *sin* or *cos* function tuned by Gaussian function, and the mathematic form is as follows:

$$g^i(x,y)=\frac{1}{2\pi\sigma_x\sigma_y}e^{-\frac{x_1^2}{\sigma_x}+-\frac{y_1^2}{\sigma_y}}e^{i\varpi x_1} \tag{1}$$

where

$$\begin{cases} x_1 = x\cos(\theta)+y\sin(\theta) \\ y_1 = -x\sin(\theta)+y\cos(\theta) \end{cases} \tag{2}$$

σ_x and σ_y are dimensional parameters. ϖ is frequency parameter. θ determines the orientation of Gabor function. $g^i(x,y)$ is a plural form, which including odd part (a *sin* function tuned by Gaussian function) and even part (a *cos* function tuned by Gaussian function). Figure 1 shows a group of Gabor templates.

After using hypercolumns vector to represent image, image matching can be replaced by hypercolumns vector matching.

We define $f(x,y)$ and $f^{'}(x,y)$ are two images which acquired from different angles. The projective transformation relation is as follows:

$$f^{'}(x,y)= A(\rho)\hbar\, f(x,y) \tag{3}$$

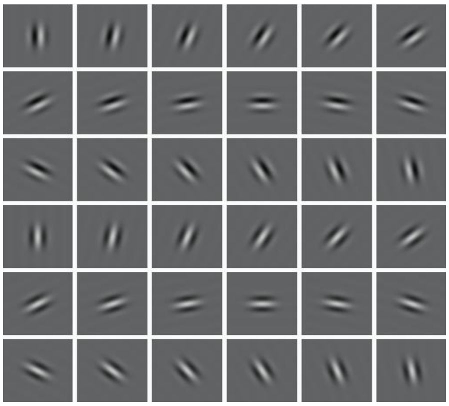


Fig. 1. A group of Gabor Templates

$A(\rho)$ denotes projective transformation operator. ρ denotes 8 projective transformation parameters.

The hypercolumns vector of $f^{'}(x,y)$ image is $\left\{\left\langle g^{i} \quad f^{'}\right\rangle\right\} i=1,2,3....n$. One element of vector is as follows:

$$\left\langle g^{i} \quad f^{'}(x,y)\right\rangle=\left\langle g^{i} \quad A(\rho)\hbar f(x,y)\right\rangle=\left\langle A^{*}(\rho)\hbar g^{i} \quad f(x,y)\right\rangle \tag{4}$$

$A^{*}(\rho)$ is dual operator of $A(\rho)$. Formula 4 denotes hypercolumns vector can be kept unchanged through transforming reception field function to compensate image transformation. We define Euclidean distance between hypercolumns vectors as follows:

$$D=\sqrt{\sum_{i=1}^{n}(\left\langle g^{i} \quad f^{'}(x,y)\right\rangle-\left\langle A^{*}(\rho)g^{i} \quad f(x,y)\right\rangle)^{2}} \tag{5}$$

If $f(x,y)$ and $f^{'}(x,y)$ represent the same target, we can minimize D to get transformation parameter and complete image matching.

$$A^{*}(\rho)=\arg\min(D) \tag{6}$$

For the same target, suppose the former frame of image is denoted by $\boldsymbol{f_1(u,v)}$, the latter frame of image is denoted by $\boldsymbol{f_2(x,y)}$, a affine transformation between targets in two frames exists as follows:

$$\begin{cases} \boldsymbol{x=r_0\times u+r_1\times v+r_4} \\ \boldsymbol{y=r_2\times u+r_3\times v+r_5} \end{cases} \tag{7}$$

When the expression which represents relation between hypercolumns vector and 6 *affine parameters* (r0~r5) is derived, 6 affine parameters can be calculated. In this paper, 6 parameters of $A^*(\rho)$, when D is minimum, are obtained by mini-least square solution for overdetermined equation. As result, if we get $A^*(\rho)$ the new coordinate of the target can be calculated out, which is used to complete target recognition and tracking.

2.2 Numerical Recipe

The numerical recipe of TRBV is shown as Figure 2.Target image windows of reference image and live images to track are both supposed to be 6400 pixels (80*80). On the other hand, 36 Gabor templates and 36×6 affine templates, which conclude 1612800 elements, are pre-stored in a matrix.

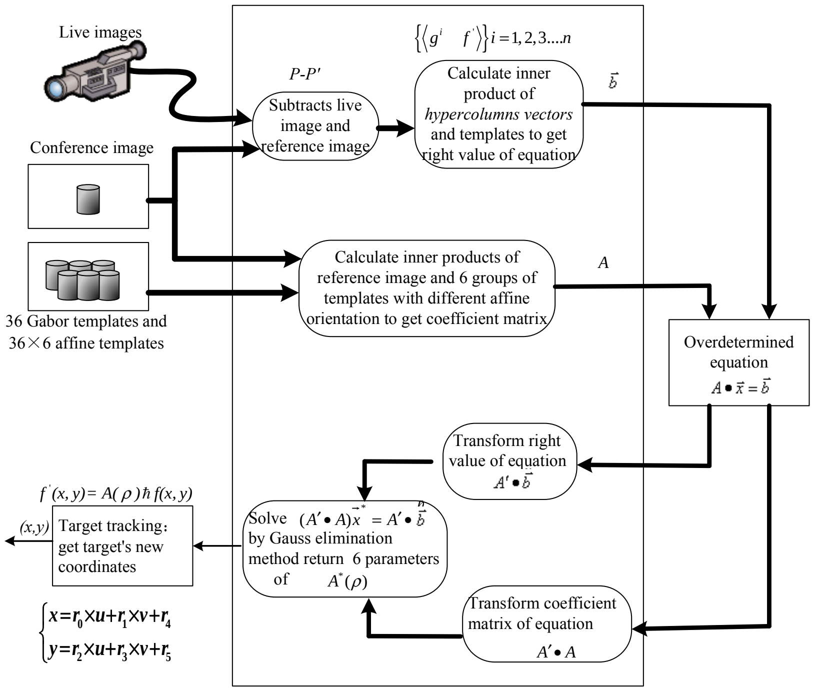

Live images
Conference image
36 Gabor templates and 36×6 affine templates
$\{\langle g^i \; f' \rangle\} i=1,2,3....n$
P-P'
Subtracts live image and reference image
Calculate inner product of *hypercolumns vectors* and templates to get right value of equation
$\vec{b}$
Calculate inner products of reference image and 6 groups of templates with different affine orientation to get coefficient matrix
A
Overdetermined equation $A \bullet \vec{x} = \vec{b}$
Transform right value of equation $A' \bullet \vec{b}$
Solve $(A' \bullet A)\vec{x}^* = A' \bullet \vec{b}$ by Gauss elimination method return 6 parameters of $A^*(\rho)$
Transform coefficient matrix of equation $A' \bullet A$
$f'(x,y)= A(\rho) f(x,y)$
(x,y)
Target tracking: get target's new coordinates
$x=r_0 \times u + r_1 \times v + r_4$
$y=r_2 \times u + r_3 \times v + r_5$

Fig. 2. Flow chart of the numerical recipe of TRBV algorithm

First subtracts live image from reference image, then performs inner product with all Gabor templates to obtain a vector (vector length is 36). In fact, this vector is the deference of hypercolumns vectors of live image and reference image. Let the group of vectors to be right values of equation. And let a 36×6 matrix to be a coefficient matrix, which is obtained by performing inner products of reference image and 6 groups of templates with different affine orientations. The major count of computation is about

1612800 multiplications and 12419200 additions. Then $A^*(\rho)$ can be get by resolve equation $A \bullet \vec{x} = \vec{b}$ (n=36):

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 + a_{15}x_5 + a_{16}x_6 = b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 + a_{25}x_5 + a_{26}x_6 = b_2 \\ \bullet\bullet\bullet \\ a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + a_{n4}x_4 + a_{n5}x_5 + a_{n6}x_6 = b_n \end{cases} \tag{8}$$

A and $\vec{b}$ are respectively the coefficient matrix and right value mentioned before. This is an overdetermined equation, which we can get its least squares resolution by transform the equation into following form:

$$(A' \bullet A)\vec{x}^{\,*} = A' \bullet \vec{b} \tag{9}$$

The paper use Gauss elimination method [17] to resolve this equation, the algorithm returns 6 parameters of $A^*(\rho)$.

3 MASA-MI Stream SoC Architecture

In terms of our numerical recipe discussed in section 2, we present a Stream System-on-Chip Architecture MASA-MI for TRBV algorithm. The overview of the architecture is shown in Figure 3. The architecture is consists of two scalar processors,

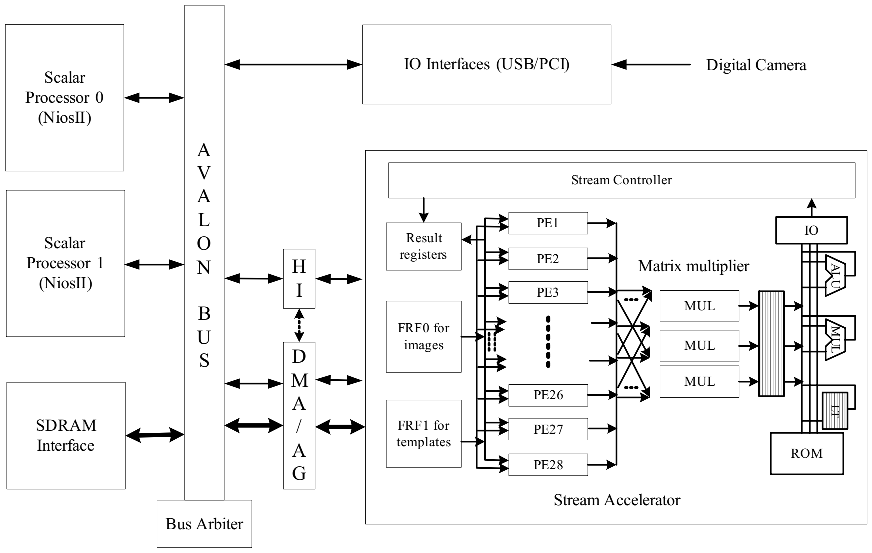


Fig. 3. The stream SoC architecture for target recognition based on biologic vision

a stream accelerator and several peripheral cores, which of all are connected together into a SoC system by Avalon bus [13]. It is designed for 80*80 target widow, and can scale to any other widow sizes easily. Finally MASA-MI is implemented on Altera EP2S60 FPGA, so it is special optimized for the features of FPGA device.

Scalar processor 0 runs programs for controlling entire SoC. Firstly, when the system starts up, scalar processor 0 initializes each peripheral and loads template matrix and reference image from flash to SDRAM. Then, it sends commands to start up stream accelerator and scalar processor 1. After that, scalar processor 1 continually incepts the image from digital camera to SDRAM and runs image segmentation algorithm to approximately localize the place of target. Meanwhile, in terms of pre-arranged task schedule, stream accelerator loads images and templates from SDRAM and calculates the coordinate transformation operator $A^{*}(\rho)$ by TRBV algorithm. This will occupy more than 95% of the amount of computation of total target recognition task. At last, scalar processor 0 gets the operator from stream accelerator to calculate the new coordinate of target. To accelerate processing of TRBV algorithm, the accelerator is designed base on stream processing [11], which is optimized for image stream with extremely high bandwidth hierarchy and large scale *Processing Element* (PE) array.

3.1 Stream Buffer and Memory

The most crucial part of the entire architecture design process of the stream accelerator is the design of memory system, because TRBV algorithm demands for both capacity and bandwidth of storage system. Since image processing has very typical stream characters that the pixel data pass filter once and will not be reused, cache is not suitable for [12]. We designed a novel stream memory system for TRBV algorithm as shown in Figure 4, which contains three memory hierarchies: off-chip SDRAM, on-chip *frame register file* (FRF) and *stream buffer* (SB).

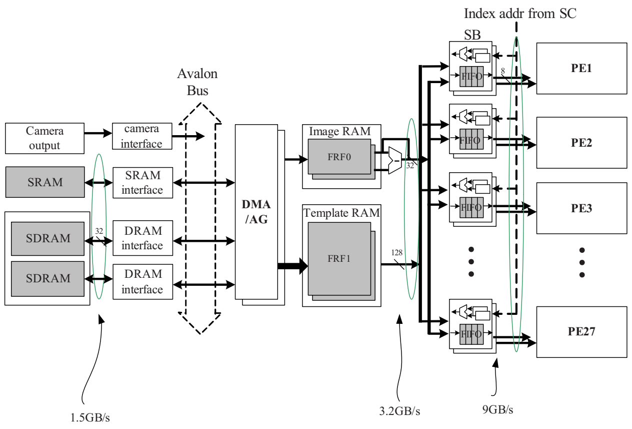


Fig. 4. Stream memory hierarchy[1]

[1] The on-chip bandwidth corresponds to an operating frequency of 166 MHz.

Reference image and templates both reside in two 4M×32 bit PC100 SDRAMs on backboard, and live image resides in 1MB SRAM on backboard. Two SDRAMs with SRAM together provide a 1.5GB/s peek bandwidth at 166 MHz. It's hard to continue increasing bandwidth, which is limited to the layout of PCB board and the number of pins of FPGA. Live image stream is real-time inputting through USB or PCI port from camera. The memory system supports data stream transferring between the FRF and off-chip RAM. There are two stream channels designed across Avalon bus, one is 8bit wide for live image stream and another is 64bit wide for both reference image stream and Gabor templates. MASA-MI makes all memory references using *Stream Load/Store* instructions executed in DMA/address generate (AG) unit that transfer an entire stream between memory and the FRF. In our designs, stream elements of image are sequentially stored in memory for that image's pixels must be contiguous. Meanwhile, the memory system also supports stride and indexed addressing. In addition, because memory-reference granularity is stream, we can optimize the memory system for stream throughput rather than the reference latency of individual stream elements. For instance, access stream form SDRAM in long burst mode is much faster than random-access mode. Moreover, these references and computation can be easily overlapped. For large data size, the memory system can load streams in double buffers mode to efficiently hidden the load latency.

When data streams have been loaded on chip, all of them are stored in FRF. There are two FRFs corresponding to the two streams that are 128KB and 64KB respectively. Both FRFs work in software managed double buffer mode. A double-buffered stream access involves cycling portions of a large stream through two halves of a smaller buffer in the FRF.

As shown in figure 4, each PE's data access to FPF go through a *stream buffer* (SB), which is consist of a FIFO and a address index logic. PE makes requests to the stream buffers to read elements from a stream. SBs in turn make requests to access the location in the FRF storage where that stream resides. These requests are handled by a 28:1 arbiter in FRF controller. One SB is granted access per cycle and read 16 words into FIFO (The size of FRF's outport is 128bit, each fifo has 32 entries). Finally, the PE can read or write data from their associated stream buffer. Stream buffers prefetch 16 words data one time from the FRF, while PEs read data from stream buffers at lower continuance bandwidth but higher instantaneous bandwidth. The bandwidth between 28 SBs and 2 FRFs is 3.2GB/s, moreover total 28 SBs can provides PE array a instantaneous peak bandwidth of 9GB/s.

In the final analysis, because of appropriate memory hierarchy and stream buffers, PEs are filled in at a much higher peak bandwidth than that off-chip RAM can provides.

3.2 PE Array

The structure of PE array is shown as Figure 5. PE array is the major computing engine of the MASA-MI. It is consists of two part: inner product generator and matrix multiplier. In inner product generator, multiplex parallel pipelines are designed which takes only 144000 cycles to calculate A and $\vec{b}$ in formula (9).

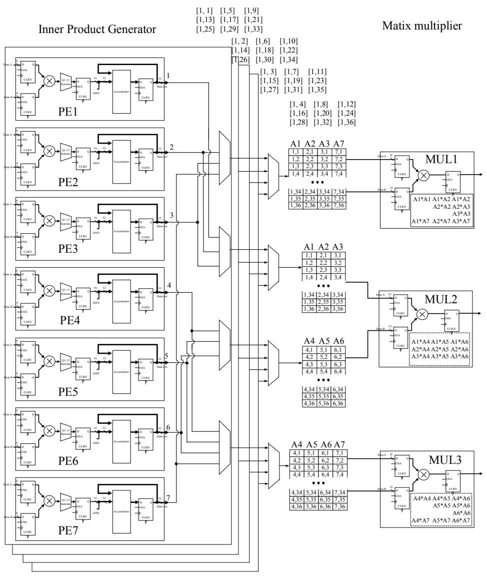


Fig. 5. The structure of PE array

Since there are abundant DSP block resources in the FPGA, we configure 28 PEs to perform inner product in parallelism. As shown in Figure 5, PE1, PE7, PE14, PE21 respectively calculate 9 rows of deference of *hypercolumns vectors*(A7), PE1~PE6, PE8~PE13, PE15~PE28 respectively calculate 9 rows of coefficient matrix ([A1:A6]). Synthesized result shows that PE can work over 200MHz. In addition, MASA-MI architecture is easy to scale to more PEs if needed.

Matrix multiplier performs matrix transpose and multiply that calculate $A' \bullet A$ and $A' \bullet \vec{b}$ in formula (9). Since the amount of computation of inner product is far more than matrix multiply, matrix multiply is not the main workload. The parallel matrix multiplier structure is shown in Figure 5, it outputs a 6x6 coefficient matrix and right value vector of the equation.

3.3 Cluster for Gauss Elimination Method

The amount of computation solving equation is the least one of entire image matching, but the procedure of Gauss elimination method includes lots of branch, floating-point multiply and floating-point division operations. If this part of computation is done on scalar core, it will become bottleneck of the whole target recognition according to Amdahl's law. So we design a small cluster architecture [14], including a floating-point multiplier, a floating-point adder, a lookup table and register file. Former two units are implemented by Altera megafunction units [7]. Intra-cluster modules are interconnected by a cross switch. Microcodes of the Gauss elimination method is stored in ROM, which are issued to and executed on cluster. The result is transferred to scalar processor through *host interface* (HI).

In addition, two Altera NiosII 32b embedded IP cores [7] with 4KB instruction cache and 2KB data cache are applied as the host processors of MASA-MI. NiosII can get 110MIPS performance at 100MHz.

As an integrated SoC chip, MASA-MI integrates a lot of peripheral equipment interface, which can interconnect with devices on system board directly. The design use Avalon bus as on-chip system bus, Avalon masters and slaves interact with each other based on a technique called slave-side arbitration. Multiple masters such as stream accelerator and scalar processors can perform bus transactions simultaneously, as long as they do not access the same slave during the same bus cycle.

4 Implementation Results and Performance

4.1 Implementation Result

MASA-MI has been implemented on Altera StratixII EP2S60 FPGA. Figure 6 shows a filter view of MASA-MI with major modules presented in section 3 highlighted. The architecture was modeled using Verilog as hardware description language and synthesized, placed and routed using Altera Quartus II 5.1. As shown in Figure 6, Nios cores and steam accelerator occupy the most regions of floorplane. Especially, stream accelerator used all of the MRAM resources and a large number of DSP blocks of FGPA. In detail, Table 1 summarizes the resource utilization by main modules.

Statistics show that MASA-MI occupies 72% on-chip RAM and use 84 DSPs. These resources are all standard megafunction units integrated in today’s FPGA, which are abundant and works at high clock speed. In addition, multiple clock domains are used in MASA-M1 implementation by *Enhanced PLL* [7] on FPGA. At last, the critical path

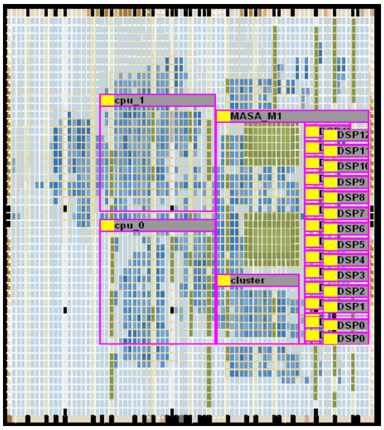


Fig. 6. Fitter view of MASA-MI

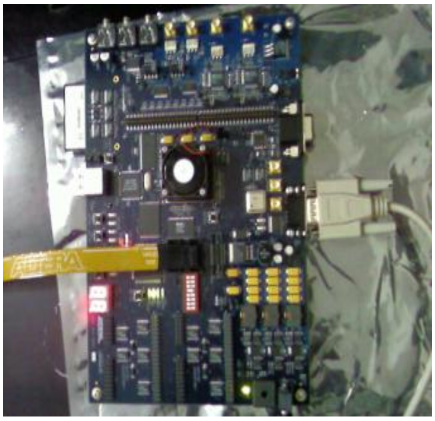


Fig. 7. Development board of MASA-MI

Table 1. FPGA resource utilization by main modules

	#ALUT	#ALM	#LCCOM	#LCFF	# RAM	#DSP
PE unit	16	8	16	16	0	2
matrix mul unit	32	16	32	32	0	4
matrix ram	0	0	0	0	24576	0
total PE aaray	556	280	544	544	24576	68
cluster	1456	831	1177	1073	46400	0
FRF0	0	0	0	0	524288	0
FRF1	0	0	0	0	1048576	0
single SB	32	36	44	72	256	0
All SBs	1728	1944	2464	3888	13824	0
Stream Controlle	159	135	158	144	0	0
HI	135	104	70	121	512	0
Memory controller	551	225	157	457	0	0
CPU0(niosII)	1664	1010	1069	1199	40336	8
CPU1(niosII)	2675	1631	1949	1757	39808	8
others	868	344	278	635	105682	0

Table 2. Architecture parameters of MASA-M1 SoC

Stream accelerator clock rate	166MHz
Arithmetic bandwidth of stream accelerator	8bit: 4.5 GOPS 32bit: 0.5 GOPS float: 0.32GOPS
Peak bandwidth of storage hierarchy	off-chip memory: 1.5GB/S FRF: 3.2GB/s SB: 9GB/S
Capacity of storage hierarchy	SDRAM : 32MB SRAM: 1MB FRF: 192KB
Scalar Processor clock rate	100MHz
Peak performance of Scalar processor	110MIPS
Power estimate (chip only)	775 mw
Power estimate (chip on board)	1422 mw

delay of the implementation of stream accelerator is SDRAM interface, which determines 166MHz operation frequency of stream accelerator. In conclusion, some important architecture parameters of MASA-M1 SOC are discussed in Table 2.

4.2 Performance Analysis

Group of experiments are run on development board of EP2s60 shown in Figure 7. Unfortunately we do not find a suitable high speed camera (over 500fps) for the moment, so a section of pre-shot video or sequential photos loading from a CF card is used to simulate camera inputting. Anyhow this will not influence the result of test which is major evaluating the performance of chip itself. Several practical target recognition programs were run on it, Figure 8 shows a result on real image, witch are two air photos. We randomly select some targets in the left image, and the right image shows matching results.

For comparison, we take a Pentium 4 PC[2] as a reference of general purpose processor and TI DMA642[3] (600MHz) as a reference of DSP with special instruction for image processing. All of them run the same algorithm. Table 3 shows the running time of target recognition (one matching target) on PC, TMS320DM642 and MASA-MI respectively. Because TRBV algorithm is the major computation of the application, the execution difference of TRBV algorithm in different processor is the key performance gap. Stream accelerator achieves 38.6 speedup compared to TMS320DM642 (C code). As a result, MASA-MI achieves much higher performance that is over 500fps higher than general processor or DSP. The further improvement of the speed is limited to other parts of the application such as scalar processing or IO.

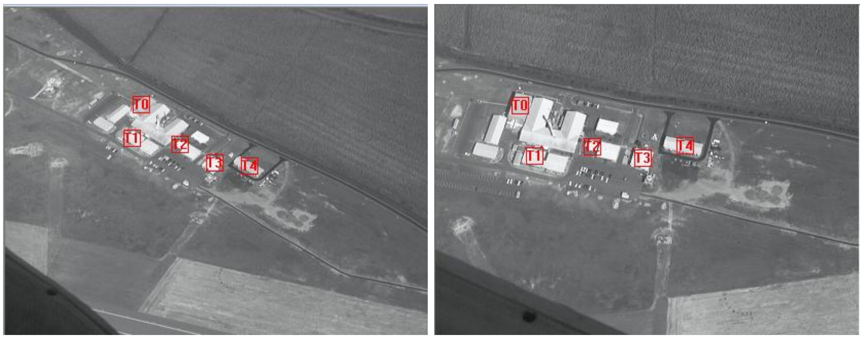


Fig. 8. Result on real image

Table 3. Comparison of some processor for real image

	PC	TMS320DM642 (600MHz)		MASA-MI
		C code	Assembly optimizer	
Time of TRBV	~80ms	36.40ms	18.65ms	0.94ms
Speed up	0.4	1	1.9	38.6
Time of others	~3ms	0.63ms	0.58	0.77ms
Speed up	0.2	1	1.1	0.8
Total time	~83ms	37.04ms	19.23ms	1.71ms
Total Speedup	0.4	1	1.8	21.2
Frame rate	12 fps	27 fps	52 fps	585 fps

(80*80 target windows, 420*320 image size)

5 Conclusions and Future Work

In this paper, we propose an application-specific stream SoC architecture: MASA-MI for high speed target recognition based on biologic vision. The MASA-MI consists of two scalar processors, a stream accelerator and several peripheral cores, which of all are connected together into a SoC system by Avalon bus. Scalar processor runs programs for controlling entire SoC, while Stream accelerator is a key special coprocessor. We design PE array and cluster for intensive computation, while we utilize stream memory hierarchy to minimize the memory references between the computations and exploit

[2] P4 2GHz, 1GB DDR400.

[3] All the result of DM642 in this paper are compiled by CCS2 in the –o3 flag, and run on a real chip.

locality. Results demonstrate that the performance of MASA-MI achieves 585fps for target recognition at 166MHz, which is an order of magnitude higher than those of today's DSP such as the TMS320DM642 (600MHz). Based on it, people can pursue higher speed carrier and larger image resolution.

This work verified stream processing's potential for target recognition. Combining stream processing and TRVB algorithm, we will try to improve MASA-MI's performance to 1000fps in future work.

Acknowledgements. This research was supported by National Natural Science Foundation of China, NSFC No. 60673148 and Specialized Research Fund for the Doctor Program of Higher Education, SRFDP No. 20069998025. We also thank Zhu Xianwei and Yu Qifeng for their knowledge of target recognition.

References

1. Xianwei, Z., Qifeng, Y.: Ground Target Recognition and Tracking based on biologic vision. In: Optical and Electronical Technology Conference of China, October 2006, Chengdu (2006)
2. Tsao, T., Wen, Z.: Image-based target tracking through rapid sensor orientation change. Optical Engineering 41(3), 697–703 (2002)
3. NEC inc.: Automotive Image Recognition Processor IMAPCAR (2006), www.nec.co.jp
4. Renesas Technology Inc.: SH7774 SoC with On-Chip Image Recognition Processing Function for Car Navigation (2006), www. Renesas.com
5. TI inc.: TMS320DM642 video/imaging Fixed-point digital Signal Processor (2007), www.ti.com
6. Mehendale, M.: Challenges in the design of embedded real-time DSP SoCs. In: Proceedings of 17th International Conference on VLSI Design, 2004, pp. 507–511 (2004)
7. ALTERA Inc.: Stratix II Device Handbook (2005), http://www.altera.com
8. He, C., Zheng, Y.F., Stanley, C: Ahalt. Object Tracking Using the Gabor Wavelet Transform and the Golden Section Algorithm. IEEE transactions on multimedia 4(4), 528–538 (2002)
9. Wu, X., Bhanu, B.: Gabor Wavelet Representation for 3-D Object Recognition. IEEE transactions on image processing 6(1), 47–57 (1997)
10. Khailany, B.: The VLSI Implementation and Evaluation of Area and Energy Efficient Streaming Media Processors. Ph.D. Thesis, Dept. of Electrical Engineering, Stanford University (June 2003)
11. Khailany, B., Dally, W.J., Kapasi, U.J., Mattson, P., et al.: Imagine: media processing with streams. IEEE micro (2001)
12. Rixner, S., Dally, W.J., et al.: A bandwidth-efficient architecture for media processing, proceedings. In: 31st annual ACM/IEEE international symposium on microarchitecture (1998)
13. ALTERA Inc.: Quartus II Version 5.1 Handbook (2005), http://www.altera.com
14. Mei, W.: Key techniques research of stream architecture, PHD thesis, National university of defense technology, China (September 2006)
15. Haoying, W.: An approach to complex target segmentation and reorganization. Journal of Beijing institute of Technology 20(2), 224–228 (2000)
16. Dongping, M.: Research on information extraction and target recognition from high resolution remote sensing image. In: Science of Surveying and Mapping vol. 3 (2005)
17. Yin, Y.: Advanced Engineering Mathematics, 3rd edn. pp. 263–265. Hua zhong University of Science and Technology Press (2001)

FPGA-Accelerated Active Shape Model for Real-Time People Tracking

Yong Dou and Jinbo Xu

Department of Computer Science,
National University of Defence Technology,
Changsha, P.R. China, 410073
yongdou@nudt.edu.cn
xujinbo@nudt.edu.cn

Abstract. Active Shape Model has been proven to be one of the most popular methods for recognizing non-rigid objects, which requires huge computation power for real time people tracking. After analyzing the parallel characteristics of the algorithm, we propose a deep pipelined structure for accelerating the Active Shape Model algorithm. The computing engine is organized into a deep pipeline network composing of multiple floating-point arithmetic units, including adders, multipliers, dividers and SQRT etc. A linear multiplication-accumulation (MAC) unit is designed to lower the complexity of the computing resources while keeping high pipeline throughput. In the optimization of the memory efficiency for loading random data in large images during the step of local search, we propose an on-chip buffer scheme to eliminate random accesses to off-chip memory. Experimental results show that our FPGA implementation achieves over 15 times of speedup compared with the sequentially-implemented software solution in Pentium 4 computer.

Keywords: FPGA, Active Shape Model, People Tracking.

1 Introduction

Tracking people and recognizing their actions in video sequences become increasingly important in many practical applications, such as human computer interaction, motion capture for animation, video surveillance, and etc. The challenge is the forms of tracked objects keep changing between consecutive frames, denoted as non-rigid objects. Over the last two decades, various methods have been proposed to deal with this task. Kass et al. proposed Active Contour Models (ACMs) in 1987 [1]. Wiskott et al. proposed a Gabor feature based Elastic Bunch Graph Matching (EBGM) method in 1997 [2]. In 1995, Cootes and Taylor proposed Active Shape Model (ASM) [3], which has been proven to be one of the most popular methods in this field. ASM has been widely used in areas, such as video surveillance (e.g. [4]), facial recognition (e.g. [5]), medical imaging (e.g. [6]), and so on. And a lot of improvements have been proposed upon traditional ASM (e.g. [7]).

© Springer-Verlag Berlin Heidelberg 2007

ASM based on statistical models belongs to time-critical applications. The advantage of ASM is that it allows for considerable variability but still specific to the class of objects or structure they intend to represent. However, since the ASM algorithm is an iterative approach, it requires huge computation performance for real time processing. Traditional embedded systems based on general-purposed processors can not afford both of the computation and power dissipation requirements. Adam Baumberg developed a sequentially implemented people tracking system [8] on the general-purposed processor, which is based on ASM. His system improves the tracking speed by decreasing the number of sample points, which will influence the tracking accuracy due to the information losses. Therefore, making a fast and accurate hardware implementation for people tracking applications is the focus of this paper.

In this paper, we present a real-time ASM-based people tracking system in a reconfigurable hardware. The hardware implementation in our work is organized into a deep pipeline network composing of many floating-point arithmetic operations. The characteristics of the algorithm are fully analyzed to exploit spatial parallelism and temporal parallelism. In order to improve the memory efficiency for loading random data in large images, we propose a hierarchical memory scheme buffering part of the whole image in on-chip RAM block to eliminate random accesses to off-chip memory, which takes full advantage of high speed of on-chip memory and large capacity of off-chip memory. Data reusability is exploited to improve the effective I/O bandwidth between the off-chip memory and FPGA. Experimental results show that by using this hierarchical memory scheme, speedups of at least 1.87 and up to 7.91 are achieved when compared with the solution without on-chip RAM. By using our own designed high-performance floating-point units, the clock speed of the system reaches 90MHz. With different size and sample points, the results show that speedups of at least 4.29 and up to 15.40 are achieved while comparing our FPGA design with sequentially-implemented software solution in Pentium 4.

The remainder of this paper is organized as follows. In Section 2, an overview of people tracking applications based on ASM is described. Section 3 analyzes the characteristics of the ASM algorithm. The design and implementation of the system are presented in detail in Section 4. Section 5 introduces the performance evaluation and experimental results. We finish with conclusion in Section 6.

2 Overview of ASM

In this section we briefly sketch the ASM framework, a more detailed description of ASM can be found in [3].

In standard ASM, given a training set of S instances of the same object class, each of them is represented by a set of landmark points $\{(x_i, y_i)\}_{i=1}^{n}$. They can be written as a $2n$-element vector $x_s = (x_1^{(s)}, y_1^{(s)}, \cdots, x_n^{(s)}, y_n^{(s)})^T$. Sample mean and covariance matrices of them are:

$$\overline{x} = \frac{1}{S}\sum_{s=1}^{S} x_s, C = \frac{1}{S-1}\sum_{s=1}^{S}(x_s - \overline{x})(x_s - \overline{x})^T \ . \tag{1}$$

Let $P = (P_1|P_2|\cdots|P_D)$ denote the matrix whose columns are the D eigenvectors corresponding to the D largest eigenvalues $\lambda_1, \cdots, \lambda_D$ of C. Any example of the training set, x_s, can be approximated by $x_s \approx \overline{x} + Pb_s$, where b_s is the D dimensional model parameter vector, computed by $b_s = P^T(x_s - \overline{x})$. The number D of eigenvectors to retain is usually calculated as the smallest D that satisfies $f_v \sum_{i=1}^{2n} \lambda_n \leqslant \sum_{d=1}^{D} \lambda_d$, where f_v is the proportion of the total variance of the data which can be explained, usually ranging between 0.900 and 0.995.

For a given shape x, iterative matching is performed to approximate it as closely as possible. Firstly, it is usually initialized with the mean shape(it means each element of b is zero) and translation (t_x, t_y), rotation θ and scale s parameters reasonably close to their 'true' values. Then, these parameters are applied upon the mean shape to generate the model point positions:

$$X = T_{t_x,t_y,s,\theta}(\overline{x} + Pb) \ . \tag{2}$$

where the function $T_{t_x,t_y,s,\theta}$ performs a rotation by θ, a scaling by s and a translation by (t_x, t_y). For instance, if applied to a single point (x, y),

$$T_{t_x,t_y,s,\theta} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} t_x \\ t_y \end{pmatrix} + \begin{pmatrix} s\cos\theta & -s\sin\theta \\ s\sin\theta & s\cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} . \tag{3}$$

Next, local search is processed around each point X_i in X to find the best new position for it. It is usual to use fixed-length, one-dimensional profiles orthogonal to the contour. For each example point, a fixed number of pixels on and to either side of the contour are sampled. The pixel which has the strongest edge response will be considered as the best match.

In order for more flexibility, the local search procedure can use self-adaptive search scale instead of using fixed-length profiles. The search scale for each sample point can be automatically controlled using the Kalman filter mechanism(as demonstrated by Blake et al[9]). The search window size ρ_i at the i'th sample point (x_i, y_i) is related to the positional variance $V(x_i)$ and $V(y_i)$ at the estimated contour point given by

$$V(x_i) = [(QGP)P_k(QGP)^T]_{2i,2i} + V(o_x) = \sum_{j=0}^{l-1} ((QGP)_{2i,j})^2 \sigma_j + V(o_x). \tag{4}$$

where Q denotes the alignment matrix, G is a $2n \times 2N$ sparse matrix mapping the $2N$ control points to regularly spaced $2n$ sample points, P_k is the covariance matrix of shape parameters, and $V(o_x)$ denotes the variance of the estimated origin of the shape $\hat{o}_x$(and a similar equation is obtained for y_i). The search scale ρ_i along the normal line n_i(represented as $((n_i)_x, (n_i)_y)$) is given by

$$\rho_i = 2\sqrt{\frac{V(x_i)V(y_i)}{(n_i)_x^2 V(y_i) + (n_i)_y^2 V(x_i)}}. \tag{5}$$

Thereinto, the normal vector $n = ((n_1)_x, (n_1)_y, \cdots, (n_n)_x, (n_n)_y)^T$ is calculated according to $n = T_{t_x,t_y,s,\theta}(\overline{n} + Nb)$(where $\overline{n}$ denotes mean normal, and

$N = (N_1|N_2|\cdots|N_D)$ denotes the transformation matrix between model and sample normal), which is similar to the calculation of model point vector X.

After processing all sample points, the parameters (t_x, t_y, s, θ, b) are updated to minimize the error between the transformed model shape X and newly found shape x. In order to ensure plausible shapes, every shape parameter b_d is constrained in a certain range, usually between $-3\sqrt{\lambda_d}$ and $3\sqrt{\lambda_d}$.

The above procedure is iterated until the shape model has converged. Fig. 1 shows the sequential ASM algorithm with self-adaptive search scale and its data flow.

1. *get_estimate*: Using the current parameters (t_x, t_y, s, θ, b) to generate the model point positions using Equation 2;
2. *get_scale*: Automatically control the searching scale ρ_i for each point X_i using the Kalman filter mechanism;
3. *local_search*: Examine a region of the image in the scale of ρ_i around each point X_i to find the best new position;
4. *update_param*: Update the parameters (t_x, t_y, s, θ, b);
5. *constrain_b*: Apply constraints to the parameters, b, to ensure plausible shapes;
6. Repeat until convergence.

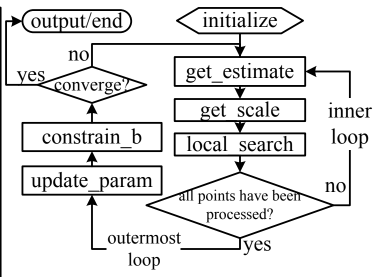


Fig. 1. The sequential ASM algorithm with self-adaptive search scale and its data flow

3 Characteristics of ASM Algorithm

As shown in Fig. 1, ASM algorithm executes in an iterative approach after the parameters (t_x, t_y, s, θ, b) are initialized. Each loop iteration of the outermost loop updates the parameters at step 4 for next iteration, which results in flow data dependence between consecutive iterations. Thus, the outermost loop iterations have to be executed in sequence. However, within a given outermost loop iteration, there exist both spatial parallelism and temporal parallelism in the inner loop from step 1 to step 3. Experimental results on general-purposed processors show that the computational cost of these three steps is about 76% of the time of the whole sequential algorithm. Therefore, we focus on these three steps in this paper. The *update_param* and *constrain_b* steps will not be emphasized.

In *get_estimate* step, the $2i$'th and the $(2i+1)$'th element in the $2n$-element vector $\overline{x}$ are the x and y value of the i'th mean point respectively. They are processed following Equation 6:

$$x_i^{'} = \overline{x}_i + \sum_{j=1}^{D} P_{2i,j} b_j, \qquad y_i^{'} = \overline{y}_i + \sum_{j=1}^{D} P_{2i+1,j} b_j. \tag{6a}$$

$$x_i = t_x + (s\cos\theta)x_i^{'} + (-s\sin\theta)y_i^{'}, \quad y_i = t_y + (s\sin\theta)x_i^{'} + (s\cos\theta)y_i^{'}. \tag{6b}$$

where $(x_1, y_1, \cdots, x_n, y_n)$ forms the vector X.

The above equation shows that $x_i^{'}$ and $y_i^{'}$ are both needed while calculating x_i or y_i. The calculation of $x_i^{'}$ and $y_i^{'}$ are arranged in two arithmetic units respectively so that the calculation of x_i and y_i can be done in parallel. In addition, different sample points in vector $\overline{x}$ are streamed from inputs to outputs in a pipeline mode to exploit temporal parallelism.

According to Equation 4 and 5, *get_scale* step computing the value of ρ_i has the similar program characteristics with *get_estimate* step. Whenever the input elements of vectors $V(x_i)$, $V(y_i)$, $(n_i)_x$ and $(n_i)_y$ are all ready, the processing of $V(x_i)V(y_i)$ and $(n_i)_x^2 V(y_i) + (n_i)_y^2 V(x_i)$ can be triggered simultaneously. Since the calculation of $V(x_i)$, $V(y_i)$, $(n_i)_x$ and $(n_i)_y$ has no data dependence at all, they can work in separate arithmetic units in parallel and form a deep arithmetic pipeline chain.

The *local_search* step examines a region of the image in the scale of ρ_i around each point X_i to find the best new position. For each pixel X_{ij} in the scale of ρ_i around each point X_i, the edge response is calculated according to

$$|\sqrt{R^2_{X_{ij+}} + G^2_{X_{ij+}} + B^2_{X_{ij+}}} - \sqrt{R^2_{X_{ij-}} + G^2_{X_{ij-}} + B^2_{X_{ij-}}}|. \tag{7}$$

where $R_{X_{ij+}}$, $G_{X_{ij+}}$, $B_{X_{ij+}}$, $R_{X_{ij-}}$, $G_{X_{ij-}}$, $B_{X_{ij-}}$ denote the R, G, B value of the two pixels on both sides of X_{ij}. The processing for each X_{ij} can be organized in pipeline. Whenever the edge response is prepared, it will be compared with the current maximal edge response. After all X_{ij} for the current i have been checked, the pixel with the maximal edge response will be chosen to update parameters for next iteration.

According to the above analysis, *get_estimate* step can work in parallel with the calculation of $V(x)$, $V(y)$ and normal vector n. They are grouped into Stage 1. The value of ρ_i, results of n_i, $V(x_i)$ and $V(y_i)$, can be grouped into Stage 2. Due to the data dependence between *local_search* step and the value of x_i, y_i and ρ_i, *local_search* step has to follow the steps of *get_estimate* and *get_scale*, which is organized in Stage 3 as shown in Fig. 2.

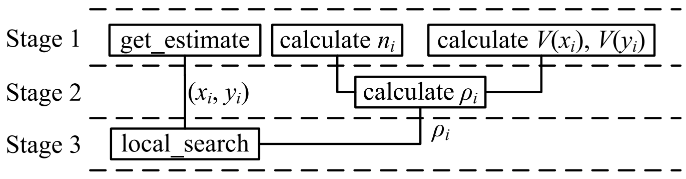


Fig. 2. Parallel characteristics of ASM kernel

4 FPGA Implementation of ASM Kernels

4.1 System Architecture

Based on the program characteristics of ASM algorithm, we implement the ASM kernels in an FPGA test-bed. Fig. 3 shows the structure of ASM hardware implementation. The FPGA test-bed includes a large capacity FPGA chip and a SDRAM module, connecting to Host processor through USB interface. All initial video frames are stored in SDRAM and the final results are transferred to Host processor for displaying.

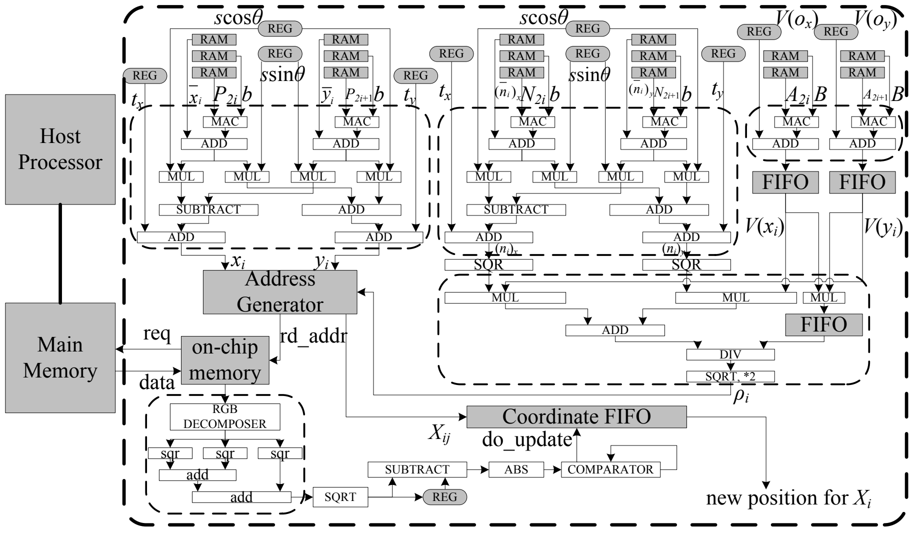


Fig. 3. System architecture for ASM kernel

The hardware implementation is organized into a deep pipeline network composing of many floating-point arithmetic operations. The *ADD*, *SUBTRACT*, *MUL*, *DIV*, *SQR*, *SQRT*, *MAC* unit in Fig. 3 represent floating-point addition, subtraction, multiplication, division, square, square root and multiplication-accumulation respectively. The *sqr*, *add* unit at the left bottom corner in Fig. 3 are fix-point operations for the processing of pixel values.

The module in the top left of Fig. 3 implements *get_estimate* step. Vector $\{\overline{x}_1, \overline{y}_1, \cdots, \overline{x}_n, \overline{y}_n\}$ is stored in the on-chip memory. According to the analysis in Section 3, this vector is decomposed into two vectors $\{\overline{x}_1, \cdots, \overline{x}_n\}$ and $\{\overline{y}_1, \cdots, \overline{y}_n\}$, which are placed in two separated on-chip memory modules so that the modules can provide two operands per cycle. Likewise, the matrix P are divided into two parts and stored in different memory modules. And constants t_x, t_y, $s\cos\theta$, $s\sin\theta$ are stored in registers. Following the pipeline chain, the results x_i and y_i are produced and stored in on-chip memory, being used to generate addresses for accessing main memory.

The module in the top middle and the module in the top right of Fig. 3 implement the calculation of $(n_i)_x$, $(n_i)_y$, $V(x_i)$ and $V(y_i)$, respectively. While calculating $V(x_i)$ and $V(y_i)$, we use matrix A to denote $\{A_{ij} = ((QGP)_{i,j})^2 | i = 1, 2, \cdots, 2n, j = 0, 1, ..., l-1\}$, and vector B denotes $\{B_j = \sigma_j | j = 0, 1, ..., l-1\}$.

Since their calculation is similar to the processing of *get_estimate* step, these two modules have the similar pipeline structure.

The above three pipeline units work independently. As the length of pipeline unit calculating $V(x_i)$ and $V(y_i)$ is shorter than the other two pipelines, it produces the results earlier. For data synchronization, $V(x_i)$ and $V(y_i)$ are buffered in FIFOs to wait for $(n_i)_x$ and $(n_i)_y$. There is still a FIFO unit for balancing the pipeline latency in the unit producing ρ_i according to Equation 5.

Address Generator unit uses values of x_i, y_i and ρ_i to get addresses for accessing a given segment in a video frame. The addresses are generated for the two pixels on both sides of each pixel X_{ij}, where X_{ij} is the pixel around X_i within the range of ρ_i along the normal line. Because the frame data are stored in *Main Memory*, random accesses to pixel X_{ij} will result in long memory latency. In order to improve the memory efficiency for loading random data, we propose a scheme buffering part of the whole image in on-chip RAM block to reduce the number of random accesses. The detail will be described in section 4.2.

In the unit at the left-bottom corner, R, G, B information of pixels decomposed from RAM blocks are calculated to get the edge response for each X_{ij}. After the maximal edge response are compared from all X_{ij} for the current i, the coordinates stored in *Coordinate FIFO* unit are chosen as the new position for X_i and used to update parameters for next iteration.

4.2 Eliminating Random Access to SDRAM by On-Chip Memory

As the frame data are stored in *Main Memory* implemented by SDRAM, random accesses to pixel X_{ij} will greatly suffer from long memory latency during the *local_search* step. On the other hand, on-chip RAM can not hold the whole frame data. In order to improve the memory efficiency for loading random data in large images, we propose a scheme buffering part of the whole image in on-chip RAM block to eliminate random accesses to SDRAM, which takes full advantage of both high speed of on-chip memory and large capacity of off-chip memory. Data reusability is exploited to reduce the number of I/O between *Main Memory* and FPGA.

According to the characteristics of local search step, all the sample points X_i generated in the *get_estimate* step compose an estimated people contour based on the mean shape, as shown in Fig. 4. For a given X_i in the contour, there exists an smallest area for searching, which is a rectangular holding the elements of all X_{ij} along the normal line of X_i as shown in the right corner of Fig 4.

The key idea of eliminating random accesses is that before searching the maximal edge response, we prefetch a block of pixel values containing a number of searching rectangular from *Main Memory* into on-chip RAM. The block data will be reused by several sample points respectively, as shown the shade rectangular area in Fig 4. The searching sequence of all X_i should be arranged from top to bottom as shown the sequence number $(1, 2, 3, \cdots)$ in Fig. 4. Accordingly, vectors $\{\overline{x}_1, \cdots, \overline{x}_n\}$ and $\{\overline{y}_1, \cdots, \overline{y}_n\}$ are both reordered in a sequence of Y-coordinate, from top to bottom, so that x_i, y_i and ρ_i are produced in this order, too.

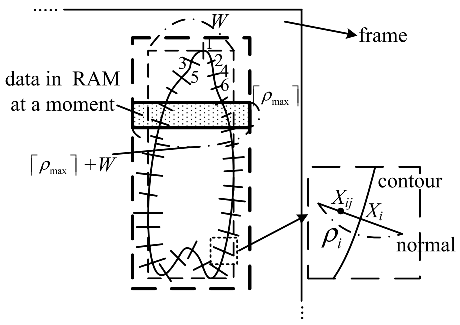


Fig. 4. Illustration of a people contour

Because the directions of normal lines for different sample points are variable, the possible positions of X_{ij} are distributed in a circle around X_i whose diameter is ρ_i. Therefore, the height of the frame data in the on-chip RAM should be no less than $\lceil \rho_{max} \rceil (\rho_{max}$ is the maximum of all ρ_i in the contour, where $i = 1, 2, \cdots, n)$. In addition, the width of the rectangle is $(\lceil \rho_{max} \rceil + W)$, where W is the width of the contour. To summarize, the number of the pixels that should be in the on-chip RAM at the same time is $(\lceil \rho_{max} \rceil + W) * \lceil \rho_{max} \rceil$. And because each pixel value containing R, G, B information is 24bits, the capacity of the on-chip memory should be $(\lceil \rho_{max} \rceil + W) * \lceil \rho_{max} \rceil * 24$bits.

Different contours may have different W and ρ_{max}. We measured the variation of search scales for 128 sample points on three people contours, whose sizes are 20×76, 32×96 and 50×110 respectively. Fig. 5 gives part of these results.

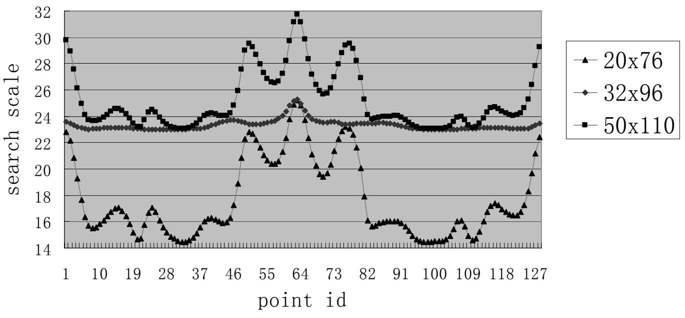


Fig. 5. Variation of search scales for different sample points

We can find that the search scale changes in a certain range, which is calculated according to Equation 5. The search scales of 128 sample points on the contour of size 20×76 range from 14.45 to 25.22. If the contour size is 32×96, the minimum is 22.99, the maximum is 25.26. For the third contour, these statistics are 23.04 and 31.71 respectively. After doing some more experiments on different contours, we conclude that the search scale will be no larger than 50 when the people contour is no bigger than 240×320. Therefore, we use 50 and 240 as the upper limit of ρ_{max} and W in our work. Hence, the block capacity should be no less than $(240 + 50) * 50 * 24 = 339.8K$bits. We deploy a M-RAM memory block of Altera FPGA with size of 512Kbits as the searching buffers.

4.3 Linear MAC Unit

get_estimate and *get_scale* step require 6 multiplication-accumulation (MAC) units, each of which operates on two vectors inputs to get a single value for result. Since there is no data reuse in this operation, it belongs to I/O bound operations. In particular, its throughput is determined by the vectors input rate.

Assuming that m pairs of numbers are to be processed in a MAC operation, tree-based MAC unit is one of implementation approaches. The left part of Fig. 6 illustrates this kind of structure. When many MAC operations are processed using this pipelined architecture, very high throughput can be achieved. It can output a result for one MAC operation in every clock cycle. The output rate of $\{x_i, y_i, \rho_i\}$ will reach one result per clock cycle. But the tree structure costs plenty of computing resources, requiring m multipliers and $(m-1)$ adders. In addition, it also requires a lot of memory resources. For reading m pairs of values from the memory blocks concurrently, we have to store data in m memory modules to provide sufficient access ports, which will bring difficulties during the Place and Route (P&R) phase.

In fact, in our work, the throughput of the *get_estimate* and *get_scale* step do not have to be very high at all, since the bottleneck of the system is the *local_search* step in most cases. As long as the output rate of $\{x_i, y_i, \rho_i\}$ is not slower than that of the *local_search* step, throughput reduction of $\{x_i, y_i, \rho_i\}$ to a certain extent will not affect the performance of the system.

In our design, all elements in each vector are stored in a single memory module. It means that only one pair of elements is loaded in each clock cycle. Considering this situation, we improve the tree-based architecture for our work in this paper, as shown in the right part of Fig. 6. m multipliers are merged into one multiplier, and $m-1$ adders are replaced by $\lceil \log_2 m \rceil$ adders.

In the improved stucture, m pairs of elements flow into it sequentially. The outputs of the multiplier are decomposed into $\lceil m/2 \rceil$ groups, each of which contains two adjacent outputs. The former output is buffered in a register for

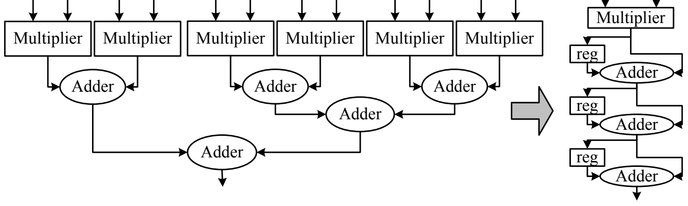


Fig. 6. Tree-based MAC unit and linear MAC unit

synchronization with the later output. Then they are sent into the next adder unit simultaneously. Similarly, two adjacent outputs of the former adder unit are sent into the next adder unit in the same way.

5 Experimental Results

In this section, we examine the performance of our design on a single FPGA. Our target device is Altera Stratix II EP2S130F1020C5. This device contains 106,032 ALUTs, about 6 Mb of on-chip memory and 743 I/O pins. In our experiments, we used Quartus II and Mentor Graphics ModelSim as development tools.

Experiment 1: Resource utilization and clock speed. We designed our own floating-point adder, multiplier, divider and SQRT unit. These floating-point units comply with the IEEE-754 single-precision format. Their characteristics are shown in Table 1. The high clock speed of these units helps a lot for increasing the performance of the system.

We have implemented *get_estimate*, *get_scale* and *local_search* on FPGA. Their performance characteristics are shown in Table 2. A lot of floating-point operations are used in the system, especially addition and multiplication. Because we use four DSP block 9-bit elements in each multiplier, about 15% of such DSP elements are used. By using our own high-performance floating-point units, the system gets a high clock speed. Because of the control logic, the clock speed is a little slower than that of the floating-point units.

Experiment 2: Performance comparison for different memory accesses. We compare the performance of the *local_search* step between using and not using the on-chip memory for buffering. Table 3 gives the details. If the *local_search* step randomly accesses the *Main Memory* directly and the on-chip memory is not used, the effective I/O bandwidth will degrade greatly, resulting in lower performance. Experimental results in row 4 show that the more sample

Table 1. Floating-point units

	Adder	Multiplier	Divider	SQRT
Stages of Pipeline	8	3	8	16
Area(ALUTs)	579	107	243	860
Clock Speed(MHz)	228	165	122	238

Table 2. ASM kernel implementation

	No. of floating-point units				DSP block	Area(ALUTs)	Clock Speed
	Adder	Multiplier	Divider	SQRT	(% of 504)	(% of 106032)	(MHz)
get_estimate	14	6	0	0	24(5%)	9968(9%)	93
get_scale	25	13	1	1	52(10%)	11725(11%)	91
local_search	1	0	0	1	5(<1%)	2415(2%)	184

Table 3. Performance comparison for different memory accesses

contour size	50 × 110		32 × 96		20 × 76		
no. of sample points	128	64	128	64	128	64	32
cycles(with RAM)	10752	10687	7112	7045	4143	4122	4107
cycles(without RAM)	52478	26174	47424	23552	32772	16384	7683
speedup	4.88	2.45	6.67	3.34	7.91	3.97	1.87

Table 4. Performance comparison with the software implementation in Pentium 4

contour size	50 × 110		32 × 96		20 × 76		
no. of sample points	128	64	128	64	128	64	32
execution time of PC(us)	977.22	526.05	843.19	403.89	769.08	379.42	320.11
execution time of FPGA(us)	123.36	122.63	82.91	82.17	49.92	49.69	49.52
speedup	7.92	4.29	10.17	4.92	15.40	7.64	6.46

points are chosen in a contour, the more time is spent. But, if we use the scheme proposed in Section 4.2, the image data can be read from the *Main Memory* sequentially and buffered in the on-chip memory. At the same time, the buffered data can be accessed randomly from RAM per cycle. Experimental results show that the maximal speedup can be near 8 and the speedup rises with increase in the number of sample points.

Experiment 3: Performance comparison with the software implementation in Pentium 4. We compare the performance of the FPGA design (90MHz) presented in Section 4 with the software solution in Pentium 4(2.8GHz) as shown in Table 4. The software solution is sequentially implemented according to Fig. 1. The program code is written in C++ language and it is optimized. For example, BLAS (Basic Linear Algebra Subprograms) library is used to optimize the matrix multiplication operations, which are frequently used in the algorithm. The execution time given in Table 4 is for a single iteration in ASM algorithm. Three people contours are used to examine the performance, whose sizes are 20×76, 32×96 and 50×110. We test the performance while choosing different number of sample points on the same contour. The results show that the more sample points are chosen in a contour, the more time is spent in the PC solution. The reason is that the system is sequentially implemented in PC, and the number of operations grows with the increase of the number of sample points. We can also find in Table 4 that the performance of the FPGA implementation depends much more on the size of the contour than on the number of the sample points. The larger the contour is, the more time is spent. That is because the system needs to read all the data in the rectangular which holds the contour from the off-chip memory to the on-chip memory, and the time requirements of this transmission influence the performance of the system. To sum up, the speedup grows with the increase of the number of sample points and the decrease of the contour size. In this experiment, speedups of at least 4.29 and up to 15.40 are achieved. The average speedup is 8.11.

6 Conclusion

In this paper, we implemented an ASM-based people tracking system in a reconfigurable hardware. ASM is one of the most popular methods for recognizing non-rigid objects. Based on the analysis to the characteristics of ASM algorithms, we proposed a pipelined architecture to prompt the performance of people tracking applications. The hardware implementation is organized into a deep pipeline network composing of multiple floating-point arithmetic units. In order to improve the memory efficiency for loading random data, we proposed a hierarchical memory scheme to eliminate random accesses to off-chip memory. All proposed design are implemented in a FPGA test-bed. The experimental results show that hardware speedups can reach from 4.29 to 15.40, compared with sequentially-implemented software solution in Pentium 4.

Acknowledgments. This work is supported by NSFC 60633050 (National Natural Science Foundation of China) and NSFC 60621003.

References

1. Kass, M., Witkins, A., Terzopoulos, D.: Snakes: Active contour models. In: Proc. First International Conference on Comuputer Vision, pp. 259–268 (1987)
2. Wiskott, L., Fellous, J., Kruger, N., der Malsburg, C.V.: Face recognition by elastic bunch graph matching. IEEE Trans. Pattern Anal. Machine Intell. 19, 775–779 (1997)
3. Cootes, T.F., Taylor, C.J., Cooper, D.H., Graham, J.: Active shape models–their training and application. Computer Vision and Image Understanding 61, 38–59 (1995)
4. Baumberg, A., Hogg, D.: An efficient method for contour tracking using active shape models. In: Proc. IEEE Workshop on Motion of Non-Rigid and Articulated Objects, pp. 194–199 (1994)
5. Li, Y., Lai, J.H., Yuen, P.C.: Multi-template asm method for feature points detection of facial image with diverse expressions. In: Proc. 7th International Conference on Automatic Face and Gesture Recognition, pp. 435–440 (2006)
6. Zhao, Z., Teoh, E.K.: A novel 3D statistical shape model for segmentation of medical images. In: Proc. of the 2006 International Symposium on Visual Computing, pp. 638–647 (2006)
7. Rogers, M., Graham, J.: Robust active shape model search. In: Proc. 7th European Conference on Computer Vision, pp. 517–530 (2002)
8. Baumberg, A.: Leaning deformable models for tracking human motion. Ph.D. dissertation. Univ. of Leeds, Leeds (1995)
9. Blake, A., Curwen, R., Zisserman, A.: A framework of spatio-temporal control in the tracking of visual contours. International Journal of Computer Vision 11, 127–145 (1993)

Performance Evaluation of Evolutionary Multi-core and Aggressively Multi-threaded Processor Architectures

Partha Tirumalai, Yonghong Song, and Spiros Kalogeropulos

Sun Microsystems, Inc., 15 Network Circle, Menlo Park, CA 94025, USA
{partha.tirumalai,younghong.song,spiros.kalogeropulos}@sun.com

Abstract. Processor architecture is undergoing a significant change in response to the rapidly escalating complexities of high-power, high-frequency, and increasingly superscalar designs. Evolutionary multi-core and aggressively multi-threaded chips are appearing in the general purpose microprocessor space. The latter offer simplicity, low power, and high performance on threaded workloads but with somewhat reduced single thread performance. This paper examines the performance of the SPARC64(TM) VI, a dual-core 4-thread processor, and the UltraSPARC(TM) T1, an 8-core 32-thread processor. Numerous workloads are executed on both designs. These include single thread speed tests, homogeneous throughput tests, and multi-threaded tests using varying amounts of data and parallelism. The results indicate a clear separation in the workloads that are best suited to each design. To reap the full benefit of these multi-threaded designs, software has to be architected to use as many threads as possible. This shift is likely to affect both software developers and compiler writers for the next several years.

1 Introduction

For about three decades processor architecture has steadily improved performance using designs that operate at increasing clock frequencies and processing larger numbers of program instructions simultaneously. Techniques such as deep pipelining, multiple instruction issue, and out-of-order execution, coupled with complex cache-memory hierarchies and prefetching have led to vast increases in performance. These advances have been made possible by continuing steps in process technology. Recently, however, progress in this architectural direction has slowed. High power dissipation and the extreme complexities of these designs have forced architects to consider alternatives for utilizing the increasing number of transistors on a chip.

Modest multi-core/multi-threaded designs are now being offered by every major general purpose microprocessor producer [1], [2], [3], [4], [5]. In these evolutionary designs, there is no radical architectural change from the past. An existing high-clock rate, superscalar core is leveraged and multiple cores are placed on one die, taking advantage of the higher transistor densities offered by process technology steps. These designs continue to focus on single thread performance, tend to consume high power, and offer only a few hardware threads. Recently, a few brand new aggressively multi-threaded designs have also emerged [6], [7] in the general purpose microprocessor space. These designs have been architected from scratch to support a large number of hardware threads. They use simple cores operating at a

© Springer-Verlag Berlin Heidelberg 2007

relatively low clock frequency and target high throughput on multi-threaded workloads with low power consumption [8], [9].

In this paper, we evaluate the performance of the above two processor architectures on a variety of workloads. For our study, we selected two designs based on the same instruction set architecture. This enables us to run exactly the same code on both designs. Aggressive CMT designs are still not very common in the general purpose microprocessor space. Azul Systems offers compute appliances based on the 24-core Vega-1 and the 48-core Vega-2 processors. However, these systems operate as Java co-processors rather than as standalone, general purpose computers. Also, a corresponding conventional design is not available. Sun Microsystems offers an 8-core, 32-thread UltraSPARC T1 processor which is based on the SPARC V9 ISA. A corresponding contemporary and conventional design is the SPARC64 VI processor [12] which is also based on the SPARC V9 ISA. Systems based on these two designs are available and can run the same SPARC/Solaris executables. This allows performance comparisons to be made across the two architectures while keeping the software stack the same.

We try to answer the following questions with this study. What is the impact of an aggressive CMT design on serial workload performance? Do parallel workloads benefit sufficiently from a design that offers many, but less powerful, hardware threads? What are the performance profiles of the two designs and how do workloads map to each? What software and compiler optimizations are important to these designs? And, finally, what kind of changes are likely to provide maximum benefit in future CMT and conventional systems.

2 Description of the Processors

Fig. 1 shows a simple block diagram of a SPARC64 VI, the conventional modest multi-core processor used in our study. This chip contains two cores which are designs leveraged from the previous SPARC64 V chip. Each core is a four-instruction-issue out-of-order execution engine and is capable of handling two threads. Each core has its own private level 1 data and instruction caches, each 128kB in size. In our system, the chip operated at 2.28GHz and had a 5MB unified level 2 cache. The level 1 caches are shared by two threads, and the level 2 cache is shared by all four threads supported by the chip.

The two threads supported by a core operate in a vertically multi-threaded fashion. A thread switch occurs either when an executing thread suffers a level 2 cache miss, or a fixed time period has elapsed without a switch. The latter is to prevent thread starvation. Within a thread, instructions can be issued out of program order, up to four at a time (sustained). Dual ported caches support up to two loads per cycle. There is logic in the cores for sophisticated branch prediction. The level 1 caches are large to reduce cache misses. There is also a hardware prefetch unit which tracks cache misses and attempts to prefetch data in anticipation of future misses. It is clear that this design hopes to extract a significant amount of instruction level parallelism and execute multiple instructions per cycle from individual threads by reducing cache miss penalties and branch mispredict related stalls.

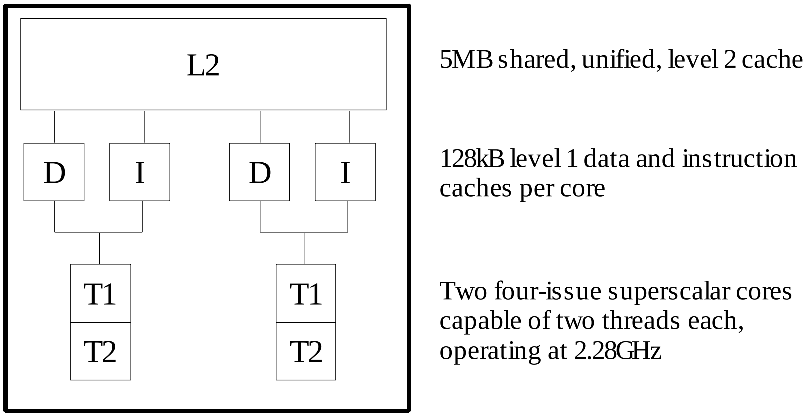

L2
D
I
D
I
T1
T2
T1
T2
5MB shared, unified, level 2 cache
128kB level 1 data and instruction caches per core
Two four-issue superscalar cores capable of two threads each, operating at 2.28GHz

Fig. 1. Block diagram of the SPARC64 VI

Fig. 2 shows the block diagram of an UltraSPARC T1 processor. This processor has eight cores each capable of handling four threads. The processor can thus execute 32 threads simultaneously. However, each core is very simple. Instructions are processed one at a time and in program order. There is no branch prediction and no hardware prefetching logic. Instructions are issued in round robin fashion from ready-to-run threads. Any long latency instruction such as a load or a branch takes a thread out of the ready-to-run pool until it is ready to run again. Thread switching is done on a per cycle basis with no overhead at all. Each core has its own level 1 data and instruction caches of 8kB and 16kB, respectively. There is also a shared unified 3MB level 2 cache on the chip. Four DDR channels are provided on the chip (not shown in the figure) to access memory with high bandwidth. It is clear that this design focusses on being able to execute a large number of threads but does not devote resources to execute a single thread with maximum speed. The efficiency of this design is targeted at workloads that have multiple concurrent threads. The clock rate also is low because the design focusses on low power operation.

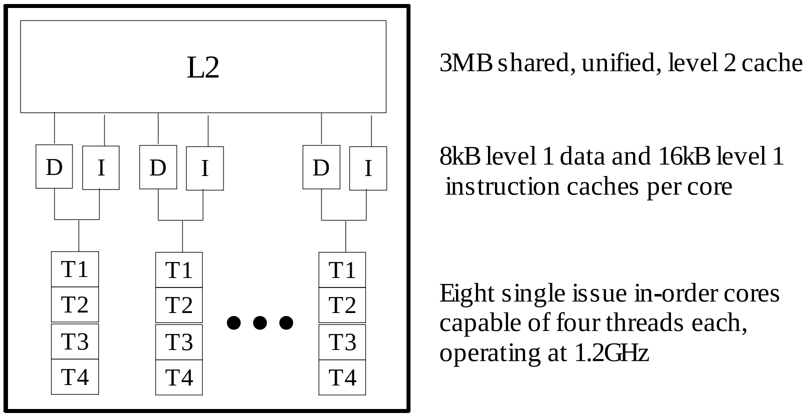

L2
D
I
D
I
D
I
T1
T2
T3
T4
T1
T2
T3
T4
T1
T2
T3
T4
3MB shared, unified, level 2 cache
8kB level 1 data and 16kB level 1 instruction caches per core
Eight single issue in-order cores capable of four threads each, operating at 1.2GHz

Fig. 2. Block diagram of the UltraSPARC T1

Systems based on the SPARC64 VI target typical enterprise back-room datacenter operations. Small, medium, and large to very large configurations are available. There is substantial focus on fault tolerance and reliability. Our test system used only one chip (2 cores, 4 threads) with a 960MHz bus and a 32GB of memory. However, systems with up to 64 chips (128 cores, 256 threads) and 2TB of memory are available. UltraSPARC T1 based systems are targeted at web services, network infrastructure workloads, and security. These systems focus on high compute density and minimizing space, power, and cooling requirements. Our test system had 1 chip (8 cores, 32 threads) with a 200MHz bus and 32GB of memory. Systems with up to 64GB of memory are available.

3 Performance of Different Workloads

3.1 Single Thread Performance

We evaluated the performance of the two processors on ten programs taken from the SPEC CPU2000 integer suite. Two of the twelve programs in this suite contain a significant amount of floating point operations even though they have been placed in the integer collection by SPEC. These two programs, eon and vpr, were discarded because the UltraSPARC T1 processor is designed only to handle incidental FP. There is only a single FPU on the chip and it is shared by all 32 threads. Some FP instructions even trap to software. As such FP intensive workloads can be summarily dismissed as not being a suitable target for the UltraSPARC T1.

Fig. 3 shows the results of the single thread performance comparison. The SPARC64 VI is designed for an ideal single thread execution rate of 2.28*4 = 9.12 BIPS. Similarly, the UltraSPARC T1 can execute at most 1.2*1 = 1.2 BIPS from a single thread. The thick line at the top of the graph represents this *ideal* speedup of 7.6. The actual speedup is considerably less, typically around 5.5, but it is quite significant. SPEC CPU integer programs are known to have a low cache miss rate. In this suite, only mcf has noticeable cache misses. On this program, the speedup achieved is only about 2.2. This is because both processors become limited by memory latency and the abundant resources in the SPARC64 VI core are less effective on it. The traditional design of the SPARC64 VI does very well on the other, largely cache resident, programs.

3.2 Throughput Performance

The SPEC CPU2000 benchmark includes a throughput metric in addition to the popular speed metric. The throughput or rate score is measured by simultaneously running *n* copies of each benchmark. Any value of *n* may be used. Typically, vendors will choose to run as many copies as the number of hardware threads in their system. However, this also means that the data footprint, and thus the cache misses suffered, of a system running a large number of copies may be more than those of a system running a fewer number of concurrent copies. This is not a fundamental problem with the metric, but in our case the two architectures we are comparing have significantly different numbers of hardware threads. To make sure we are running the same workload on both, we chose to measure throughput by running 32 copies.

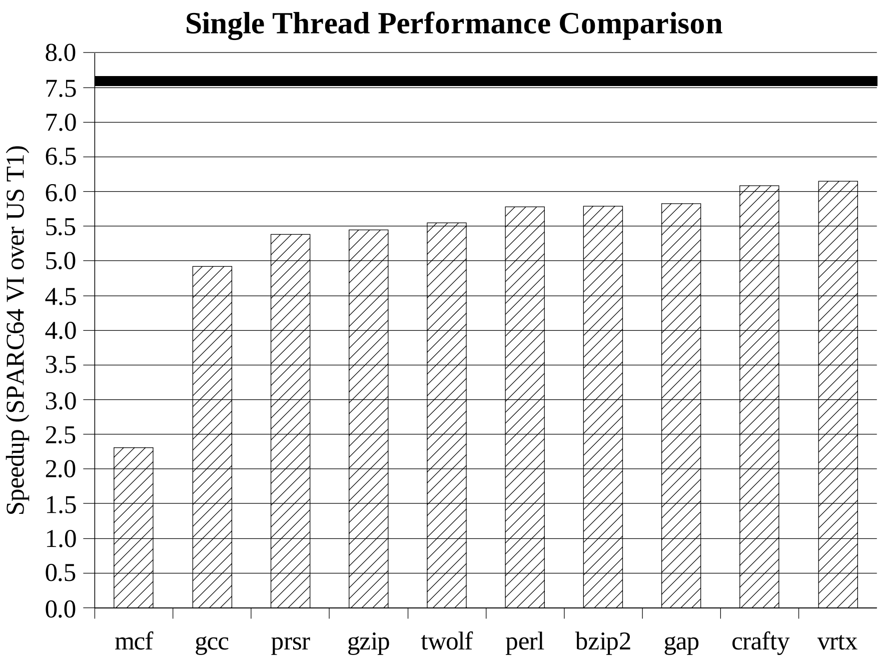


Fig. 3. Speedup on serial execution: 2.28GHz SPARC64 VI over 1.2GHz UltraSPARC T1

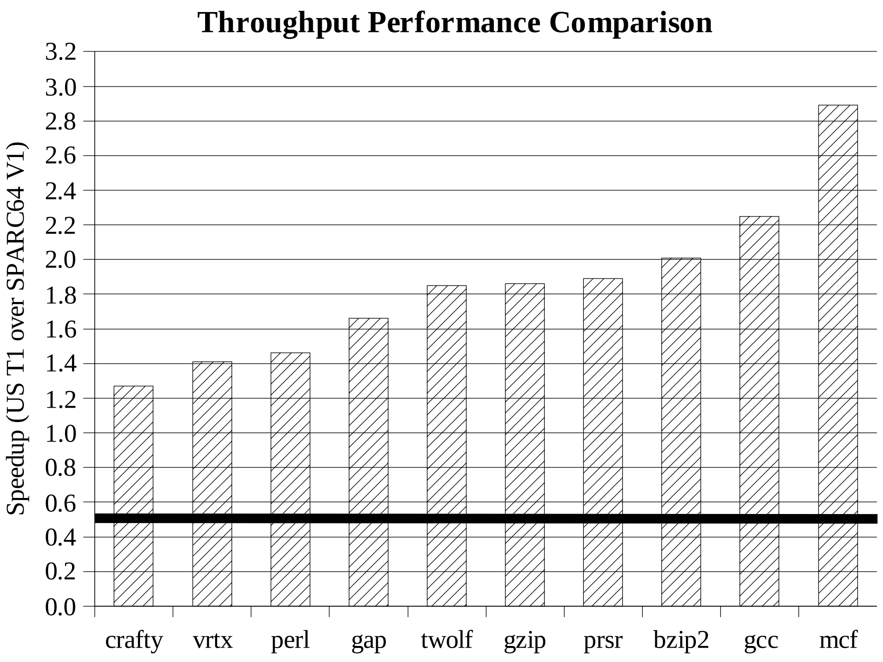


Fig. 4. Speedup on throughput: 1.2GHz UltraSPARC T1 over 2.28GHz SPARC64 VI

Fig. 4 shows the results of the throughput comparison. Note that with respect to throughput, the peak capacity of the SPARC64 VI is 2.28*4*2 = 18.24 BIPS. Correspondingly, the peak capacity of the UltraSPARC T1 is 1.2*1*8 = 9.6 BIPS. The thick line in the graph shows this *ideal* ratio of 0.53. That is, based on the designed in capacity, the SPARC64 VI should be expected to deliver 1.9X more throughput than the UltraSPARC T1. However, the results are almost exactly the opposite. The UltraSPARC T1 is able to squeeze out more throughput from its lower capacity than the SPARC64 VI. On the average, performance increases by 1.9X on the Ultra T1. Mcf, with its noticeable miss rate, achieves ~3X more performance on the UltraSPARC T1. Cache resident, high ILP programs, are the most favorable to the SPARC64 VI, though even here the UltraSPARC T1 remains ahead.

3.3 Synthetic Commercial Performance Profile

Commercial codes have behaviors that are not reflected well in the SPEC CPU programs we used in the previous two sections. However, large commercial codes are difficult to acquire, set up, and run. Many are proprietary. To study commercial code performance we used an internally developed benchmark, *Comscape.* This benchmark mimics code patterns typically observed in the traces of various customer codes [10], [11]. The instruction mix consists of approximately 20% loads, 10% stores, 50% ALU, and 20% branches. The branches are more unpredictable and the loads more difficult to prefetch than those in SPEC CPU. *Comscape* can be executed with varying numbers of threads. By increasing the size of the databases processed, the cache miss rate can be varied from very small to very large. Performance is measured in terms of the number of application instructions completed per second.

Fig. 5 shows the performance profiles generated by *Comscape* on the SPARC64 VI (dashed lines) and the UltraSPARC T1 (solid lines). When 1 thread is used, the SPARC64 VI starts at about 4 billion instructions/sec. on a cache resident dataset. This means it is processing a good 1.75 instructions/cycle when the cache miss rate is negligible. However, performance falls off rapidly as the dataset grows and the miss rate increases. Performance does improve significantly with 2 threads, coming close to the ideal 2X across the full tested range. With 4 threads, the gain in performance is not noticeable on small datasets. At high cache miss rates, the multithreading within each core does show gains approaching 20%. For the full chip (4 threads), performance drops below 1 instrn/cycle at over 16 memory accesses/1000 instrns.

On the UltraSPARC T1, performance with 1 thread is only about 0.5 instrn/cycle even when there are no memory accesses. This processor has no ability to hide load, branch, or any other stalls within a single thread. As threads are added, however, performance scales nearly linearly up to 8 threads. All 8 cores of the chip are utilized at this point. At 16 and 32 threads, the threading within the core comes into play and the execution resources inside a core are better utilized by multiple threads. Performance continues to increase beyond 8 threads for this reason, though not quite linearly. The dip in the middle with 32 threads is due to unfortunate cache conflicts. We could move the data objects to avoid this conflict but we chose to show the curve without such tuning. Such conflicts can and will occur in real life and not all users will detect them and tune their application. In spite of the dip, performance is quite good.

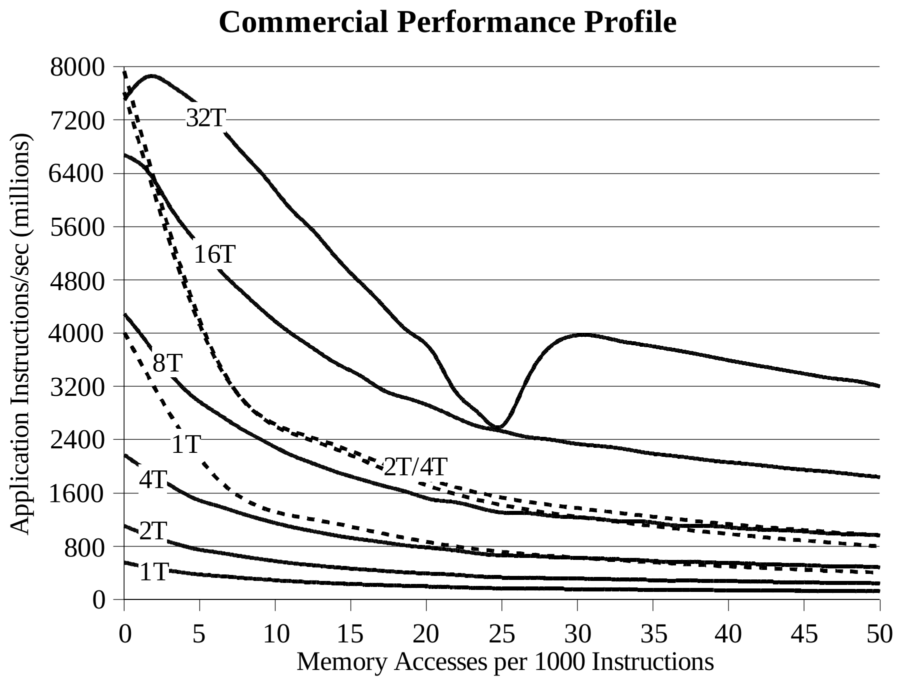


Fig. 5. Performance on *Comscape*, a synthetic benchmark for commercial codes

Comparing the two architectures, we see that if there are less than 4 threads, the SPARC64 VI is a clear winner for all but the large datasets. This is the processor of choice for workloads that have low thread count and small to medium datasets. However, when the thread count is 16 or more, the UltraSPARC T1 does very well, particularly on large datasets. Given a target workload's parallelism and memory access rate, it is then possible to estimate which architecture might be the one more suitable for executing it.

4 Impact of Compiler Optimizations

The UltraSPARC T1 has eight very simple cores. For decades, however, compiler optimizations have advanced with the goal of extracting performance from complex superscalar cores like the ones in the SPARC64 VI. Some modern optimizations such as trace scheduling, prefetching, if-conversion, and the use of non-faulting loads trade increased instruction count for more efficient instruction execution. These optimizations might not be helpful on the UltraSPARC T1. Classic optimizations such as common sub-expression evaluation, invariant hoisting, and various loop transformations operate without such a trade off, and can be expected to help both processors. The impact of optimizations on the two architectures could, therefore, be quite different on the two architectures.

Scores of optimizations are implemented in modern compilers. We studied the impact of a few selected optimizations in our compiler, Sun Studio 12, to gauge their impact on the throughput performance of the two architectures. Table 1 shows the results of this study. The top half has data on the SPARC64 VI and the bottom half the same on the UltraSPARC T1. Performance is normalized to the case when optimization is turned off (*-O0*) in our compiler. The first column has *all* the optimizations available in the compiler. The half dozen columns in between have various optimizations turned off, one at a time and are briefly described as follows:

- *nogsch* – No global scheduling. Global scheduling uses heuristics and profile feedback information to move instructions from one basic block to another. Instructions executed speculatively are executed in a non-faulting manner. Global scheduling increases the number of instructions executed.
- *noinli* – No inlining. Inlining uses heuristics and profile feedback information to selectively inline functions at appropriate call sites. Although inlining by itself would only reduce call/return overhead, it can expose opportunities for other optimizations previously barred by a call.
- *noloop* – No loop multi-versioning. Loop versioning emits two or more versions of certain loops and selects the optimal version at run time. This optimization can be useful if the knowledge of certain conditions helps optimize the loop well.
- *nounro* – No loop unrolling. Loop unrolling reduces branches and overhead instructions by executing them once for multiple iterations. It can also expose opportunities for other optimizations by combining multiple iterations together. However, it does increase code size.
- *nopre* – No software prefetching. Software prefetching attempts to fetch data in advance of cache misses. Both loads and stores may be prefetched. The benefit depends on the application's cache miss rate as well as the compiler's ability to prefetch the misses. It involves adding instructions.
- *nostre* – No strength reduction. Strength reduction involves replacing instructions with other instructions that are expected to execute faster. For example, a multiplication might be replaceable with a few shifts and adds. On many processors, shifts and adds are faster than doing the multiply.

Full optimization had a huge impact on both processor architectures. Comparing the first and last columns, performance increased by 2.9X on the SPARC64 VI. The increase was only slightly less, 2.6X, on the UltraSPARC T1. Removing most individual optimizations did not cause large performance drops. Inlining enables many optimizations and had the largest impact among the optimizations tested with drops of 16% and 11% on the SPARC64 VI and the UltraSPARC T1, respectively. The benefit of individual optimizations varies with the processor and the application behavior. For example, *mcf* is the toughest program to optimize on both machines. It has quite a lot of cache misses. On the SPARC64 VI processor, the hardware prefetch unit does a good job and so the benefit of software prefetch is not evident. However, on the UltraSPARC T1, this difficult program derives about one-third of its total optimization gain from software prefetch. Similarly, vortex has some large loops with inter-iteration dependences and control. Loop unrolling does not help it much. On the SPARC64 VI, the impact of turning off unrolling is small, but on the UltraSPARC T1 the instruction cache is quite small, and turning off this ineffective unrolling results in a noticeable gain.

Table 1. Impact of compiler optimizations on the two processor architectures

SP64 VI	all	nogsch	noinli	noloop	nounro	nopref	nostre	none
gzip	3.13	3.12	2.95	3.12	2.86	3.13	3.12	1.00
gcc	2.03	2.04	2.01	2.08	2.07	2.09	2.12	1.00
mcf	1.69	1.68	1.68	1.69	1.67	1.68	1.68	1.00
crafty	3.30	3.30	2.76	3.27	3.30	3.29	3.29	1.00
prsr	2.93	2.94	2.33	2.93	2.92	2.93	2.93	1.00
perl	3.23	3.28	3.07	3.24	3.25	3.24	3.22	1.00
gap	3.18	3.19	2.69	3.18	3.16	3.19	3.17	1.00
vrtx	3.54	3.59	2.01	3.56	3.58	3.54	3.58	1.00
bzip2	3.68	3.72	3.29	3.78	3.70	3.68	3.64	1.00
twolf	2.79	2.78	2.65	2.74	2.79	2.79	2.78	1.00
G.M.	**2.88**	**2.89**	**2.49**	**2.88**	**2.85**	**2.88**	**2.88**	**1.00**

US T1	all	nogsch	noinli	noloop	nounro	nopref	nostre	none
gzip	3.19	3.23	2.92	3.22	3.11	3.14	3.24	1.00
gcc	2.30	2.30	2.23	2.28	2.24	2.31	2.31	1.00
mcf	1.44	1.30	1.41	1.51	1.41	1.29	1.33	1.00
crafty	2.55	2.47	2.06	2.47	2.44	2.40	2.41	1.00
prsr	2.60	2.61	2.32	2.60	2.61	2.59	2.62	1.00
perl	2.84	2.85	2.81	2.81	2.72	2.79	2.79	1.00
gap	2.93	2.94	2.89	2.91	2.82	2.90	2.80	1.00
vrtx	3.49	3.54	2.33	3.48	3.72	3.48	3.51	1.00
bzip2	3.25	3.31	2.93	3.26	3.24	3.26	3.28	1.00
twolf	1.82	1.80	1.75	1.84	1.91	1.83	1.85	1.00
G.M.	**2.56**	**2.54**	**2.31**	**2.56**	**2.54**	**2.51**	**2.52**	**1.00**

5 Conclusion

Escalating power consumption and the design complexity associated with increasing clock frequency and extracting performance from serial workloads is forcing a shift in general purpose processor architecture. In this paper we have compared two designs that combat this problem. The first design, a 2.28GHz SPARC64 VI, is an evolutionary one leveraging traditional superscalar cores and possessing limited parallelism. The second design, a 1.2GHz UltraSPARC T1, supports a large number of threads but with limited capability within each. It also operates at a modest frequency to keep power consumption down. Both designs implement the SPARC V9 instruction set and run the Solaris(TM) operating system. This makes it possible to compare the exact same software stack executing on the two architectures.

On serial workloads, the SPARC64 VI is clearly superior. It wins by about 5X on small footprint, cache resident single threaded programs. When the memory accesses increase, this lead drops considerably as both designs become dominated by memory latency. On workloads containing many threads, the 8X more threads in the UltraSPARC T1 provided an excellent gain. In spite of this chip having about half the total instruction execution capability of the SPARC64 VI, it achieved about twice the throughput. Results from a synthetic commercial benchmark indicate that a performance gain of up to 3X is possible for either chip depending on the characteristics of the workload. The SPARC64 VI is to be preferred when the number of threads is less than 4, and the memory access rate is less than about 16 per one thousand instructions. With 12 or more threads, and higher memory access rates, the UltraSPARC T1 is the processor of choice. Optimization was not unimportant for the simple UltraSPARC T1 design. In fact, it was almost as valuable as for the SPARC64 VI, providing approximately 2.4X in performance compared to unoptimized code. Inlining was an important optimization because it created opportunities for many other optimizations, but otherwise no single optimization had a big impact. This suggests that the road for compiler writers is a long and difficult one – while there is much total opportunity, there is no quick and easy win.

For the future,it appears natural that both architectures will evolve and move toward each other as they work to eliminate their weaknesses. A large number of fairly powerful threads will soon be available to the average programmer. Tools for parallelization and the education of a new generation of software developers to "think parallel" when designing applications will become critically important.

References

1. Tendler, J.M., et al.: Power4 System Micro-architecture. IBM Journal of Research and Development 46(1), 5–26 (2002)
2. Keltcher, C., McGrath, K., Ahmed, A., Conway, P.: The AMD Opteron Processor forMultiprocessor Servers. IEEE Micro 23(2), 66–76 (2003)
3. Krewell, K.: UltraSPARC IV Mirrors Predecessor - Sun Builds Dual-Core Chip in 130nm. Microprocessor Report, In-Stat/MDR (November 10, 2003)
4. McGregor, J.: A Day at the Races - AMD and Intel Rush Dual-Core Introductions. Microprocessor Report, In-Stat/MDR (May 9, 2005)
5. Kalla, R., Sinharoy, B., Tendler, J.: IBM Power5 Chip: A Dual-Core Multi-threaded Processor. IEEE Micro 24(2), 40–47 (2004)
6. Kongetira, P., Aingaran, K., Olukotun, K.: Niagara: A 32-Way Multi-threaded SPARCProcessor. IEEE Micro 25(2), 21–29 (2005)
7. Krewell, K.: Are Instruction Sets Irrelevant? - Scott Sellers' Azul Systems Bets On It. Microprocessor Report, In-Stat/MDR (February 14, 2005)
8. Tullsen, D.M., Eggers, S.J., Levy, H.M.: Simultaneous Multi-threading: Maximizing On-Chip Parallelism. In: Proceedings of the 22nd International Symposium on Computer Architecture (ISCA 1995), June 1995, pp. 392–403 (1995)
9. Davis, J.D., Laudon, J., Olukotun, K.: Maximizing CMP Throughput With Mediocre Cores. In: Malyshkin, V. (ed.) PACT 2005. LNCS, vol. 3606, pp. 51–62. Springer, Heidelberg (2005)
10. Alameldeen, A.R., Mauer, C.J., et al.: Evaluating Non-deterministic Multi-threaded Commercial Workloads. In: Proceedings of the 5th Workshop on Computer Architecture Evaluation Using Commercial Workloads (CAECW 2002) (February 2002)
11. Barroso, L., Gharachorloo, K., McNamara, R., et al.: Memory System Characterization of Commercial Workloads. In: Proceedings of the 25th International Symposium on ComputerArchitecture (ISCA 1998), June 1998, pp. 3–14 (1998)
12. McGhan, H.: SPARC64 VI: Ready for Prime Time. Microprocessor Report, In-Stat/MDR (October 23, 2006)

Synchronization Mechanisms on Modern Multi-core Architectures

Shaoshan Liu and Jean-Luc Gaudiot

Department of Electrical Engineering and Computer Science
University of California, Irvine
{shaoshal,gaudiot}@uci.edu
http://pascal.eng.uci.edu

Abstract. While the semiconductor industry has provided us with powerful systems for personal supercomputing, how to efficiently harness the computing power of these systems still remains a major unsolved problem. This challenge must be approached by simultaneously solving the synchronization problem and the parallel programmability problem. This paper reviews the synchronization issues in modern parallel computer architectures, surveys the state of the art approaches used to alleviate these problems, and proposes our Request-Store-Forward (RSF) model of synchronization. This model splits the atomic synchronization operations into two phases, thus freeing the processing elements from polling operations. Finally, we show how we could learn from nature and improve the overall system performance by closely coupling peripheral computing units and functional units.

1 Introduction

Due to diminishing returns in single-core design in recent years, all major competitors in the semiconductor industry have all announced a multi-core designs. For instance, Niagara is a Chip-Multithreading (CMT) processor from SUN that features eight cores, each able to simultaneously executing four threads [8]. Cyclops is a Chip Multi-processors (CMP) design from IBM, which contains 160 thread units and is under active research [9]. Intel's Teraflops Research Chip is another many-core design that packs 80 cores on a single die [10]. For data storage, Niagara utilizes a two-level cache system. With this architecture, data communication and synchronization take place through the shared cache and may rely on its cache coherence mechanism. On the other hand, both Cyclops and the Teraflop Research Chip utilize a uniform address space distributed memory system, and data communication and synchronization take place through an on-chip network, which allows message passing from one core to another.

While the industry has provided us with the tools for personal supercomputing, how to utilize the enormous computing power from tens or even hundreds of computing cores still remains a difficult problem. For instance, although Intel has demonstrated that their Teraflop research chip is able to deliver more than one trillion floating-point operations per second, it is also implied in their white paper that this chip is not destined for general-purposed computing [11]. The major challenge facing the semiconductor industry today is not only to develop high-performance chips, but

© Springer-Verlag Berlin Heidelberg 2007

more importantly, to harness the computing power of these systems. Specifically, to achieve high performance, this challenge has to be addressed at the software and the architectural level instead of only at the software level. To achieve this goal, two problems, namely programmability and synchronization, need to be addressed in concert. Programmability can be roughly defined as the easiness of mapping parallel real-world applications onto parallel architectures, whereas synchronization ensures the correctness of parallel execution by enforcing dependencies between instructions. A good programming model demands the support of a good synchronization mechanism. A fine-grain parallel programming model would not be useful if the corresponding synchronization mechanism were coarse-grain. As a result, research in parallel computing should be following three steps: the design of a good synchronization mechanism, the design of a good parallel programming model, and the synthesis of the two.

The goal of this paper is to review some of the several approaches that have been proposed in recent years to address the synchronization problems. We also propose a split-phase Request-Store-Forward synchronization model to address the challenges faced by synchronization mechanisms on modern parallel computing architectures. The schemes we survey include cache and register-based synchronization mechanisms aiming to reduce the synchronization latency by shortening the distance between the processing elements and the location where synchronization operations take place [4, 16], transactional memory which attempts to reduce synchronization latency through speculative execution [1, 2], synchronization buffer which approaches aggregate synchronization states in a buffer, thereby increasing memory utilization efficiency [6, 7], and the Synchronized Pipelined Parallelism Model is a software synchronization model that provides a good temporary solution for the synchronization problems [5]. While these approaches each aims at improving on one aspect of the synchronization problems, an ideal hardware-based model has yet to appear.

2 Problems with Conventional Synchronization Mechanisms

Conventional synchronization mechanisms are constrained by long synchronization latency, resource contention, as well as synchronization granularity. Synchronization latency can be as high as hundreds of cycles if synchronization takes place through a shared memory system. The atomicity of synchronization operations often forces the requesting processors to stall during the operation. For fine-grain synchronization operations, this latency dominates the execution time. In addition, the polling mechanism introduces serious contention problems. When multiple processes are attempting to lock a shared variable in memory, only one process will succeed, while all other attempts are strictly overhead of several kinds: 1) *Performance Overhead*: in the busy-waiting stage, the requesting processors are tied up sending out polling messages. In addition, contention may lead to deadlock situations that require extra mechanisms for deadlock prevention, which further degrade system performance. 2) *Communication Overhead*: until the lock is obtained, the requesting processors need to continuously place lock requests on the system bus. In a shared-memory multiprocessor system with spin-lock synchronization, the number of synchronization requests grows nonlinearly with the number of contending processes, making the system not scalable [3]. 3) *Power*

Consumption Overhead: a side effect of the previous two overheads. Furthermore, with a coarse-grain synchronization mechanism, a data structure instead of a word needs to be locked for synchronization although only one word is under synchronization at any instance of parallel execution. This not only results in unnecessary serialization of the access to data structures but may also introduce cache contention problems.

An ideal synchronization mechanism for Chip-Multi-Processors, especially many-core systems, should possess the following characteristics:

- *Fine-Grain*: the degree of parallelism that can be exploited in a parallel computing system is limited by the granularity of synchronization.
- *Low-Latency*: a synchronization mechanism is useful only if its latency does not dominate the execution time.
- *Contention Free*: to efficiently synchronize the operations of different processing elements, the ideal synchronization mechanism should be contention free.
- *Scalability*: the ideal synchronization mechanism should be able to scale as the number of cores in the system increases.
- *Flexibility*: the ideal synchronization mechanism should be application-independent.

In this paper, we propose the Request-Store-Forward (RSF) model of synchronization for modern parallel computing systems. This model splits atomic synchronization operations into two phases: upon the arrival of synchronization requests, it stores the synchronization state and allows the requestors to continue with other tasks; when the synchronization operation finishes, it forwards a notification message as well as the requested data to the requestor.

3 Cache and Register-Based Synchronization Mechanisms

To address the synchronization latency problem, some mechanisms have been proposed where synchronization operations would take place at the shared cache level, or even at the register level [16, 17, 18, and 4]. For instance, Yamawaki *et al.* [4] propose to utilize a cache coherence mechanism to perform synchronization. In their design, all on-chip caches are split into global cache, which stores shared data, and local cache, which stores data only local to the processing element. All the shared caches are able to communicate with each other through a cache-to-cache direct access bus.

In this design, two fundamental operations, store word with synchronization (*sws*) and load word with synchronization (*lws*) are utilized. The *sws* operation stores a word as well as a counter to keep track of the number of times the stored data would be consumed. The *lws* operation loads a word and decrements the counter by one. For producer-consumer synchronization, the producer utilizes *sws* to store the item produced and the consumers utilize *lws* to load the data produced. Thus, when the counter associated with the data produced becomes zero, it signals the completion of the producer-consumer operation. For mutual exclusion, the counter of the shared variable is set to 1, and all requesting processing elements issue an *lws* operation, but only one *lws* operation would succeed and decrement the counter to zero. When other requestors observe that the counter value is zero, they stall and wait for the succeeding thread to exit the critical section. Upon completion, the succeeding thread issues an *sws* operation to reset the counter to one, which signals other requestors to resume. For barrier synchronization, the counter of the synchronization variable is set to the number of threads under barrier synchronization. Upon the arrival of each thread, an *lws* operation is issued to decrement the counter. When the counter value becomes zero, all threads are released from the barrier.

In this approach, each time a synchronization operation is issued, this operation as well as the data are broadcasted through the dedicated cache-to-cache direct access bus. The global cache of each processing unit then updates its data with the broadcasted data and adjusts the counter value according to the synchronization operation. This approach combines communication and synchronization with coherence maintenance, thus resulting in efficient data communication and synchronization. Nevertheless, this design is not scalable and may not fit many-core architectures. The broadcasting of synchronization operations as well as of the data produced creates a bus bandwidth requirement that may grow linearly with the number of cores in the system. This requirement is very hard to meet in a large-scale parallel computing system that contains hundreds of cores.

4 Transactional Memory

Motivated by the high degree of parallelism in database transactions, transactional memory utilizes speculation to reduce synchronization latency [1, 2]. In this context, a transaction is defined as a chunk of instructions that are guaranteed to execute only as an atomic unit. Each transaction produces a block of writes called the write state which will be committed to the shared memory after the completion of the execution. Although there is no parallelism *within* a transaction, parallelism does exist *between* transactions. With this design, an application program is partitioned into transactions, and each processor is assigned one transaction. Then all processing elements in the system can start speculative execution simultaneously by assuming that the current transaction does not depend on the write states of other transactions. When a transaction completes, its write state is broadcasted throughout the system. Then other transactions can listen to this write state and determine whether they have used data that has been modified by it. If not, then parallelism is successfully exploited through thread level speculation. Otherwise, the current transaction has used data modified by other transactions, which causes the execution to be incorrect due to inconsistent memory states. In this case, the current thread has to roll back to its previous state, invalidate all its execution results, and restart the transaction.

In this model, system consistency is maintained by imposing a sequential ordering only between transaction commits. Thus, no serialization of instructions is required for execution of transactions, which allows thread level speculation. If parallelism were successfully exploited through speculative execution, then it would create the appearance that synchronization has zero latency. In addition, this scheme eliminates the expensive cache coherence mechanism in that the processor cores are able to store both the unmodified data lines and the speculatively modified data lines. Thus, it can be implemented with or without a cache system. Furthermore, when inserting transaction boundaries in application programs, the programmers do not have to worry about data dependency between instructions, which makes parallel programming much easier. Nevertheless, to efficiently utilize this mechanism, the programmers still have to carefully choose the size of the transaction. If the transaction is too large, then it may result in frequent rollbacks, which reduces system efficiency. On the other hand, if the transaction is too small, then synchronization overhead may dominate execution time.

Despite all the benefits it provides, the transactional memory scheme is not scalable for two reasons. First, it requires system-wide bus arbitration for commit permission. When a transaction completes its execution, its write state needs to be committed through the system bus. In a many-core system, system bus resource contention is incurred when multiple cores attempt to simultaneously access the system bus. This may result in a large arbitration latency, which degrades system performance. Second, the broadcasting of write states demands a very high bus bandwidth and causes large communication overheads. In addition to the scalability problems, speculation failures in this scheme often lead to costly rollbacks, which are inefficient in terms of performance as well as power consumption.

5 Synchronization Buffer

Based on the observation that only a small fraction of memory locations are actively participating in synchronization at any instance of parallel execution, Zhu *et al.* [7] proposed the Synchronization State Buffer (SSB), which records and manages the states of frequently synchronized data at the word level. When a synchronization operation is issued, it first checks whether the word under synchronization already exists in the synchronization buffer. If the word under synchronization exists in the synchronization and is being locked, then the requesting processing element gets a failure message and attempts to lock the word again. Otherwise, a buffer line is allocated for the word and the requesting processing element can move on with its execution. For mutual exclusion, this scheme provides two basic operations, *read lock* and *write lock*. For producer-consumer synchronization, it allows both single-producer-single-consumer and single-producer-multiple-consumer synchronization. For the single-producer-multiple-consumer case, instead of requiring the consumers to continuously issue consume requests until the data under synchronization has been produced, this scheme records all consume requests in the synchronization buffer and notifies the consumers to consume data upon the arrival of the data produced. As a result, the synchronization contention problem in producer-consumer synchronization is eliminated.

By aggregating the synchronization states of the whole memory into a small buffer, this design eliminates the synchronization-state-tracking full/empty bit from memory, thus resulting in a more efficient use of memory space. Also, the synchronization buffer records the synchronization state of individual words, thus providing fine-grain synchronization capability that prevents unnecessary serialization of the accesses to data structures. In addition, this mechanism enables split-phase operations for producer-consumer synchronization, and thus, to some degree alleviates the resource contention problem introduced by the conventional polling synchronization mechanism. However, for mutual exclusion synchronization, the proposed mechanism fails to record the synchronization states, thus leaving the contention problem unsolved.

To address the synchronization contention problem, Monchiero *et al.* [6] have proposed the Synchronization Operation Buffer (SOB). SOB is similar to what we propose in this paper: it utilizes a synchronization buffer to intercept all synchronization requests, record these requests, and then notify the requesting processes when the lock becomes available. By recording synchronization requests in memory, each processing element only has to issue one synchronization request for each synchronization operation, thereby eliminating the synchronization contention problem. It has been demonstrated in this paper [6] that this approach not only results in a performance improvement but also reduces system power consumption. Although this scheme is theoretically simple and effective, only mutual exclusion synchronization is considered and no detailed mechanisms for the implementation of wait queue storage and data forwarding have been provided.

In both designs, the buffer size is fixed. When the buffer becomes full, the system needs to utilize software synchronization routines, which could take hundreds of cycles to complete. Note that the designs of the synchronization buffer are for fine-grain synchronization, but that once the software routines are utilized, the synchronization latency will dominate the execution time and consequently, the benefit of fine-grain synchronization will be lost. On the other hand, some applications have a low degree of parallelism, thus the buffer is under-utilized for most of the time, resulting in a waste of on-chip hardware resources.

6 Synchronized Pipelined Parallelism Model

Unlike the previous hardware mechanisms, the Synchronized Pipelined Parallelism Model (SPPM) proposed by Vladamani *et al.* [5] is a software-based synchronization mechanism that exploits parallelism by restructuring applications into a pipeline of producers and consumers. This approach targets CMP and SMT systems with multiple levels of shared cache. The authors suggest that the commonly used spatial decomposition parallel programming model overlooks the temporal parallelisms which otherwise exists in shared data, thus resulting in an inefficient use of the shared caches and the memory interface. As the number of application threads increases in a parallel computing system, the effective size of the shared cache seen by each processor is reduced, which would result in more cache misses. Under these conditions, when a producer produces a data item, this data item may soon be replaced out of the shared cache, forcing the consumers to go to the shared memory to fetch this data item, resulting in long synchronization latency.

The essence of this approach is to force the data under synchronization to stay in the shared cache until it has been consumed. As a result, the number of cache misses incurred by consumers can be reduced and consumers can fetch data with a low latency. When the working set of the application is too large to fit into the shared cache, this approach sets the shared cache as a medium for the flow of data from the producers to the consumers, thus facilitating efficient communications between the producer and the consumer. Under this synchronization model, there are three states between the producer and the consumer. First, when both the producer and the consumer are actively producing and consuming data, they are in the state "*Processing Data.*" In this state, parallelism is successfully exploited by concurrently running the producer and the consumer. Second, when the consumer consumes faster than the producer produces, it switches to the state "*Waiting for Producer*" to allow for the producer to catch up. In this state, the consumer is stalled so that parallelism between the consumer and the producer cannot be exploited; on the other hand, the consumer would not be able to incur unnecessary cache misses, resulting in an efficient utilization of the shared cache. Similarly, when the producer produces faster than the consumer consumes, it is stalled and switches to state "*Waiting for Consumer*" to allow for the consumer to catch up.

Simulation results have demonstrated that compared with the traditional spatial decomposition programming model, SPPM is able to exploit parallelism without introducing high memory bus utilization overheads and additional cache misses. Thus, this approach can be seen as a great temporary solution to the problem of parallel processing synchronization. However, this approach suffers from the legacy code problem: to gain any benefits, the software needs to be re-written following SPPM, making this approach impractical. One counter argument is that this programming model will be implemented by the compiler, which is able to automatically restructuring any parallel applications into a pipeline of producers and consumers; however this has yet to be implemented. In addition, software synchronization routines introduce high performance overheads (each software operation can take hundreds of cycles to complete), thus preventing the exploitation of fine-grain parallelism.

7 The Request-Store-Forward (RSF) Synchronization Model

To meet the needs of modern parallel computing systems, we propose the Request-Store-Forward (RSF) model of fine-grain synchronization based on the idea of I-Structures. An I-Structure is a data structure proposed by Arvind *et al.* [21]. In contrast to the traditional atomic data transaction operations, I-Structure supports split-phase transaction, thereby reducing communication overheads and alleviating contention problems. It has been demonstrated by Lin *et al.* in [19, 20] that a software version of I-Structure can significantly reduce network traffic while providing efficient producer-consumer synchronization. Like I-Structure, the RSF model is a split-phased mechanism. Instead of keeping track of the synchronization states by attaching a full/empty bit to each word in memory, RSF utilizes a synchronization buffer to keep track of all synchronization operations. It follows the following steps:

- *Request*: when a process demands synchronization on a shared variable, it sends a synchronization request to the synchronization buffer.
- *Store*: If the request cannot be served immediately, instead of setting the requestors in polling state, the synchronization buffer records and orders all synchronization requests in a FIFO queue. Further, the requesting process switches to sleep mode. By recording the synchronization states, contention is eliminated.

- *Forward*: after the current synchronization operation is done, the synchronization controller removes a request from the wait queue and sends a notification message as well as the requested data to the requestor, thereby completing synchronization and communication in one operation.

The proposed RSF model targets many-core architectures with either shared cache systems or uniform address space distributed memory systems. It meets all the requirements for an ideal hardware synchronization mechanism. First, it is a fine-grain word-level synchronization mechanism. Each line in the synchronization buffer is able to keep track of the synchronization states of individual words in memory. Second, by recording the synchronization states, the contention problem is eliminated. Third, all synchronization operations take place in the shared cache or on-chip memory so that the synchronization latency can be reduced to less than ten cycles. Fourth, instead of dedicating a fixed-size buffer for synchronization, we propose to dynamically allocate the shared cache or on-chip memory space for the synchronization buffer so that the buffer can be flexible to meet the needs of different applications. Last, in the producer-consumer synchronization, the forwarding of data to multiple consumers may restrict the scalability of multi-core systems. To address the scalability problem, a scalable forwarding scheme is proposed in the RSF model.

7.1 Dynamic Buffer Storage

For the design to be flexible, we propose to implement the synchronization buffer inside the cache or on-chip memory and dynamically adjust its size to fit the needs of different applications. For applications that require a very low degree of synchronization, such as OS multiprogramming workloads, buffer space can be freed up for data storage. For applications that require a large degree of synchronization, buffer space can be allocated to prevent the use of software routines. The dynamic allocation algorithm has to meet several requirements. First and most importantly, there should be enough space allocated for the synchronization buffer so that no software routine is needed, thereby guaranteeing that the synchronization latency is small. Second, since the buffer is implemented in cache, the cache contention situation needs to be taken into consideration. Third, the algorithm should be simple so that it can be implemented in hardware.

Algorithm for dynamic allocation:
```
if buffer full
   grow by G;
if there is contention and buffer not full for n cycles
    shrink by 1;
```

When the synchronization buffer becomes full, it is grown by G lines, where G reflects the current cache contention situation in the system. On the other hand, the buffer size is shrunk to free up storage space only when the cache contention is high and the buffer has been underutilized for *n* cycles. The design parameter *n* represents the sensitivity of the allocation logic. If this logic is too sensitive (*n*=1), it might result in oscillatory actions of buffer line allocation and deallocation, which introduces extra power consumption due to excessive hardware operations. If the logic is not sensitive enough, then it acts as a fixed-size buffer, therefore losing its flexibility.

7.2 Synchronization Requests

To provide effective synchronization, both Mutual Exclusion Synchronization (MES) and Producer-Consumer Synchronization (PCS) [12] are required. For MES, each time only one of the contending processes is allowed to update a shared variable, so that there is no parallelism existing between requests. For PCS, the consumers must wait until the producer generates the required value, otherwise, the operation will be incorrect. But once the data is produced, all consumers can simultaneously consume that data. In the RSF model, MES is implemented through three operations:

- *Write Lock:* *(return value) = wrlock (address, value)*
- *Read Lock:* *(return value, data) = rdlock(address)*
- *Unlock:* *unlock (address)*

When a MES request is issued, the synchronization controller first checks the synchronization buffer. If the requested data is not in the synchronization buffer, then one line is allocated to the buffer to record the current synchronization state. If the requested data is already in the synchronization buffer with mode *No Lock*, the mode block is updated to either *Write Lock* or *Read Lock* and thread ID is updated to the ID of the requestor. In both cases, the synchronization operations will succeed. For a write request, *value* is stored in location *address* and *return value* success is sent back to the requestor. For a read request, a "*return value success*" as well as the requested *data* in location *address* is returned to the requestor. With this approach, both the lock acquisition and the read/write operation can be accomplished in one operation.

Otherwise, if the requested data is already present in the synchronization buffer and is in the write mode, then the current request is added to the wait queue and a "*return value failure*" is returned to the requestor. Upon receiving the failure message, the requestor goes into the sleep mode for power saving. When the requested item is in the read mode, the requestor with a write request receives a "*return value failure*" and goes into the sleep mode. Further, its thread ID is added to the wait queue. Note that if the requested item is in the read mode, a read lock request will succeed and receive a "*return value success*" along with the requested *data*. This is because multiple read requests do not conflict with each other and should be served simultaneously.

After the synchronization operation has been completed, an explicit unlock operation is issued to release the lock. At this stage, if the wait queue is not empty, then the synchronization controller takes an item from the wait queue and executes the synchronization operation. On the other hand, if the wait queue is empty, then *mode* is set to No Lock. Unlike conventional atomic synchronization operations, the proposed approach splits the synchronization operations into two phases if the request fails, thereby completely eliminating the contention problem.

PCS is implemented in two operations:

- *Produce:* *(return value) = produce (address, value, consumer count)*
- *Consume:* *(return value, data) = consume (address)*

As opposed to MES, PCS does not require an explicit unlock message to release the lock. Instead, it utilizes the counter in synchronization buffer. If the producer produces before the consumers consume, then it allocates a line in the synchronization buffer and sets counter to *consumer count*, or the number of consumers. Each time a consume request is issued, the counter is decremented by one and the *data* along with *return value* success are returned to the requestor. When the counter hits zero, the synchronization buffer line is released and set to the No Lock mode. On the other hand, if a consumer issues a consume request before the requested data has been produced, a synchronization buffer line is allocated with the counter set to 1, and also a wait queue is initialized to store the requestor's ID. Meanwhile, the *return value* failure is returned to the requestor, which instructs the requestor to go into sleep mode. Subsequent consume requests each increment the counter and append their IDs to the wait queue, then go to sleep. Once the producer produces the data item, it stores the data item *value* in location *address* and sets counter to the number of consumers, or *consumer count*. Then the consumers are removed from the wait queue and the requested data item is forwarded to all consumers. Each time a consumer is removed from the queue, the counter is decremented by one. When the counter hits zero, the synchronization buffer line is released and set to the No Lock mode. Similarly, PCS operations are split-phase, thereby avoiding the contention problem.

7.3 Storage of Wait Queue

When the synchronization requests cannot be served immediately, these requests are stored in a wait queue so as to eliminate contention and the pointer to the wait queue is stored in the synchronization buffer. These wait queues should be implemented in the shared cache or in the on-chip memory to keep the synchronization latency low. If the wait queues were stored in the lower-level memory, then the thread ID fetch time would dominate execution time. One could argue that these wait queues place a large burden on the shared cache. Nevertheless, as mentioned in the previous sections, once a thread ID is added to the wait queue, the requesting thread goes into the sleep mode. Therefore, in the worst case, the size for all wait queues should be $n*p$, where n represents the number of threads supported in the system and p represents the size of a thread ID.

7.4 Data Forwarding

When a lock is released and its corresponding wait queue is not empty, a thread ID is removed from the wait queue. Then, a message is forwarded to wake up the thread and notify it that it has successfully obtained the lock. This procedure is repeated until the wait queue becomes empty. This Sequential Centralized Forwarding Scheme (SCFS) works well for MES due to the lack of parallelism among mutual exclusion locks. In contrast, parallelism does exist in PCS such that all consumers are able to simultaneously consume a data item. As a result, SCFS introduces two problems for PCS. First, it creates a serious resource contention problem on the shared memory. It requires the memory controller to be tied up for a long time to handle the synchronization operations on only one memory location. Second, this approach performs forwarding in a sequential order, and thus the last consumer in the wait queue will be getting the requested data after a long delay.

For system scalability, we propose the Tree Forwarding Scheme (TFS). The wait queue is stored as a binary tree, and implemented as an array. When the data is produced, the data along with the wait queue are forwarded to the first consumer. Then the first consumer extracts the left sub-tree and the right sub-tree from the wait queue and stores them in two new arrays. Next, it forwards the data item as well as the left sub-tree of the wait queue to its left child; and the data along with the right sub-tree to its right child. Subsequently, the left and the right child repeat this procedure until the leaf nodes are reached. The resource contention problem is addressed in this approach because the memory controller only needs to send out one copy of the data item. Second, the latency problem is also alleviated by finishing the forwarding operation in $\log_2 n$ steps instead of n steps, where n is number of consumers.

An analytical model can be used to evaluate the resource contention on the memory controller and the latency for the forwarding mechanism. In this analysis, *n* represents the number of consumers in the wait queue, *p* represents the size of each thread ID, *w* is the size of the word under synchronization, and *t* represents the inverse of bus bandwidth, or the time for sending out a unit of data. Notice that the word size *w* should be far greater than the thread ID size *p*. In a typical system, each word has a size 32 bits or 64 bits whereas the thread ID size *p* equals $\log_2 m$, where *m* is the number of threads supported in the system. For a 100 thread system, the size of p is only 7 bits. With SCFS, the memory controller is responsible to forward the data item to all consumers in a sequential order. Thus, the workload on the memory controller is to send out $n * w$ units of data, and the forwarding operation takes $n * w * t$ units of time to finish. With TFS, the memory controller forwards the data item and the wait queue to the first consumer, and the rest of the forwarding work is done in a distributed and parallel fashion. The resulting workload on the memory controller is $n * p + w$ units of data, but forwarding can be done in $\log_2 n$ steps. As a result, as demonstrated by equations (2), (3), and (4), for a large n, TFS not only prevents the resource contention problem introduced in SCFS, but also outperforms both SCFS in forwarding latency.

$$p << w \tag{2}$$

$$n * p + w << n * w \tag{3}$$

$$\sum_{k=0}^{\log_x n} [(\frac{1}{2})^k * n * p * t + 2 * w * t] << n * w * t \tag{4}$$

8 Heterogeneous Multicore: A Lesson From Nature

In order to efficiently implement the RSF model of synchronization, the memory controller needs to dynamically allocate memory space for the synchronization buffer, update the synchronization buffer as synchronization operations take place, and forward data and synchronization messages to different cores. Especially for many-core systems, these tasks may impose heavy burden on memory controllers. Can the current implementation of memory controllers handle all these tasks? Or, in the first place, should these tasks be implemented in memory controllers? We could seek a hint to the answer from nature.

Biological systems are highly complicated systems composed of many different cell types, with each focusing on one function. For instance, the two major components of the human brain are gray matters and white matter. Gray matter cells are powerful computing units responsible for computation-intensive tasks, whereas white matter cells are less computationally powerful components responsible for communication. The intermingling of these two matters enables human brains to efficiently perform extremely complicated computations. While the major competitors of the computing industry are proposing homogeneous many-core systems, the authors believe that heterogeneous multi-core systems, similar to those found in nature, will eventually outperform homogeneous multicore systems. In fact, it has already been demonstrated that heterogeneous multi-core systems deliver similar performance with lower power consumption [13].

We advocate that peripheral computing units should be closely coupled with functional units to maximize system performance. As proposed in this paper, by handling synchronization in the computing units embedded in the memory system, the major computing cores can be freed up from synchronization overheads. Indeed, there are two benefits to embed computing units in the memory system: first, peripheral functions, such as synchronization, can be offloaded from the computing cores. Second, some simple computations can be done in memory without suffering from any communication overheads. For example, test-and-set is a commonly used function for synchronization. The processing element that successfully obtains the lock fetches the data from memory, performs an increment function, and stores this data back into memory. Although it is merely a simple addition operation, it actually requires two memory requests, one read and one write, which result in high communication overheads. One way to eliminate these overheads is to have the increment done in memory. Processor-In-Memory (PIM) technology embeds simple computing units in memory [14, 15] and it is certainly one option to offload peripheral functions, such as synchronization, from the main computing cores.

9 Conclusions and Future Work

How to efficiently harness the computing power of many-core systems is the main challenge now faced by the semiconductor industry. This challenge should be approached by solving the programmability problem as well as the synchronization problem. This paper has reviewed the synchronization problems in modern parallel computer architectures and has proposed the Request-Store-Forward (RSF) model of synchronization. This model splits the atomic synchronization operations into two phases, thus freeing the processing elements from being tied up for polling operations. We have also demonstrated that this model is fine-grain, low-latency, contention-free, flexible, and scalable, and that it thus fits well modern multi-core computing systems.

To implement the RSF model of synchronization, some computing tasks need to be offloaded from the computation cores to the memory system. Although this may increase the burden on the memory controllers, by closely coupling peripheral computing units and functional units, such as memory, the overall system performance may be improved. Nevertheless, more work needs to be done to investigate how different levels of coupling may impact overall system performance, implementation area, as well as power consumption. In addition, further research will be needed to evaluate how the proposed synchronization scheme impacts the cache performance, especially cache contention

Acknowledgements

This work is partly supported by the National Science Foundation under Grant No. CCF-0541403. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. Hammond, L., Wong, V., Chen, M., Carlstrom, B.D., Davis, J.D., Hertzberg, B., Prabhu, M.K., Wijaya, H., Kozyrakis, C., Olukotun, K.: Transactional memory coherence and consistency. In: ISCA 31 (June 2004) 102–113 (2004)
2. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: ISCA 20 (May 1993), 289–300 (1993)
3. Hennessy, J.L., Patterson, D.A.: Computer Architecture-A Quantitative Approach. Morgan Kaufmann, San Francisco (2006)
4. Yamawaki, A., Iwane, M.: Coherence Maintenances to realize an efficient parallel processing for a Cache Memory with Synchronization on a Chip-Multiprocessor. In: proc of ISPAN 8 (2005)
5. Vadlamani, S., Jenks, S.: Architectural Considerations for Efficient Software Execution on Parallel Microprocessors. In: proc of IPDPS 21 (2007)
6. Monchiero, M., Palermo, G., Silvano, C., Villa, O.: An Efficient Synchronization Technique for Multiprocessor Systems on-Chip. ACM SIGARCH Computer Architecture News 34(1) (March 2006)
7. Zhu, W., Sreedhar, V.C., Hu, Z., Gao, G.R.: Synchronization State Buffer: Supporting Efficient Fine-Grain Synchronization on Many-Core Architectures. In: ISCA, 34 (June 2007)
8. Kongetira, P., Aingaran, K., Olukotun, K.: A 32-way multithreaded Sparc processor. IEEE Micro, 40–47 (March/April 2005)
9. Denneau, M., Warren Jr., H.S.: 64-bit Cyclops: Principles of operation (April 2005)
10. Vangal, S., Howard, J., Ruhl, G., et al.: An 80-tile 1.28TFLOPS network-on-chip in 65nm CMOS. In: Procs. of ISSCC 2007 (February 2007)
11. Held, J., Bautista, J., Koehl, S.: From a Few Cores to Many: A Tera-Scale Computing Research Review, White Paper, Intel Research2006 (2006)
12. Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J, Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., Yelick, K.A.: The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report No. UCB/EECS-2006-183 (December 2006)

13. Kumar, R., Tullsen, D.M., Jouppi, N.P.: Heterogeneous Chip Multiprocessors. Computer, IEEE Computer Society (2005)
14. Jasionowski, B. J.,Lay, M. K., Margala, M.: A Processor-In-Memory Architecture for Multimedia Compression. IEEE Transaction on VLSI Systems (April 2007)
15. Sterling, T.L., Zima, H.P.: Gilgamesh: A Multithreaded Processor-In-Memory Architecture for Petaflops Computing. ACM/IEEE Supercomputing Conference (2002)
16. Breach, S.E., Vijaykumar, T.N., Sohi, G.S.: The Anatomy of the Register File in a Multiscalar Processor. In: Proc. MICRO-27 (December 1994)
17. Keckler, S.W., Dally, W.J., Maskit, D., Carter, N.P., Chang, A., Lee, W.S.: Exploiting Fine-Grain Thread Level Parallelism on the MIT Multi-ALU Processor. In: Proc. 25th ISCA (June 1998)
18. Kobayashi, R., Iwata, M., Ogawa, Y., Ando, H., Shimada, T.: An On-Chip Multiprocessor Architecture with a Non-Blocking Synchronization Mechanism. In: Proc. 25th EUROMICRO (1999)
19. Lin, W.-Y., Gaudiot, J.-L., Amaral, J.N., Gao, G.R.: Performance Analysis of the I-Structure Software Cache on Multi-Threading Systems. In: Proc. of the 19th IEEE International Performance, Computing, and Communication Conference (IPCCC ’00) (February 2000)
20. Lin, W.-Y., Amaral, J.N., Gaudiot, J.-L., Gao, G.R.: Caching Single-Assignment Structures to Build a Robust Fine-Grain Multi-Threading System. In: Proc. of the 14th International Parallel and Distributed Processing Symposium (IPDPS ’00) (May 2000)
21. Arvind, Nikhil, R.S., Pingali, K.K.: I-structures: data structures for parallel computing. ACM Transactions on Programming Languages and Systems (TOPLAS) 11(4), 598–632 (1989)

Concerning with On-Chip Network Features to Improve Cache Coherence Protocols for CMPs

Hongbo Zeng[1,2], Kun Huang[1,2], Ming Wu[1,2], and Weiwu Hu[1]

[1] Key Laboratory of Computer System and Architecture,
Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
[2] Graduate University of the Chinese Academy of Sciences, Beijing, China
{hbzeng,huangkun,wuming,hww}@ict.ac.cn

Abstract. Chip multiprocessors (CMPs) with on-chip network connecting processor cores have been pervasively accepted as a promising technology to efficiently utilize the ever increasing density of transistors on a chip. Communications in CMPs require invalidating cached copies of a shared data block. The coherence traffic incurs more and more significant overhead as the number of cores in a CMP increases. Conventional designs of cache coherence protocols do not take into account characteristics of underlying networks for flexibility reasons. However, in CMPs, processor cores and the on-chip network are tightly integrated. Exposing the network features to cache coherence protocols will unveil some optimization opportunities. In this paper, we propose distance aware protocol and multi-target invalidations, which exploit the network characteristics to reduce the invalidation traffic overhead at negligible hardware cost. Experimental results on a 16-core CMP simulator showed that the two mechanisms reduced the average invalidation traffic latency by 5%, up to 8%.

1 Introduction

The wide availability of chip multiprocessors (CMPs) has demonstrated their capabilities to efficiently utilize the ever increasing number of transistors. On-chip networks [1], which have been used to interconnect multiple processing elements on a chip, is a promising technology that targets the delay and power consumption problems of global wires [2].

Like distributed shared memory (DSM) machines, CMPs maintain data coherence by cache coherence protocols. Conventionally, the cache coherence protocol and the network are considered as two non-related components in a DSM system. The design concepts and optimization techniques of protocols do not take into account the characteristics of underlying networks. Likewise, network optimizations concentrate on reducing communication latency without consciousness of up-level protocols. Considerable flexibility is achieved as the protocols can be deployed on a wide variety of networks. When CMPs are concerned, however, processor cores and the interconnect network are tightly integrated and, in addition, parameters of the on-chip network are determined at design time. These

© Springer-Verlag Berlin Heidelberg 2007

natures motivated us to expose network characteristics to the protocols so as to explore new approaches to improve the performance of CMPs.

As more and more cores are placed on future CMPs, one practical design methodology is to assemble tiles of same-sized cores into an array by an on-chip network as depicted in figure 1 [3,4]. Each core contains a fraction of the L2 cache which is shared by multiple cores though physically distributed. Compared with private L2 caches, shared caches have the advantage of allowing more capacitance for each core and avoiding duplicated copies of the same cache line in private caches. Directory-based cache coherence protocol [5] is a more appropriate option for maintaining coherence of data copies among L1 caches than bus-based snoopy protocol for scalability reasons. A directory keeps track of the global coherence states and the sharers' identifications of all cache lines in L2 cache. Before a processor core can modify the data of a cache line, it must send a read exclusive request to the directory which invalidates remote copies of that cache line. When the directory receives acknowledgments from all the sharers, it replies the requester with write grant. Figure 2 illustrates the communications incurred. The invalidation process introduces high overhead and its significance is growing as the number of cores in a CMP increases. This paper presents two mechanisms exploiting the network features to reduce the invalidation overhead at negligible hardware cost.

Following the discussion above, traditional designs of directory-based protocols may result in sub-optimal operations as they have little knowledge of the network. As shown in figure 1, where a 2-D mesh network with XY routing algorithm [6] is applied in a CMP, if the directory in core A needs to invalidate data

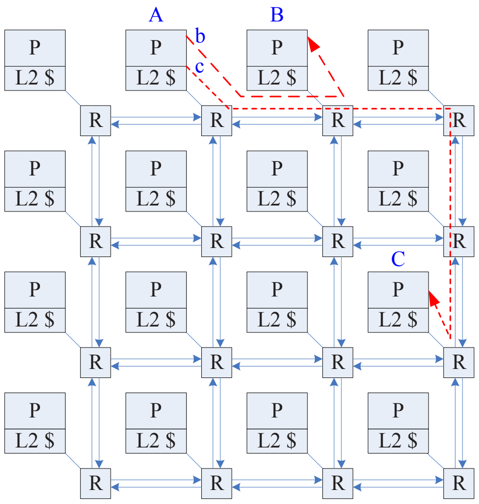


Fig. 1. An example architecture of a tiled CMP. Each core contains private L1 caches and a fraction of the shared L2 cache. Multiple cores are connected by a 2-D mesh routing on-chip network (boxes marked with *P* represent processor cores and boxes marked with *R* represent routers).

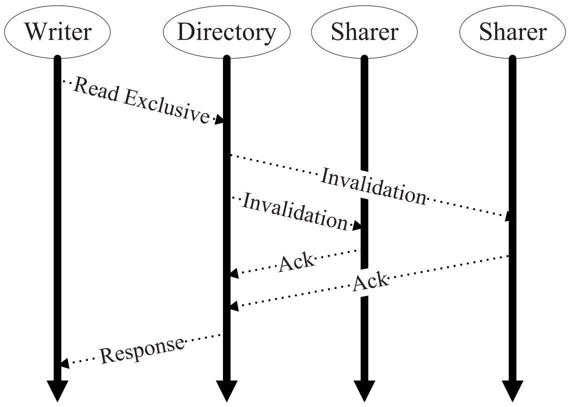


Fig. 2. The directory invalidates all the shared copies before replying the requestor

copies in L1 caches of cores B and C, the protocol may first send out the invalidation message b for B and then message c for C. The problem is that c takes more hops – so more clock cycles – than b to complete, but it is sent later, which increases the overall delay. Lacking the information of the network (C is further than B) causes the protocol to make sub-optimal schedules. We propose that the coherence protocols designed for CMPs should comprehensively consider the distance between cores.

Observations showed that the number of cores sharing a cache line is usually larger than one. Conventional approaches, which send one invalidation message for each sharing core respectively, would create bursts into the network and cause significant contention. This would bring negative impact on performance. We extend the above optimization technique to compact multiple invalidation requests into one network packet, which effectively lower the network load.

Using a cycle-accurate execution-driven simulator of a 16-core CMP, we evaluate our proposed mechanisms with a set of scientific computation workloads. We find that the two mechanisms together reduced the average overhead of invalidation traffic by 5%, up to 8%.

This paper is organized as follows: Section 2 explains the distance aware optimization technique; Section 3 extends this mechanism to deliver multiple invalidation requests instead of one within a network packet; Section 4 discusses the simulation methodology and the workloads we use; Section 5 presents experiment results; Section 6 describes related work and we conclude in section 7.

2 Distance Aware Protocol

Conventional cache coherence protocols of DSMs are not designed for a dedicated network so as to facilitate the flexibility. Protocols neither care about whether the network is of mesh topology or torus topology, nor require messages to be delivered in order [5]. More specifically, processors proceed without the knowledge of what their positions are in the system and what is the distance from one to another.

Although flexible, this methodology led to missing some optimization opportunities. Shared copies of a cache line must be invalidated before the data can be modified or evicted from L2 cache. With oblivious policy, the invalidation message for the further node, which takes more cycles to reach its destination, could be sent late resulting in increased overall latency. As we can also see from figure 1, it has little benefit of getting the acknowledgement early from node B, because the directory has to wait until the last acknowledgment arrives which is most probably from C – the furthest node. So sending message for C early may be a better solution.

The coherence protocol should be aware of the network topology and set the priority of dispatching invalidation messages based on the distances of sharers away from the directory. Each cycle, the distances of the sharers, which are left to be sent invalidation messages to, are calculated and then, the protocol sends an invalidation message to the furthest sharing node. As such, the total cost commonly will not exceed the delay of the invalidation-acknowledgment loop for the furthest node. This mechanism sends out long-delay messages first to hide the latency of short-range ones.

Take an XY routing mesh topology network for example, one intuitive way of defining the distance between two nodes i and j is by the Manhattan distance:

$$Distance_{i,j} = |x_i - x_j| + |y_i - y_j| \tag{1}$$

where x and y are the coordinates of a node. To reduce the calculation latency in hardware, we could approximate the distance by hops in one dimension, which almost achieves the same improvement in practice.

3 Multi-target Invalidations

Observations showed that the number of cores sharing a cache line is usually larger than one. It is especially the case for instruction cache lines which are shared by almost all processor cores while running parallel programs. When invalidating all the copies, coherence protocols conventionally send one invalidation message for each sharing core respectively. This could create message bursts into the network resulting in significant contention and negative impact on performance. This problem will be exacerbated in on-chip network environment where buffer resources are limited because of power and area budget.

Compacting multiple invalidation requests into one network packet could reduce invalidation messages in flight. The processor cores are divided into several groups. Messages destined for the cores within a group potentially share one routing path to some extent. As such, directories would send one invalidation message for each group within which multiple cores are targeted. Routers in the network deliver the multi-target invalidation to the specified group and dispatch the message to the targets one by one. In conjunction with the mechanism described in the above section, the multi-target invalidation for the furthest group should be sent first. This approach adds to each invalidation message a vector representing the targets in a group and the identification of that group. In routers, each buffer entry of the invalidation channel needs only to be augmented with a few extra bits, which is negligible.

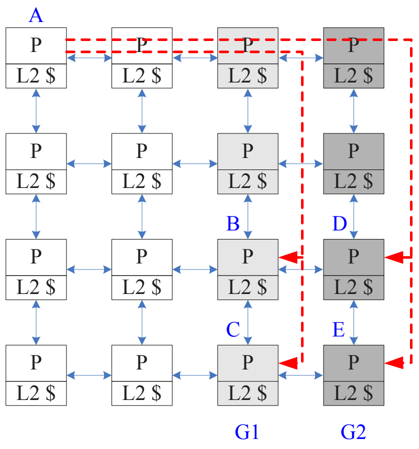


Fig. 3. Multi-target invalidations. Group *G1* consists of the light shaded nodes and group *G2* consists of the heavy shaded nodes.

Again, we demonstrate the mechanism with an XY routing mesh network. Figure 3 shows how multi-target invalidations are done. Cores in the same column form a group, because, with XY routing algorithm, messages for cores in the same column would first go through the same path along X dimension (e.g. messages for B and C from A share all the path from A to B). As illustrated, when the directory in core A needs to invalidate copies in the L1 caches of cores B, C, D and E, instead of sending out four invalidations respectively, it sends just two messages for the two groups. Each message has two targets, and the result is a 50% saving of the number of messages. After the message for A and B arrives at the router attached with core B, the router will find it has multiple targets and continue to send the message down to core C in parallel with invalidating the cached copy in the core B.

4 Simulator and Workloads

We use a cycle-accurate execution-driven simulator to evaluate our proposed mechanisms. The processor cores modeled in the simulator conform to the architecture of the Godson2 processor [7] which is a high performance microprocessor implementing MIPS ISA and featuring 4-issus, out-of-order execution and non-blocking caches, etc. We implement the directory-based write-invalidate cache coherence protocol and on-chip network in significant detail to make the simulator behave in strict accordance with the hardware implementation. This methodology provides accurate simulation results while at the cost of long simulation time. The simulator models a 16-core CMP with an on-chip network using mesh topology and XY routing algorithm. We employ an aggressive implementation of routers which take two cycles to forward a packet without contention. The contention within the on-chip network is also simulated. The wires are optimistically assumed to take just two cycles to deliver a packet from a router to the next. We believe that as the technology evolves, the speed of processor cores will go further beyond that of wires, and the advantages of our mechanisms will be more evident as wire delay increases. The detail architecture parameters are summarized in Table 1.

Table 1. System configurations

Parameters	Value
Number of cores	16
Processor	4-issue, out-of-order
Cache block size	32B
L1 I-cache	64KB, 4-way, 1-cycle latency
L1 D-cache	64KB, 4-way, 1-cycle latency
Shared L2 cache	8MB, 4-way, 4-cycle latency
DRAM latency	100 processor cycles latency
Network Topology	4*4 2-D mesh
Router	2 pipeline stages
Wire delay	2 processor cycles latency

To test our ideas, we employ a set of scientific applications consisting of seven programs from the SPLASH-2 benchmark suite. The programs are run to completion, but all experimental results reported in this paper are for the parallel phases of these applications. Table 2 presents the applications and the input data sets we use in the evaluation.

Table 2. Applications and input data sets

Applications	Problem sizes
FFT	256K complex data points
LU	512*512 matrix
Water nsquared (WATERNS)	512 molecules, 3 timesteps
Water spatial (WATERSP)	512 molecules, 3 timesteps
Cholesky	D750
LU noncontiguous (LUNC)	128*128 matrix
Ocean	130*130 array, 1e-7 error tolerance

When evaluating the multi-target invalidation mechanism, we divide the cores into four groups by columns. Four cores lying on the same column belong to one group. Each multi-target invalidation message needs a 4-bit vector to represent the four cores in a group and extra two bits to identify the group routed to.

5 Results

This section describes the simulation results of applying both the two mechanisms, compared with the baseline protocol.

Figure 4 depicts the distributions of the number of sharers when a block needs to be invalidated. The home node of a cache block is not counted, as invalidating the copy in the L1 cache of a block's home node does not incur a message into the network. As we can see, programs demonstrate various behaviors. Almost all the data in FFT and ocean has just one sharer except for the home node. We can predict that these two applications can gain little improvement as few messages can be saved from multi-target invalidations. Some applications (like LU, WATERSP and Cholesky) have modest number of sharers, but barely exceeds 5. The performance of these applications can be expected to boost. LUNC is different from others as most of its data is shared by multiple cores, however, the number of sharers also hardly surpasses 5.

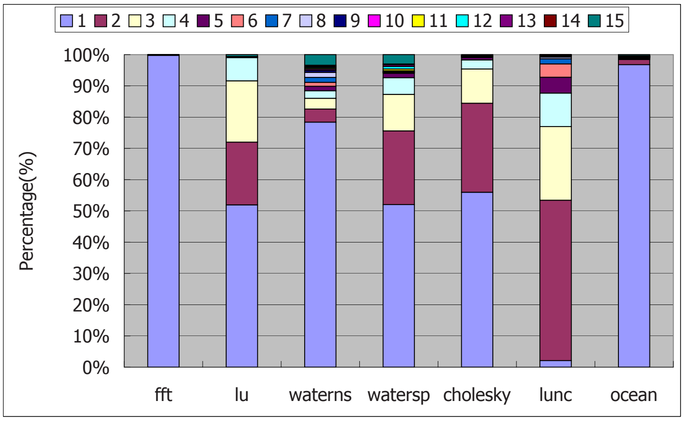


Fig. 4. Distributions of the number of sharers when a block needs to be invalidated

Reductions in cycles spent during invalidation process are shown in figure 5. The two mechanisms reduced the average invalidation traffic overhead by 5%. Among the applications, LU achieved the most improvement with nearly 8% decrease in latency. This can be inferred from its distribution of number of sharers. As we predicted, FFT and ocean barely have improvement. However, the traffic overhead within LUNC has not been reduced as we expected from the data in figure 4. We can explain the phenomena with figure 6. The mechanisms have limited impact on overall performance as the dominant factor is memory access latency.

Figure 6 presents distributions of the number of targets in each multi-target invalidation message. In some applications, data is shared by several cores, however, the sharers are scattered around other than kept together in groups. So

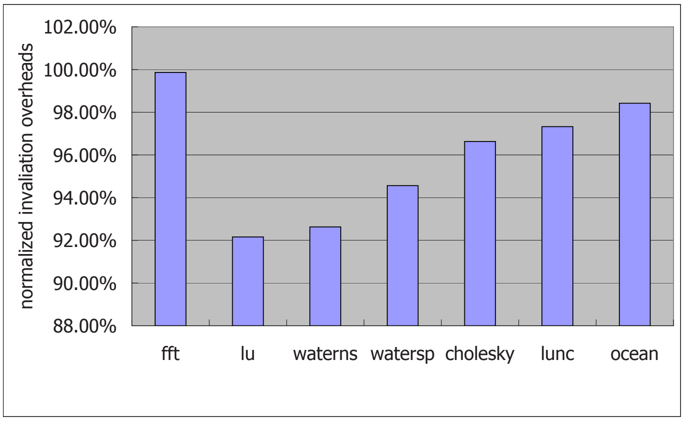


Fig. 5. Invalidation overhead scaled to the baseline protocol

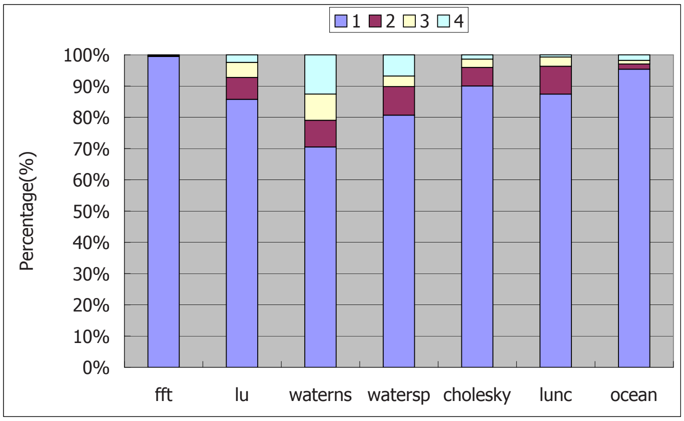


Fig. 6. Distributions of the number of targets in each multi-target invalidation message

each multi-target invalidation message is only responsible for few targets, decreasing its effect on reducing the overhead. This explains why LUNC receives modest improvement as most of its multi-target invalidations aim for just 1 or 2 sharers.

6 Related Work

A large body of literature focuses on optimization techniques for cache coherence protocols. [8,9] proposed adaptive coherence protocols for different data sharing patterns. [10,11] effectively eliminated the overhead of remote misses by making producers push data to consumers in advance, instead of fetching data when consumers' read misses actually happened. Lebeck and Wood [12] proposed

Dynamic Self-Invalidation (DSI) to automatically write back the writer's dirty copy of data at synchronization boundaries so as to save the coherence messages occurred when a sharer subsequently reads the same cache line. Lai et al. [13] extended DSI with last-touch predictor to invalidate the data more timely and avoid potential message bursts into the network.

On-chip networks also attract considerable attention. Researchers aim to reduce the transmission latency by shortening the depth of router pipeline stages and alleviate contention with adaptive routing algorithms [14,15].

All the researches mentioned above devote to their own field. Recently, however, we have noticed some work that demonstrated the benefits of coupling cache coherence protocols with underlying networks more tightly. Noel et al. [16] proposed embedding directories within each router node to satisfy requests with nearby data copies. Cheng et al. [17] leveraged wires of different power and latency properties to delivery different coherence protocol messages depending on their bandwidth-latency requirements.

7 Conclusions and Future Works

In this paper, we proposed two techniques to reduce invalidation traffic overhead in CMPs within which processor cores are connected by on-chip networks. We are motivated by one major difference between DSMs and CMPs: as far as flexibility is concerned, traditional cache coherence protocols of DSMs are not designed for a dedicated network missing some optimization opportunities; as for CMPs, parameters of the on-chip network are determined at design time, so we can jointly consider the both. Distance aware optimization chose to dispatch invalidations by the order of how far the sharers are away from the directory. Compared with oblivious policy, this mechanism guarantees long latency events to be processed first so as to lower overall overhead. Multi-target invalidations convey multiple invalidation messages for a group of cores within one network message. This approach can decrease invalidation traffic and alleviate message bursts into the network when crowd of cores share a cache line.

We conducted simulation on a 16-core CMP simulator using a subset of SPLASH-2 benchmark suite. The experimental results showed that the two mechanisms together reduced the average invalidation traffic overhead by 5%, up to 8%.

In the future, we will optimize the simulator to support more number of cores. As for now, the simulator models the coherence protocol in significant detail, so it takes prohibitively long time to simulate over 32 cores. We will refine the implementation in a more efficient and configurable way, and we believe that these two approaches will achieve more notable improvement in CMPs containing much more cores.

Acknowledgements. We would appreciate the anonymous reviewers for their valuable advice. Our work is supported by the National Natural Foundation of China for Distinguished Young Scholars under Grant No. 60325205, the National Natural Science Foundation of China under Grant No. 60673146 and No.

60603049, the National High Technology Development 863 Program of China under Grant No. 2006AA010201, the National Basic Research Program of China under Grant No. 2005CB321600 and the Beijing Natural Science Foundation under Grant No. 4072024.

References

1. Dally, W.J., Towles, B.: Route packets, not wires: on-chip inteconnection networks. In: DAC '01: Proceedings of the 38th conference on Design automation, New York, NY, USA, pp. 684–689. ACM Press, New York (2001)
2. Ho, R., Mai, K.W., Horowitz, M.A.: The future of wires. Proceedings of the IEEE 89(4), 490–504 (2001)
3. Zhang, M., Asanovic, K.: Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In: ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture, Washington, DC, USA, pp. 336–345. IEEE Computer Society, Los Alamitos (2005)
4. Held, J., Bautista, J., Koehl, S.: From a Few Cores to Many: A Tera-scale Computing Research Overview. Technical report, intel (2006)
5. Laudon, J., Lenoski, D.: The sgi origin: a ccnuma highly scalable server. In: ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture, pp. 241–251. ACM Press, New York, NY, USA (1997)
6. Dally, W.J., Towles, B.: Principles and Practices of Interconnection Networks. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2003)
7. Hu, W., Zhang, F., Li, Z.: Microarchitecture of the Godson-2 Processor. Journal of Computer Science and Technology 20(2), 243–249 (2005)
8. Cox, A.L., Fowler, R.J.: Adaptive cache coherency for detecting migratory shared data. In: ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture, New York, NY, USA, pp. 98–108. ACM Press, New York (1993)
9. Kaxiras, S., Goodman, J.R.: Improving CC-NUMA Performance Using Instruction-Based Prediction. In: Proceedings of the Fifth IEEE Symposium on High-Performance Computer Architecture, pp.161–170 (1999)
10. Abdel-Shafi, H., Hall, J., Adve, S.V., Adve, V.S.: An evaluation of fine-grain producer-initiated communication in cache-coherent multiprocessors. In: Third International Symposium on High-Performance Computer Architecture, 1997, pp. 204–215 (1997)
11. Koufaty, D.A., Chen, X., Poulsen, D.K., Torrellas, J.: Data forwarding in scalable shared-memory multiprocessors. In: ICS '95: Proceedings of the 9th international conference on Supercomputing, pp. 255–264. ACM Press, New York, NY, USA (1995)
12. Lebeck, A.R., Wood, D.A.: Dynamic self-invalidation: reducing coherence overhead in shared-memory multiprocessors. In: ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture, pp. 48–59. ACM Press, New York, NY, USA (1995)
13. Lai, A.-C., Falsafi, B.: Selective, accurate, and timely self-invalidation using last-touch prediction. In: ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture, pp. 139–148. ACM Press, New York, NY, USA (2000)

14. Mullins, R., West, A., Moore, S.: Low-latency virtual-channel routers for on-chip networks. In: ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture, Washington, DC, USA, p.188. IEEE Computer Society, 188 (2004)
15. Kim, J., Park, D., Theocharides, T., Vijaykrishnan, N., Das, C.R.: A low latency router supporting adaptivity for on-chip interconnects. In: DAC '05: Proceedings of the 42nd annual conference on Design automation, pp. 559–564. ACM Press, New York, NY, USA (2005)
16. Eisley, N., Peh, L.S., Shang, L.: In-network cache coherence. In: MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, Washington, DC, USA, pp. 321–332. (2006)
17. Cheng, L., Muralimanohar, N., Ramani, K., Balasubramonian, R., Carter, J.B.: Interconnect-aware coherence protocols for chip multiprocessors. In: ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture. Washington, DC, USA, pp. 339–351. IEEE Computer Society (2006)

Generalized Wormhole Switching: A New Fault-Tolerant Mathematical Model for Adaptively Wormhole-Routed Interconnect Networks

F. Safaei[1,3], A. Khonsari[1,2], M. Fathy[3], N. Talebanfard[4], and M. Ould-Khaoua[5,6]

[1] IPM School of Computer Science, Tehran, Iran
[2] Dept. of ECE, Univ. of Tehran, Tehran, Iran
[3] Dept. of Computer Eng., Iran Univ. of Science and Technology, Tehran, Iran
[4] Faculty of Mathematical Sciences, Shahid Beheshti Univ., Tehran, Iran
[5] Dept. of Electrical and Computer Eng., Sultan Qaboos Univ., Al-Khodh, Oman
[6] Dept. of Computing Science, Univ. of Glasgow, UK
{safaei,ak}@ipm.ir, {f_safaei,mahfathy}@iust.ac.ir, mohamed@dcs.gla.ac.uk

Abstract. In this paper, we introduce a new variant of WS which makes any adaptive routing algorithm augmented with virtual channels tolerate multiple fault regions. More specifically, we propose a mathematical model for this switching mechanism using Markov chain to calculate the probability of header message blocking and to capture the traffic rates on channels in the presence of faulty components. Simulation results based on the network topology confirm the validity of the analytical approximation and demonstrate the localizer efficiency.

1 Introduction

The switching method determines the way messages visit intermediate routers and one of the most important parameters to improve the network performance. The Wormhole Switching (WS) is a popular technique used to provide inter-processor communication for contemporary communication systems. It divides a message into a few *flits* and spreads them over several nodes. As the *header flit* containing routing and control information advances along a specific route; the sequential flits follow it in a pipelined fashion [1]. Wormhole messages, therefore, realize very good performance, but are prone to deadlock in the presence of faults. Whenever the header flit cannot find a route to the destination because of busy/faulty links/nodes, the remaining flits stop advancing and have to wait until the channel is empty or repaired. Otherwise, the message has to be removed. This behavior of WS can lead to situations where the routing header can become blocked, no longer make progress, and hence cause the network to become deadlocked.

In this correspondence, we propose the use of a new variant of WS flow control mechanism for fully adaptive routing in pipelined networks. The paper contributes this flow control mechanism at the switching layer. Routing protocols can be designed such that in the vicinity of faulty components, messages use the proposed style where backtracking can be allowed to avoid failures and deadlock configurations. The

© Springer-Verlag Berlin Heidelberg 2007

proposed switching mechanism does not require any complex scheme to provide fault-tolerance capability, and does not require additional virtual channels for routing messages around the faulty components. The hardware amount for the proposed method is almost equal to that for the WS, making the implementation very feasible. Moreover, we develop a mathematical model to investigate the efficiency of the proposed mechanism coupled with virtual channels and fully adaptive routing in the presence of faulty components. The model makes latency predictions that are in good agreement with those obtained from simulation experiments. Recently, a crude simulation experiment [2] based on WS has been proposed to capture the effects of Duato's Protocol [1] in the presence of random failed links. The proposed model, however, is different in three aspects from that reported in [2]. First, it uses theoretical results of probabilistic analyzes and queuing theory to present a new analytical model for computing the message latency of a fault-tolerant switching based on the WS. Second, it can be used both for Duato's methodology and fully adaptive routings. Third, it can capture the characteristics of the most common fault patterns and calculate the probability of a message confronting such patterns.

This article includes five main sections. Section 2 introduces a few definitions, and the node structure. The proposed flow control mechanism is also introduced in this section. Section 3 presents the derivation of the analytical model. Section 4 validates the mathematical model through simulation experiments. Finally, Section 5 concludes the paper with future research directions.

2 Preliminaries

2.1 Node Configuration

Although the Generalized WS can be used in any topology, the present mathematical model is discussed in the context of the torus due to the popularity of this topology in current interconnect networks. The class of networks considered in this paper is the torus connected, bidirectional, 2 dimensional (2-D). A $k \times k$ torus is a direct network with $N = k^2$ nodes; k is called the radix. Each node can be identified by a 2-digit radix k address (a_1, a_2). Nodes, with address (a_1, a_2) and (b_1, b_2) are connected if and only if $a_1 = (a_2 + 1) \bmod k$ or $b_1 = (b_2 + 1) \bmod k$. Each node consists of a Processing Element (PE) and router. A node is connected to its neighboring nodes via the input and output channels. The *injection*/*ejection* channel is used by the processor to inject/eject messages to/from the network.

2.2 The Proposed Switching Mechanism

In this section, we consider a new variant of WS by improving it to have backtracking capability. From low to moderate number of faults, the proposed switching mechanism has the advantage of considering faulty nodes/links over WS, which can lead to deadlock-free fault-tolerant routing protocol whose performance is superior to WS with comparable reliability. Since the header message is followed immediately by data flits in WS, the header cannot backtrack to the preceding node. Thus, if the header cannot progress due to a faulty component, the message is blocked in place indefinitely, holding buffer resources and blocking other messages. This situation can eventually result in a deadlocked configuration of messages. While techniques such as adaptive routing can alleviate the problem, it cannot by itself solve the problem. In the proposed approach, on the other hand, when the header encounters a faulty node or experiences blocking because all the required virtual channels are busy, it releases the last reserved virtual channel and backtracks to the preceding node by copying the routing information into one fictive flit followed immediately by it. By storing the routing information of the header into a fictive flit, the fictive flit taking the role of the header, goes back to the preceding node, and searches for an alternative path to progress towards its destination.

We will now outline the modifications that have to be made in order to enable backtracking of WS. We assume that the header is in fact a part of the message body, namely we insert K $(K \geq 0)$ fictive flits, which include no information, between the header and the first real data flit. Thus, the distance between the header and the first data flit is always at most K hops. This means that the number of new flits including the header itself is at most K. When $K = 0$, the flow control mechanism is equivalent to the WS mechanism without backtracking capability, while large values of K can ensure path set-up prior to data transmission if a path exists. Intermediate values of K can permit the data flits to follow the header at distance, allowing the header or a fictive flit to backtrack. Therefore, when the header (fictive flit) reaches the destination, the first data flit arrives shortly thereafter rather than immediately as in WS. It is apparent that the number of fictive flits indicates the maximum number of backtrackings that a message can perform to avoid the faulty components. The less number of fictive flits in a message results in the less backtracking capability. However, with the large values of K, substantial traffic can be introduced into the network, dominating the effect of an increased overhead associated with the path set-up time. Hence, the selection of an optimal value of K is dependent upon the network traffic and the fault patterns, and is a trade-off between the traffic, and the increased backtracking that occurs during the path construction in network.

3 Mathematical Model

In this section, we describe an analytical model for assessing the performance of WS with fictive flits in an adaptively-routed torus. The most important performance metric in our model is the mean message latency.

3.1 Assumptions

The model is based on the following assumptions, which are accepted in the literature [3-10], and are listed below.

- Nodes generate traffic independently of each other, following a Poisson process with an average rate of λ_{node} messages/ node/cycle.
- Message destination nodes are uniformly distributed across the network.
- The message length is fixed at M flits, each of which requires one cycle to cross from one router to the next.

- The local queue at the injection channel in the source node has infinite capacity. Messages at the destination node are transferred to the local PE one at a time through the ejection channel.
- $V\ (\geq 1)$ virtual channels per physical channel are used. When there are more than one virtual channels available that bring a message closer to its destination, one is chosen at random.
- Nodes (processors) are more complex than links and thus have higher failure rates [1-4]. So, we assume only node failures.
- Fault patterns are static [1, 2, 4]; distributed uniformly through the network, and do not disconnect the network.
- Each node failure occurs randomly and independently to the other ones with probability θ .
- Messages are assumed to always follow the shortest paths in the absence of faults.

In the sequel, we will derive a mathematical model that approximates the behavior of 2-D torus communication system using the proposed switching mechanism augmented with virtual channels and fully adaptive routing around fault regions.

3.2 Communication Analysis

The mean message latency is composed of the mean network latency, $\bar{T}$, which is the time to cross the network and the mean waiting time seen by the message in the source node, $\bar{W}_s$, before entering the network. However, to capture the effects of virtual channels multiplexing, the mean message latency has to be scaled by a factor, say $\bar{V}$, representing the average degree of virtual channels multiplexing, that takes place at a given physical channel. Therefore, the mean message latency can be approximated as [6]

$$Mean\ Message\ Latency = (\bar{T} + \bar{W}_s)\bar{V} \tag{1}$$

Under the uniform traffic pattern, the average number of channels that a message visits along a given dimension and across the network, $\bar{k}$, $\bar{D}$ respectively, are given by Agarwal [7]

$$\bar{k} = \begin{cases} \dfrac{k}{4} & k \equiv 0 \pmod 2 \\ \dfrac{1}{4}(k - \dfrac{1}{k}) & k \equiv 1 \pmod 2 \end{cases}, \quad \bar{D} = 2\bar{k} \tag{2}$$

where $\equiv$ signifies the congruence relation.

Calculation of the Traffic Rate on a Network Channel

Fully adaptive routing allows a message to use any available channel that brings it closer to its destination resulting in an evenly distributed traffic rate on all network channels. Calculation of the traffic rate of messages received by each channel, $\lambda_{channel}$, can be approximated as follows. The header and the fictive flits may have to make more than one attempts before the path is successfully constructed. On average, $\overline{C}$, channels (is determined below) are visited to establish a path. Due to the uniformity of traffic inside the network, the message arrival process exhibits similar statistical behavior across all network channels. Thus, the channel arrival rate can be found by dividing the total channel arrival rates over the number of non-faulty channels in the network. In a 2-D torus, each node has $4(1-\theta)^2$ non-faulty output network channels. Thus, the traffic rate arriving at a channel can be written as

$$\lambda_{channel} = \lambda_{node}\overline{C} / \left(8(1-\theta)^2\right) \tag{3}$$

Calculating the Mean Time to Set-Up a Path

When the header finds all potential virtual channels at a given intermediate node busy, or it can not find a network path because of faulty components, the header is forced to backtrack to the preceding node by copying the routing information into the fictive flit followed by it. The fictive flit then takes the role of the header message and can resume searching for an alternative path. In order to compute the mean time, $\overline{C}_r$, to establish a path for an r -hop message (i.e., a message that needs to make r hops to cross from source to destination), the header and actions (e.g., advancing and backtracking) are modelled as a Random Walk problem [10] where the associated Markov chain is illustrated in Fig. 1. We wish to propose a mathematical model to evaluate the performance behaviour of the proposed switching mechanism. To do so, we will consider a somewhat unusual labelling of the nodes. A state in the Markov chain represents the current location (i.e., node) of the header message along its network path. State $\omega_{-(M+K),0}$ denotes that the header is at the source node. We will assume that the last data flit is at node with label $-(M+K)$ and the destination would be at the same node with label r , meaning that the destination is r hops away from the header. The message needs to cross $r+K-1$ intermediate nodes to reach its destination. We let $\omega_{i,j}$ be the state denoting that the last data flit is $M+K+i$ hops away from the source, and the header message is $M+K+j$ hops far from the source node. A transition out from state $\omega_{i,j}$ to $\omega_{i+1,j+1}$ implies that the header has succeeded in reserving the next required virtual channel that brings it one hop closer to its destination, and this occurs with the probability $1-\left(Pb_j+P_{hit}\right)$; where Pb_j and P_{hit} denote the probability of header blocking and the probability of facing fault patterns, respectively. When the faults are randomly distributed through the network, the parameter P_{hit} is equal to an independent node failure probability (i.e., θ). However, in the vicinity of failures, P_{hit} must be calculated in a different manner. The details of calculation of P_{hit} using adaptive routing scheme have been reported in [4]. Similarly, a transition out from state $\omega_{i,j}$ to $\omega_{i,j-1}$ corresponds to the case where the header has encountered blocking/faulty situation and one of the fictive flits has been used. The transition rate is the probability, Pb_j+P_{hit} . Further, a transition out from state $\omega_{i,j}$ to $\omega_{i,j}$ happens when all the fictive flits have been consumed and the header message has stopped at the node corresponding to state $\omega_{i,j}$.

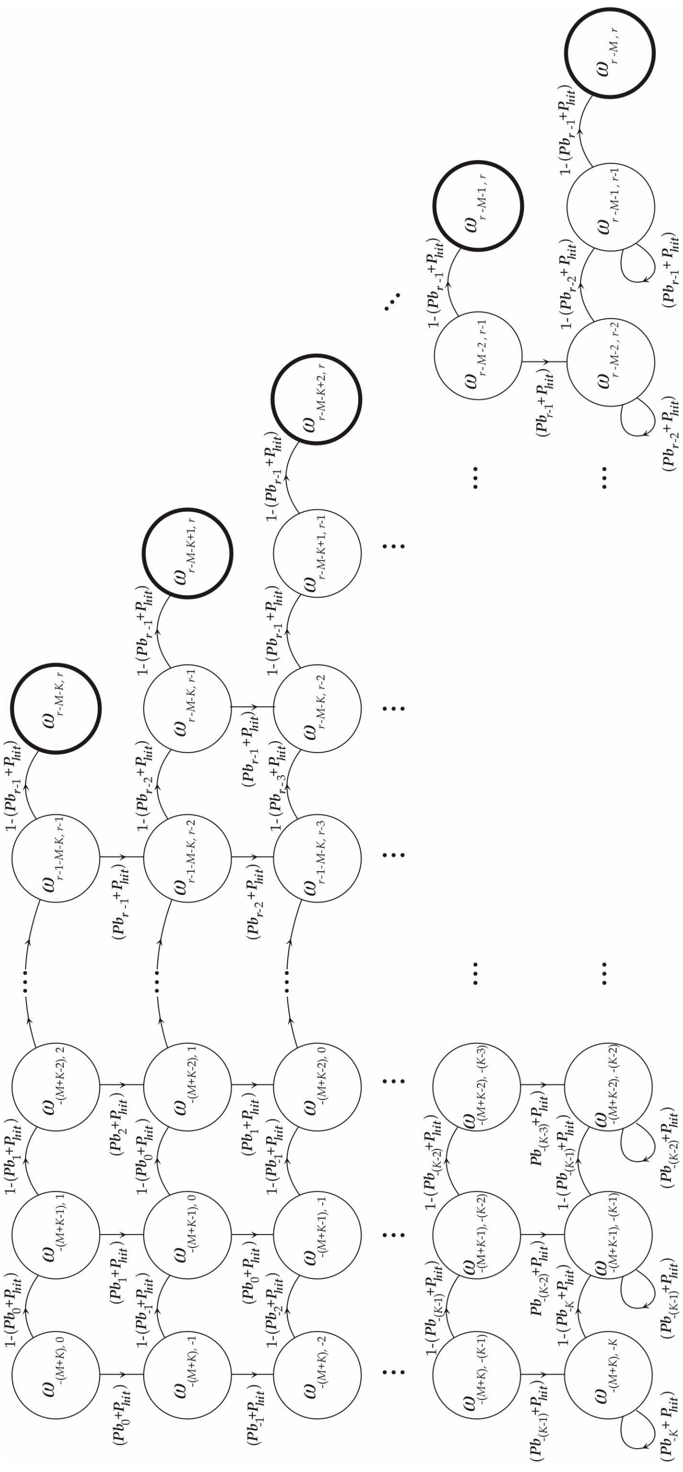


Fig. 1. The Markov chain for calculating the mean path set-up time; the final states are shown as solid circles

Each row $g\ (0 \le g \le K)$ in the Markov chain corresponds to the number of fictive flits that are already used. It is evident from the figure that any state $\omega_{i,r}$ is a final state. Let $\overline{C}_{i,j}$ denote the expected time interval that the system has reached at state $\omega_{x,r}$, $r-M-K \le x \le r-M$, starting form state $\omega_{i,j}$. Given that the header message requires one cycle to move from one node to the next, one can easily check that the following difference equations hold for $\overline{C}_{i,j}$.

$$\overline{C}_{i,j} = \begin{cases} 0 & j = r \\ \left(1-\left(Pb_j + P_{hit}\right)\right)\overline{C}_{i+1,j+1} + \left(Pb_j + P_{hit}\right)\overline{C}_{i,j-1} + 1 & j \ge M+i+1 \\ \left(1-\left(Pb_j + P_{hit}\right)\right)\left(\overline{C}_{i+1,j+1} + 1\right) + \left(Pb_j + P_{hit}\right)\overline{C}_{i,j} & j = M+i \end{cases} \tag{4}$$

Let the initial state be $\omega_{-(M+K),0}$, the mean path set-up time can be calculated as

$$\overline{C}_r = \overline{C}_{-(M+K),0} \tag{5}$$

We now give some insight in order to illustrate the problem of calculation of $\overline{C}_{-(M+K),0}$. We present the following algorithm in details and analyze its complexity.

```
00 Algorithm I
01 Input: An array $\overline{C}_{i,j}$
02 Output: A value indicating the mean path set-up time
03   for $i \leftarrow -(M+K)$ to $r-1-M$ do
04     $\overline{C}_{i,i+M} \leftarrow r-i-M$
05   for $i \leftarrow r-M-K$ to $r-M$ do
06     $\overline{C}_{i,r} \leftarrow 0$
07   for $i \leftarrow 1$ to $K$ do
08     for $j \leftarrow r-1$ downto $i-K$ do
09       $\overline{C}_{j-M-i,j} \leftarrow \left(1-\left(Pb_j + P_{hit}\right)\right)\overline{C}_{j-M-i+1,j+1} + \left(Pb_j + P_{hit}\right)\overline{C}_{j-M-i,j-1} + 1$
```

Lemma 1: *Algorithm I correctly computes the quantity of* $\overline{C}_{i,j}$ *for all* i,j *with proviso* $j \le i+M+K$ *, in time* $\mathrm{O}(K(r+K))$.

Proof: $\overline{C}_{i,i+M} = r-i-M$ is followed from the fact that the mean time to set-up a path in the traditional WS is the same as the distance between the destination and the header message. Now, we prove that $\overline{C}_{j-M-i,j}$ is obtainable, when $1 \le i \le K$ and $1 \le j \le r-1$. According to Eq. (4), we get

$$\overline{C}_{j-M-i,j} \leftarrow \left(1-\left(Pb_j + P_{hit}\right)\right)\overline{C}_{j-M-i+1,j+1} + \left(Pb_j + P_{hit}\right)\overline{C}_{j-M-i,j-1} + 1$$

It is clear that, $\overline{C}_{j-M-i+1,j+1}$ and $\overline{C}_{j-M-i,j-1}$ are already computed in the previous steps and therefore $\overline{C}_{j-M-i,j}$ could be computed. To calculate the time complexity of the Algorithm I, we note that the number of times in which the **for** loop in line 8 of algorithm will be executed is at most $r + K - 1$. Thus, using the formula above, we obtain a bound of $O(K(r + K))$ on the time complexity. ♦

Averaging over the $N(1-\theta)-1$ possible healthy destination nodes in the network gives the mean time to set-up a path, $\bar{C}$, from source node $\mathcal{S}$ to destination node $\mathcal{D}$ as

$$\bar{C} = \frac{1}{N(1-\theta)-1} \sum_{\mathcal{D} \in \mathcal{G} \backslash \{\mathcal{S}\}} \bar{C}_r \tag{6}$$

where $\mathcal{G}$ is the set of all surviving nodes in the network and the operator "\" denotes the set difference between $\mathcal{G}$ and $\mathcal{D}$. In what follows, we will describe the calculation of the following quantities: $\bar{T}$, Pb_j, $\bar{W}_s$, and $\bar{V}$.

3.2.1 Calculating the Mean Network Latency

Since, the torus topology is symmetric and message destinations are uniformly distributed across the network the arrival patterns of messages (and the service times seen by messages) at the network channels exhibit similar statistical behaviour. Let $\bar{T}_r$ denote the mean network latency seen by an r-hop message. In the proposed switching method, the mean network latency consists of two parts: one the time to setup a path ($\bar{C}_r$), and the other, the delay due to the actual message transmission time ($\bar{T}_r$). Hence, the network latency of an r-hop message can be written as

$$\bar{T}_r = M + (K-1) + \bar{C}_r \tag{7}$$

where $\bar{C}_r$ denotes the mean time to set-up a path for an r-hop message and M is the message length. Note that in Eq. (7) the term $(K-1)$ accounts for $(K-1)$ cycles that are required to backtrack in the direction leading to the real data flits. For a given non-faulty node in the network, the mean latency seen by a message originated at that node to enter the network, $\bar{T}$, is equal to the average of all $\bar{T}_r$ resulting in

$$\bar{T} = \frac{1}{N(1-\theta)-1} \sum_{\mathcal{D} \in \mathcal{G} \backslash \{\mathcal{S}\}} \bar{T}_r \tag{8}$$

Calculating the Probability of Header Message Blocking

Examining the Eqs. (4) and (6) reveals that the probability of blocking, Pb_j, is required to calculate $\bar{C}$. In this section, we compute the probability of header message blocking. The probability that the header message is blocked at a given channel depends on its current network position. This is because the number of alternative paths that the header can take to progress is determined by the number of hops made by the header, and the way that these hops are distributed among the dimensions [6]. Let Pb'_j denote the probability that the header of an $(M+K+r)$-hop message is blocked after making j hops. It is clear that $Pb_j = Pb'_{j+M+K}$. Therefore, in order to compute Pb_j, we must only evaluate Pb'_j for each j. To this end, let Θ_j^t be the probability that the header message has entirely crossed t $(0 \le t \le 1)$ dimensions after making j hops. The details of calculation of Θ_j^t have been developed elsewhere [6]. We recollect briefly here the main equations for the calculation of Θ_j^t. The number of channels, and thus the number of virtual channels, that the header can select at a given hop depends on the number of dimensions still to be visited. When the header has made $j\,(0 \le j \le M+K+r-1)$ hops, these hops can be a combination of (x,y) hops, with x and y being the number of hops achieved in the first and second dimensions respectively, where $(x+y=j), \left(0 \le x,y \le \bar{k}\right)$. To determine the probability that the header message has crossed all the channels of one dimension, two cases need to be considered:

(1) When $(0 \le j < \bar{k})$, the header has not yet crossed any dimension since it has to make $\bar{k}$ hops along each dimension. Therefore, the header can choose among virtual channels of both dimensions.

(2) When $(\bar{k} \le j < M+K+r-1)$, the number of ways to distribute these hops along the two dimensions is $(M+K+r-j+1)$. In only two cases, $(x=\bar{k}, y=j-\bar{k})$ and $(x=j-\bar{k}, y=\bar{k})$, the header has crossed all channels of one dimension and thus, all the remaining hops have to be made in other dimension.

So, when the header has made j hops, the probability that there remains only one dimension to be crossed, P_{φ_j}, can be written as

$$P_{\varphi_j} = \begin{cases} 0 & 0 \le j < \bar{k} \\ \dfrac{2}{M+K+r-j+1} & \bar{k} \le j < M+K+r-1 \end{cases} \tag{9}$$

When the header arrives at the j-th hop channel, it has already made $(j-1)$ hops and has entirely crossed, say, t $(0 \le t \le 1)$ dimensions. At its next hop, the header message can select any available $(2-t)V$ virtual channels from the remaining $(2-t)$ dimensions. Blocking occurs when all possible virtual channels at the remaining dimensions to be visited are occupied. If P_V (given by Eq. (14)) denotes the probability that V virtual channels at a given physical channel are busy, the probability Pb'_j, that the header is blocked is given by

$$Pb'_j = \sum\nolimits_{t=0}^{1} \Theta_j^t (P_V)^{2-t} \qquad 0 \le j \le M+K+r-1 \tag{10}$$

where Θ_j^t is the probability that the header has entirely crossed t dimensions along on its j-hop path and is given by

$$\Theta_j^t = \begin{cases} 1 - P_{\varphi_j} & t = 0 \\ P_{\varphi_j} & t = 1 \end{cases} \tag{11}$$

3.2.2 Calculating the Mean Waiting Time at the Source Node

In this section, we compute the average waiting, $\bar{W}_s$, that a message experiences at the source node before entering the network. To this end, the injection virtual channel in the source node is treated as an M/G/1 queuing system [9]. Since a source node generates messages with a mean rate, λ_{node}, and a message can enter the network through any of the V virtual channels, the mean arrival rate at each injection virtual channel is λ_{node} / V. Under the uniform traffic pattern using adaptive routing results in the mean service time seen by messages at all source nodes being identical and equal to the mean network latency, i.e., $\bar{T}$ [6]. To simplify the development of our model while maintaining a good degree of accuracy in predicting the message latency we follow a suggestion by Draper and Ghosh [8] for computing the variance of the service time distribution. Since a message makes, on average, $\bar{D}$ hops to reach its destination, the minimum path set-up time is $\bar{D} + K - 1$, where K indicates the number of backtrackings permitted to one message transmission. So, the minimum network latency seen by the message is $M + K + \bar{D} - 1$. Applying the Pollaczek-Khinchine (P-K) mean value formula [9] with an approximated variance $(\bar{T} - M - K - \bar{D} + 1)^2$ [8] yields the mean waiting time seen by a message at the source node as

$$\bar{W}_s = \lambda_{node}\left(\bar{T}^2 + (\bar{T} - M - K - \bar{D} + 1)^2\right) / \left(2\left(V - \lambda_{node}\bar{T}\right)\right) \tag{12}$$

3.2.3 Calculating the Mean Degree of Virtual Channels Multiplexing

The probability, P_v ($0 \le v \le V$), that v virtual channels at a given physical channel are busy can be determined using a Markovian model (details of the model can be found in [4-6]). In the steady state, the model yields the following probabilities [5].

$$Q_v = \begin{cases} \left(\lambda_{channel}\bar{T}\right)^v & 0 \le v \le V - 1 \\ \left(\lambda_{channel}\bar{T}\right)^V \big/ (1 - \lambda_{channel}\bar{T}) & v = V \end{cases} \tag{13}$$

$$P_v = \begin{cases} \left(\sum_{v=0}^{V} Q_v\right)^{-1} & v = 0 \\ P_0 Q_v & 1 \le v \le V \end{cases} \tag{14}$$

In virtual channel flow control, multiple virtual channels share the bandwidth of a physical channel in a time-multiplexed manner. The average degree of virtual channel multiplexing, that takes place at a given physical channel, can be found to be [5]

$$\overline{V} = \sum_{v=1}^{V} v^2 P_v \,/ \sum_{v=1}^{V} v\, P_v \tag{15}$$

4 Model Validation

To further understand and evaluate the performance issues of the proposed switching mechanism, we have used a discrete-event simulator that operates at the flit level. For each simulation experiment, statistics were gathered for a total number of 100 000 messages. Statistic gathering was inhibited for the first 10 000 messages to avoid distortions due to the start-up transient. The results of simulation and analysis for the 8×8 (N = 64) and 20×20 (N = 400) networks with message length M = 32 and 64 flits, number of fictive flits K= 3 and 6, probability of facing fault pattern $P_{hit} = 0.1, 0.25$ and V =1, 6 virtual channels per physical channel are depicted in Fig. 2.

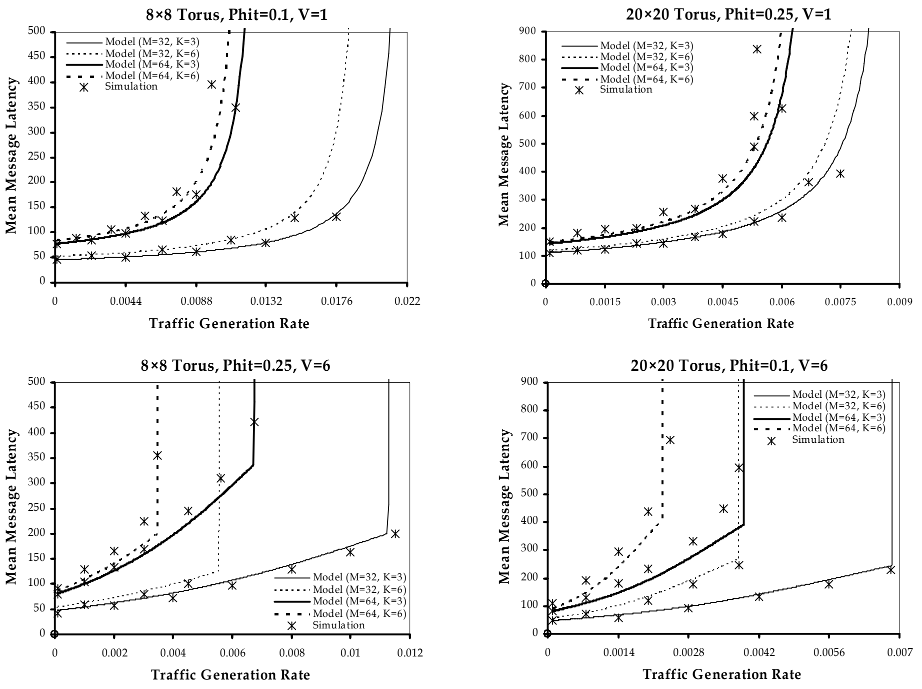


Fig. 2. Comparing the analytical model and flit-level simulation experiments in the 8×8 and 20×20 torus networks using Generalized Wormhole Switching with message length $M = 32, 64$ flits, $V = 1, 6$ virtual channels per physical channel, number of fictive flits $K = 3, 6$, and probability of facing fault patterns $P_{hit} = 0.1, 0.25$

In all graphs, the horizontal axis represents the traffic generation rate while the vertical axis shows the mean message latency in crossing from source to destination. The figure indicates that the analytical model predicts the mean message latency with a good degree of accuracy in all regions. However, some discrepancies around the saturation point are apparent. This is a result of the approximations made when constructing the analytical model, e.g. the approximation used to estimate the variance of the service time distribution at a channel. This approximation greatly simplifies the model by avoiding the computation of the exact distribution of the message service time at a given channel.

5 Conclusion

In this paper, we have proposed a new variant of Wormhole Switching (WS) for supporting inter-processor communications in interconnect networks due to its ability to preserve both communication performance and fault-tolerant demands in such systems. More specifically, we have proposed a mathematical model to evaluate the relative performance merits of this switching in tori when coupled with virtual channels and fully adaptive routing around fault regions. The proposed switching mechanism does not require any complex scheme to provide fault-tolerance capability, and does not require additional virtual channels for routing messages around the faulty components. The router designed to support the proposed mechanism requiring the same amount of hardware as a router supporting WS, makes the implementation very feasible. Future efforts are redesigning the router for supporting the dynamic fault-tolerance version of the proposed switching mechanism, and determining the optimal number of the fictive flits by means of the mathematical expressions.

References

1. Dao, B.V., Duato, J., Yalamanchili, S.: Dynamically configurable message flow control for fault-tolerant routing. IEEE Transactions on Parallel and Distributed Systems 10(1), 7–22 (1999)
2. Sueishi, M., Kitakami, M., Ito, H.: Fault-Tolerant Message Switching Based on Wormhole Switching and Backtracking. In: Proceedings of the 10th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'04), pp. 183–190 (2004)
3. Xu, J.: Topological structure and analysis of interconnection networks. Kluwer Academic Publishers, Dordrecht (2001)
4. Safaei, F., et al.: Performance Analysis of Fault-Tolerant Routing Algorithm in Wormhole-Switched Interconnections. Journal of Supercomputing (2007)
5. Dally, W.J.: Virtual channel flow control. IEEE Transactions on Parallel and Distributed Systems 3(2), 194–205 (1992)
6. Ould-Khaoua, M.: A performance model of Duato's adaptive routing algorithm in k-ary n-cubes. IEEE Transactions on Computers 48(12), 1–8 (1999)
7. Agarwal, A.: Limits on interconnection network performance. IEEE Transactions on Parallel and Distributed Systems 2(4), 398–412 (1991)
8. Draper, J.T., Ghosh, J.: A comprehensive analytical model for wormhole routing in multicomputer systems. Journal of Parallel and Distributed Computing 32(2), 202–214 (1994)
9. Kleinrock, L.: Queuing Systems, vol. 1. John Wiley, New York (1975)
10. Feller, W.: An introduction to probability theory and its applications. John Wiley, New York (1967)

Open Issues in MPI Implementation

Rajeev Thakur and William Gropp

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439, USA
{thakur,gropp}@mcs.anl.gov

Abstract. MPI (the Message Passing Interface) continues to be the dominant programming model for parallel machines of all sizes, from small Linux clusters to the largest parallel supercomputers such as IBM Blue Gene/L and Cray XT3. Although the MPI standard was released more than 10 years ago and a number of implementations of MPI are available from both vendors and research groups, MPI implementations still need improvement in many areas. In this paper, we discuss several such areas, including performance, scalability, fault tolerance, support for debugging and verification, topology awareness, collective communication, derived datatypes, and parallel I/O. We also present results from experiments with several MPI implementations (MPICH2, Open MPI, Sun, IBM) on a number of platforms (Linux clusters, Sun and IBM SMPs) that demonstrate the need for performance improvement in one-sided communication and support for multithreaded programs.

1 Introduction

MPI (the Message Passing Interface) is a widely used paradigm for parallel programming. It is used across the entire spectrum of parallel machines—from small Linux clusters to the largest parallel machines in the world such as IBM Blue Gene/L and Cray XT3. The MPI standard has existed for a long time—MPI-1 was released in 1994 and MPI-2 in 1997—and a number of MPI implementations are available. Free, portable implementations include MPICH, MPICH2, MVAPICH, MVAPICH2, LAM, and Open MPI. In addition, all computer-system and network-hardware vendors (such as IBM, Cray, Sun, HP, SGI, Intel, Microsoft, NEC, Hitachi, Fujitsu, Myricom, Quadrics, Mellanox, and QLogic) provide implementations of MPI. (Many of the vendor implementations are derived from the public-domain implementations.) Although MPI implementations have matured over the years, improvements are still needed in a number of areas. It is not sufficient just to provide the lowest possible ping-pong latency and highest possible large-message bandwidth between two processes. Users expect good performance across all aspects of the MPI standard.

In this paper, we discuss several areas in which MPI implementations still need improvement. These include performance, scalability, fault tolerance, support for debugging and verification, topology awareness, collective communication, derived datatypes, parallel I/O, one-sided communication, and support for

© Springer-Verlag Berlin Heidelberg 2007

multithreaded programs. For the last two areas, we also present results from experiments with several MPI implementations (MPICH2, Open MPI, Sun, IBM) on a number of platforms (Linux clusters, Sun and IBM SMPs) that demonstrate the need for performance improvements.

2 Areas Needing Improvement in MPI Implementations

Below we discuss in broad terms several areas in which better support is needed from MPI implementations. It is not a comprehensive list, but it covers most of the important topics.

2.1 Basic Performance

The holy grail of message-passing performance is to achieve sub-microsecond latency for short messages. That goal has already been achieved on shared-memory machines [4] but not yet on distributed-memory systems. In addition to achieving low latency and high bandwidth on ping-pong benchmarks, it is essential to deliver good performance across the entire range of message sizes, avoiding sharp jumps in between. However, this is not the case in many MPI implementations that use a different protocol for short and long messages (eager versus rendezvous delivery) to minimize the need for internal buffering. An example in shown in Figure 1: On the IBM Blue Gene/L, a large jump occurs around 1024 bytes because of the transition from eager to rendezvous protocol. Smoothing out such performance jumps is a difficult challenge because of the tradeoffs between performance and resource consumption.

Another basic performance requirement is that a user should be able to achieve better (or equal) performance by using a single MPI function than by using a combination of other MPI functions that can implement the same functionality [29]. This requirement is not met in some cases. For example, in Figure 1, a user with a 1500-byte message will achieve better performance by sending two 750-byte messages. More such examples can be found in [29].

2.2 Scalability

MPI implementers must bear in mind that the number of processes in an MPI application may no longer be limited to a few hundred or a few thousand. Machines with much larger numbers of processors already exist. For example, the IBM Blue Gene/L at Lawrence Livermore National Laboratory has 131,072 processors. Larger systems are expected in the near future. As a result, MPI implementations must pay close attention to aspects of their code that grow linearly with the number of processors. Such aspects include the size of internal data structures, the number of connections established during `MPI_Init`, and the complexities of algorithms used anywhere in the implementation. Connecting all processes to each other in `MPI_Init` is no longer an option. If the underlying network requires connections, they must be set up only if and when

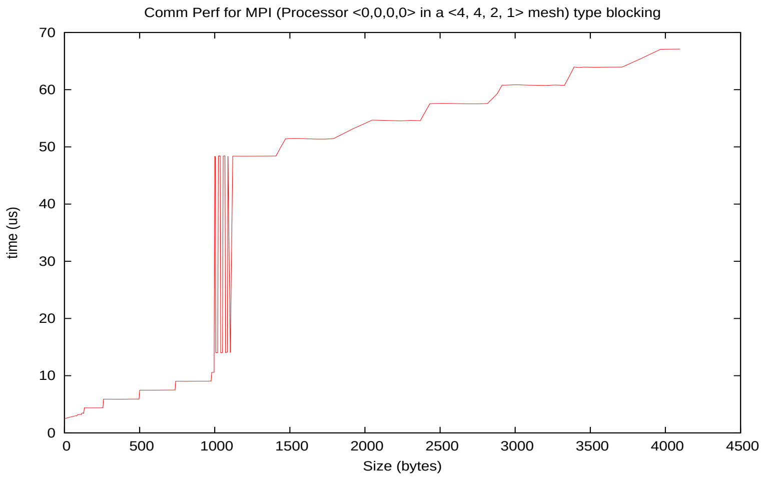


Fig. 1. Measured performance of short messages on IBM Blue Gene/L. Note the large jump around 1024 bytes; this is the transition from eager to rendezvous protocol in the MPI implementation.

needed, for example, when a process first tries to communicate with another process (MPICH2 already does this). Similarly, connections may need to be closed dynamically if they remain idle for a long time; and collective-communication algorithms, including all-to-all, may need to limit the number of connections.

2.3 Fault Tolerance

Closely tied to scalability is the need for increased tolerance to failure. As systems get larger, the probability of failure of components is larger. MPI implementations must improve their ability to handle failures, such as broken connections and dead processes, to the extent possible. A number of research efforts in fault-tolerant MPI implementation exist [2,7,13]. However, production MPI implementations need to improve their support for fault tolerance. In addition, fault tolerance often comes at a cost, and a careful balance must be struck between performance and fault tolerance.

2.4 Support for Debugging and Verification

MPI applications can be difficult to write and equally difficult to debug. To help application programmers, MPI implementations must provide better support for tools that help with debugging and verification. For example, MPI implementations must integrate better with parallel debuggers (e.g., TotalView) [6,9]. Auxiliary tools that help in debugging are also useful. An example is the `collchk` library [8] provided with MPICH2 that can check for inconsistencies in parameters passed to collective functions on different processes, such as the root for an `MPI_Bcast`. In a large and complex application with many MPI functions, `collchk` has helped find bugs that would otherwise have been very difficult to catch. A similar tool is described in [31]. Other tools also exist for checking program correctness, such as MARMOT [16], Umpire [32], and Intel Trace Analyzer and Collector [12], but more work is needed in this area.

Parallel programs are also prone to suffer from deadlocks and race conditions that may remain undetected for a long time because they are timing dependent [17]. For example, the byte-range locking algorithm proposed in [27] has a race condition that results in deadlock. It was discovered only a year later with the help of formal-verification methods [18]. Easy-to-use tools that use formal verification would be invaluable.

2.5 Virtual-to-Physical Topology Mapping

Today's large parallel machines, such as IBM Blue Gene/L and Cray XT3, have nodes arranged in a 3D torus topology. On such machines, it is more efficient to have MPI processes mapped on the nodes in a way that results in the majority of the communication taking place between nearest neighbors in the torus. MPI defines process-topology functions that allow users to create virtual process topologies and organize their communication among nearest neighbors on such topologies. However, the MPI implementation must efficiently map the virtual topology onto the physical processor layout such that nearest neighbors in the virtual topology are also nearest neighbors in the physical topology. This efficient mapping is often lacking in MPI implementations and must be provided. Applications may also need `MPI_COMM_WORLD` to be mapped appropriately on the machine.

2.6 Derived Datatypes

Derived datatypes in MPI allow users to specify noncontiguous memory layouts and thereby communicate noncontiguous data with a single function call. They are intended to provide higher performance than having the user pack all the data contiguously before calling MPI. However, MPI implementations have historically performed very poorly with derived datatypes, to the extent that users don't even think about using them. This situation defeats the purpose of having derived datatypes in the standard. Although some research efforts have optimized the processing of derived datatypes [21,30], their performance still often lags behind that of manual packing. One promising effort demonstrated higher performance than user packing by exploiting knowledge of the memory architecture of the machine and doing memory copies efficiently [5]. However, this work is yet not incorporated in the official release of MPICH2. More research, development, and incorporation into widely used MPI implementations clearly is needed for derived datatypes.

2.7 Collective Communication

MPI collective communication functions, such as broadcast and reduce, play a big role in helping applications achieve good performance. Although a lot of research has been done on collective communication algorithms [1,3,28] and some implementations have incorporated optimized algorithms [19,26], more work is still needed in some areas. For example, with the advent of multicore chips, MPI applications will routinely have multiple processes on a single node connected with multiple processes on other nodes by an interconnection network. Therefore, collective communication algorithms must be designed to effectively use such a hierarchical communication topology. Although research has been done on topology-aware collectives [14,15,22], not all production implementations have incorporated such algorithms yet. Furthermore, optimized algorithms are needed for the entire set of collectives in MPI, not just a select few.

The best algorithm for a particular collective communication function often depends on the message size and number of processes. In MPICH2, for example, the MPI collective functions use multiple algorithms, and one of them is selected for a specific message size and number of processes [26]. However, the cutoff points for switching between algorithms are based on measurements performed some time ago on one platform. They may not be right for other platforms. A better approach is needed that determines the right cutoff points for the specific machine being used. Dynamic tuning of algorithms may also be needed.

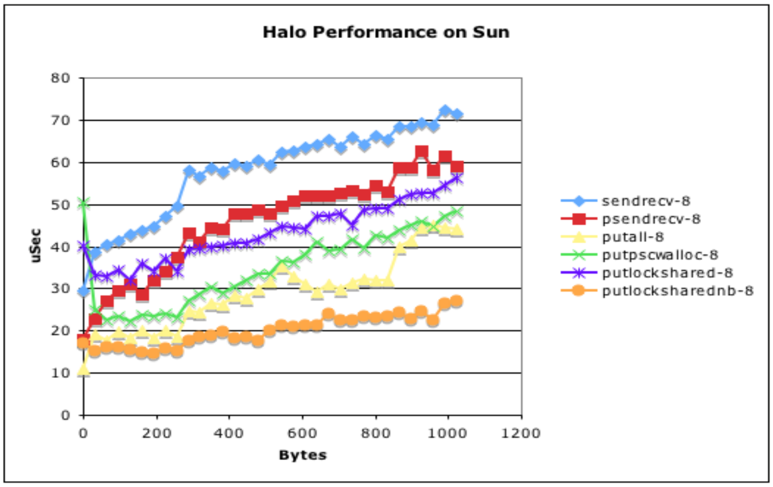
2.8 Parallel I/O

MPI-2 includes an interface for parallel file I/O, commonly referred to as MPI-IO. The most commonly used implementation of MPI-IO is ROMIO [20,24]. To our knowledge, almost all MPI implementations, except IBM's MPI for the SP, use ROMIO as the basis for MPI-IO. Although ROMIO has many optimizations that improve I/O performance substantially, such as data sieving and collective I/O [25], more work is needed in improving those algorithms and selecting the right internal buffer sizes for I/O and the right number of I/O aggregators on large systems. Applications would also benefit from a production-quality client-side caching system that can take advantage of the default weak consistency semantics of MPI-IO. Furthermore, many implementations do not yet support the portable external32 data format and user-defined data representations that are part of the MPI standard. These features are needed for standards compliance and for being able to write files that can be read on any architecture.

In the following sections, we discuss in greater detail two other areas needing improvement, namely, one-sided communication and support for multithreaded programs.

3 One-Sided Communication

The MPI-2 standard added one-sided communication operations to MPI. These operations offer a different programming model from the regular MPI-1 point-to-point operations: A process can directly write to or read from the memory of a remote process via *put* and *get* operations. A key feature of MPI one-sided communication is that data transfer and synchronization are separated. This



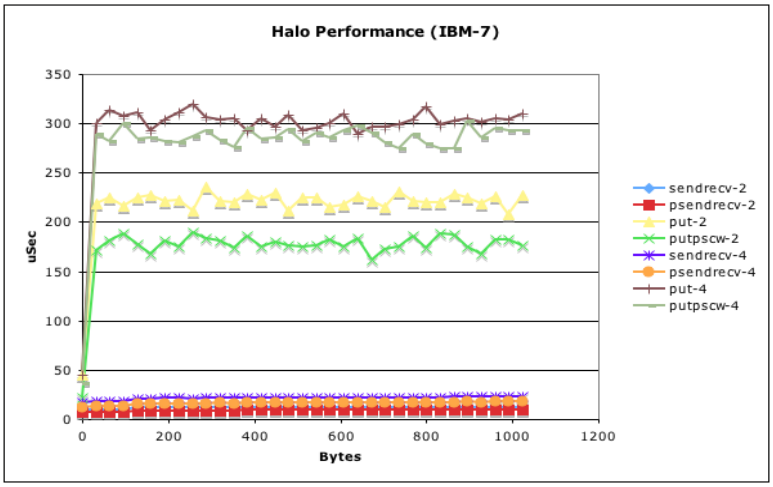


Fig. 2. Performance of one-sided communication for halo exchange on Sun Fire with 16 processes (top) and IBM p655+ with 7 processes (bottom). putall is the fence version with all assert options, putpscwalloc is the post-start-complete-wait synchronization with `MPI_Alloc_mem`, putlockshared is passive target with shared locks, and putlocksharednb omits the barrier that is necessary to ensure completion at the target.

feature allows multiple transfers to use a single synchronization operation, thus reducing the total overhead. MPI supports three synchronization methods: fence (collective synchronization), post-start-complete-wait (only communicating processes synchronize), and passive target (only the origin process calls lock-unlock functions).

To test how MPI implementations perform for one-sided communication, we wrote a benchmark that mimics the common “halo exchange” (or ghost-cell exchange) operation in applications that approximate the solution to partial differential equations. The code for this communication pattern, using MPI point-to-point communication, is as follows.

```
for (j=0; j<n_partners; j++) {
    MPI_Irecv( rbuffer[j], len, MPI_BYTE, partners[j], 0,
               MPI_COMM_WORLD, &req[j] );
    MPI_Isend( sbuffer[j], len, MPI_BYTE, partners[j], 0,
               MPI_COMM_WORLD, &req[n_partners+j] );
}
MPI_Waitall( 2*n_partners, req, MPI_STATUSES_IGNORE );
```

We wrote a number of versions of this benchmark with one-sided communication and using all three synchronization mechanisms. We ran the benchmark on a Sun Fire SMP at the University of Aachen and an IBM p655+ SMP at the San Diego Supercomputer Center using the native vendor MPI implementations (Sun and IBM).

Figure 2 shows a subset of the results on the Sun and IBM machines. The results on the Sun machine indicate that it is possible to get good performance with one-sided communication; in fact, on this system the performance with lock-unlock synchronization is better than with point-to-point communication. On the other hand, the IBM system performs very poorly for one-sided communication. With eight processes on an eight-node SMP, the one-sided communication performance was on the order of forty times slower than the point-to-point performance (data not shown). With seven processes on the same eight-node SMP, the one-sided communication performance is still poor (as shown in Figure 2) but an order of magnitude faster than with eight processes. The significant change in performance between eight and seven processes suggests that a thread is used for implementing the one-sided communication operations and that the implementation is not prepared to handle the case where there are more threads than processors. The results on the IBM machine demonstrate that efforts are needed to improve the performance of MPI one-sided communication.

Additional results for other MPI implementations and platforms can be found in [11].

4 Efficient Support for MPI_THREAD_MULTIPLE

MPI-2 allows users to write multithreaded MPI programs and defines the interaction between MPI and threads. MPI implementations that support the highest level of thread safety for user programs, `MPI_THREAD_MULTIPLE`, are becoming widely available. Thread safety does not come for free, however, because the implementation must protect certain data structures or parts of the code with mutexes or critical sections. Developing a thread-safe MPI implementation is a fairly complex task, and the implementers must make several design choices, both for correctness and for performance [10]. To simplify the task, implementations often focus on correctness first and performance later (if at all). As a

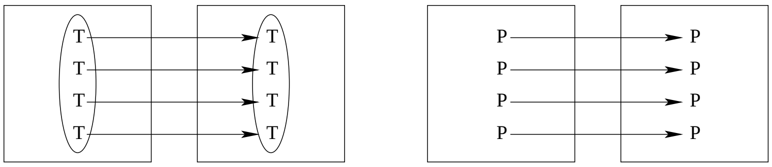


Fig. 3. Communication test when using multiple threads (left) versus multiple processes (right)

result, even though an MPI implementation may support multithreading, its performance may be far from optimized.

To determine how current implementations perform, we ran tests that measure the bandwidth and latency obtained when multiple threads of a process communicate with multiple threads of another process compared with multiple processes instead of threads (see Figure 3). We ran the tests on the Sun Fire and IBM p655+ SMPs and on a Linux cluster at Argonne National Laboratory. The cluster has nodes with two dual-core AMD Opterons and Gigabit Ethernet as the interconnect. We used the native vendor MPI implementations on the Sun and IBM machines and two implementations on the Linux cluster: MPICH2 and Open MPI.

The first test measures the cumulative bandwidth obtained and demonstrates how much thread locks affect the cumulative bandwidth; ideally, the multiprocess and multithreaded cases should perform similarly. Figure 4 shows the results. On the Linux cluster, the tests were run on two nodes, with all communication happening across nodes. We ran two cases: one where there were as many processes/threads as the number of processors on a node (four) and one where there were eight processes/threads running on four processors. Both cases show no measurable difference in bandwidth between threads and processes with MPICH2. With Open MPI, there is a decline in bandwidth with threads in the oversubscribed case. On the Sun and IBM SMPs, on the other hand, there is a substantial decline (more than 50% in some cases) in the bandwidth when threads were used instead of processes.

We also ran a version of the test that measures the time (latency) for individual short messages instead of concurrent bandwidth for large messages. Figure 5 shows the results. On the Linux cluster with MPICH2, there is a 20 μs overhead in latency when using concurrent threads instead of processes. With Open MPI, the overhead is about 30 μs. With Sun and IBM MPI, the latency with threads is about 10 times the latency with processes.

The overhead of threads is much more noticeable on the shared-memory machines because the overall message-passing performance on those machines is high (very low latency and very high bandwidth). Minimizing the overhead of thread-related locking in such environments is a difficult problem, and more research is needed in this area.

Additional results for several other tests can be found in [23].

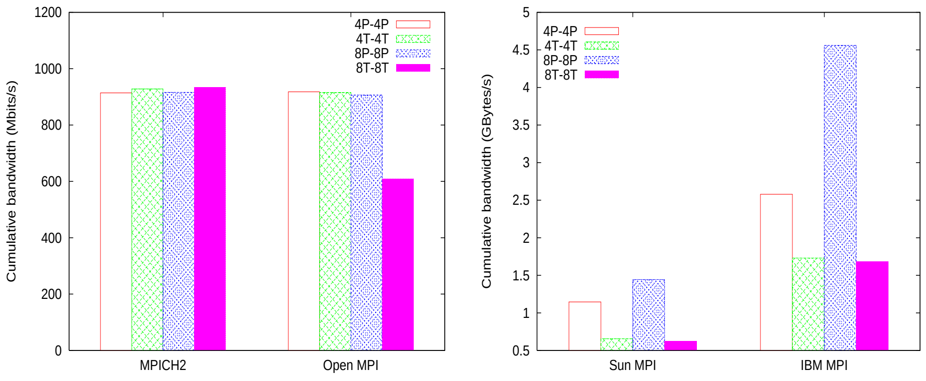


Fig. 4. Concurrent bandwidth test on Linux cluster (left) and Sun and IBM SMPs (right)

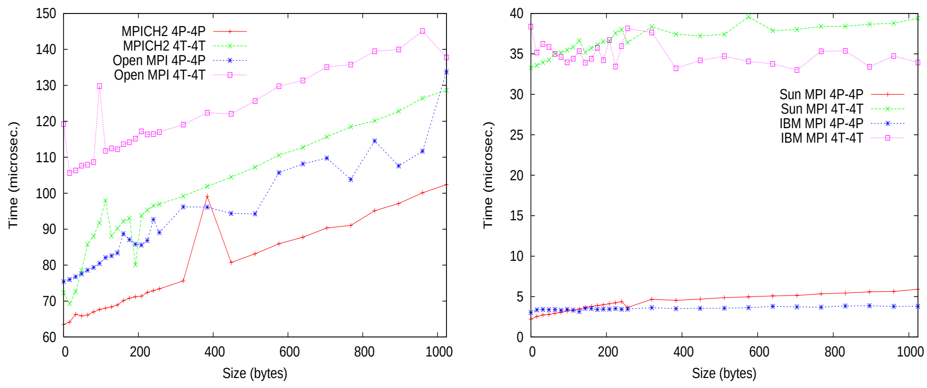


Fig. 5. Concurrent latency test on Linux cluster (left) and Sun and IBM SMPs (right)

5 Summary

Although the MPI standard has existed for a long time and MPI implementations have matured over the years, many areas remain in which MPI implementations still need improvement. Some of these improvements are necessitated by new developments in parallel systems, such as very large scale (more than 100,000 processors) and the advent of multicore chips. Others are just hard topics that need more work. Special efforts are needed in the areas of scalability, fault tolerance, one-sided communication, support for multithreading, and topology awareness. Continued research in these areas and incorporation of research results into production implementations will enable users to take full advantage of the enormous power of the leading supercomputers, which is rapidly approaching 1 petaflop/s.

Acknowledgments

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

References

1. Barnett, M., Gupta, S., Payne, D., Shuler, L., van de Geijn, R., Watts, J.: Interprocessor collective communication library (InterCom). In: Proceedings of Supercomputing '94 (November 1994)
2. Bosilca, G., Bouteiller, A., Cappello, F., Djilali, S., Fedak, G., Germain, C., Herault, T., Lemarinier, P., Lodygensky, O., Magniette, F., Neri, V., Selikhov, A.: MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In: Proceedings of SC 2002, IEEE, Los Alamitos (2002)
3. Bruck, J., Ho, C.-T., Kipnis, S., Upfal, E., Weathersby, D.: Efficient algorithms for all-to-all communications in multiport message-passing systems. IEEE Transactions on Parallel and Distributed Systems 8(11), 1143–1156 (1997)
4. Buntinas, D., Mercier, G., Gropp, W.: Implementation and shared-memory evaluation of MPICH2 over the Nemesis communication subsystem. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 4192, pp. 86–95. Springer, Heidelberg (2006)
5. Byna, S., Gropp, W., Sun, X.-H., Thakur, R.: Improving the performance of MPI derived datatypes by optimizing memory-access cost. In: Proceedings of the IEEE International Conference on Cluster Computing (Cluster 2003)(December 2003), pp. 412–419 (2003)
6. Cownie, J., Gropp, W.: A standard interface for debugger access to message queue information in MPI. In: Margalef, T., Dongarra, J.J., Luque, E. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 1697, pp. 51–58. Springer, Heidelberg (1999)
7. Fagg, G.E., Dongarra, J.J.: FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In: Dongarra, J.J., Kacsuk, P., Podhorszki, N. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 1908, pp. 346–353. Springer, Heidelberg (2000)
8. Falzone, C., Chan, A., Lusk, E., Gropp, W.: Collective error detection for MPI collective operations. In: Di Martino, B., Kranzlmüller, D., Dongarra, J.J. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 3666, pp. 138–147. Springer, Heidelberg (2005)
9. Gottbrath, C., Barrett, B., Gropp, W.D., Lusk, E., Squyres, J.: An interface to support the identification of dynamic MPI-2 processes for scalable parallel debugging. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 4192, pp. 115–122. Springer, Heidelberg (2006)
10. Gropp, W., Thakur, R.: Issues in developing a thread-safe MPI implementation. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 4192, pp. 12–21. Springer, Heidelberg (2006)

11. Gropp, W., Thakur, R.: Revealing the performance of MPI RMA implementations. Technical Report ANL/MCS-P1419-0507, Mathematics and Computer Science Division, Argonne National Laboratory (May 2007)
12. Intel Trace Analyzer and Collector 7.0 for Linux `http://www.intel.com`
13. Jitsumoto, H., Endo, T., Matsuoka, S.: ABARIS: An adaptable fault detection/recovery component framework for MPIs. In: Proceedings of 12th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems (DPDNS '07) in conjunction with IPDPS 2007 (March 2007)
14. Karonis, N., de Supinski, B., Foster, I., Gropp, W., Lusk, E., Bresnahan, J.: Exploiting hierarchy in parallel computer networks to optimize collective operation performance. In: Proceedings of the Fourteenth International Parallel and Distributed Processing Symposium (IPDPS '00), pp. 377–384 (2000)
15. Kielmann, T., Hofman, R.F.H., Bal, H.E., Plaat, A., Bhoedjang, R.A.F.: MagPIe: MPI's collective communication operations for clustered wide area systems. In: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99), pp. 131–140. ACM Press, New York (1999)
16. Kramme, B., Müller, M.S., Resch, M.M.: MPI application development using the analysis tool MARMOT. In: Bubak, M., van Albada, G.D., Sloot, P.M.A., Dongarra, J.J. (eds.) ICCS 2004. LNCS, vol. 3038, pp. 464–471. Springer, Heidelberg (2004)
17. Lee, E.A.: The problem with threads. Computer 39(5), 33–42 (2006)
18. Pervez, S., Gopalakrishnan, G., Kirby, R.M., Thakur, R., Gropp, W.: Formal verification of programs that use MPI one-sided communication. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 4192, pp. 30–39. Springer, Heidelberg (2006)
19. Ritzdorf, H., Träff, J.L.: Collective operations in NEC's high-performance MPI libraries. In: Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS 2006) (April 2006)
20. ROMIO: A high-performance, portable MPI-IO implementation. `http://www.mcs.anl.gov/romio`
21. Ross, R., Miller, N., Gropp, W.D.: Implementing fast and reusable datatype processing. In: Dongarra, J.J., Laforenza, D., Orlando, S. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 2840, pp. 404–413. Springer, Heidelberg (2003)
22. Sistare, S., vandeVaart, R., Loh, E.: Optimization of MPI collectives on clusters of large-scale SMPs. In: Proceedings of SC99: High Performance Networking and Computing (November 1999)
23. Thakur. R., Gropp, W.: Test suite for evaluating performance of MPI implementations that support MPI_THREAD_MULTIPLE. Technical Report ANL/MCS-P1418-0507, Mathematics and Computer Science Division, Argonne National Laboratory (May 2007)
24. Thakur, R., Gropp, W., Lusk, E.: On implementing MPI-IO portably and with high performance. In: Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems, pp. 23–32. ACM Press, New York (1999)
25. Thakur, R., Gropp, W., Lusk, E.: Optimizing noncontiguous accesses in MPI-IO. Parallel Computing 28(1), 83–105 (2002)
26. Thakur, R., Rabenseifner, R., Gropp, W.: Optimization of collective communication operations in MPICH. International Journal of High-Performance Computing Applications 19(1), 49–66 (2005)

27. Thakur, R., Ross, R., Latham, R.: Implementing byte-range locks using MPI one-sided communication. In: Di Martino, B., Kranzlmüller, D., Dongarra, J.J. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 3666, pp. 120–129. Springer, Heidelberg (2005)
28. Träff, J.L.: A simple work-optimal broadcast algorithm for message-passing parallel systems. In: Kranzlmüller, D., Kacsuk, P., Dongarra, J.J. (eds.) PVM/MPI. LNCS, vol. 3241, pp. 173–180. Springer, Heidelberg (2004)
29. Träff, J.L., Gropp, W., Thakur, R.: Self-consistent MPI performance requirements. In: Technical report. Euro PVM/MPI 2007 (May 2007) (Submitted)
30. Träff, J.L., Hempel, R., Ritzdoff, H., Zimmermann, F.: Flattening on the fly: Efficient handling of MPI derived datatypes. In: Margalef, T., Dongarra, J.J., Luque, E. (eds.) PVM/MPI. LNCS, vol. 1697, pp. 109–116. Springer, Heidelberg (1999)
31. Träff, J.L., Worringen, J.: Verifying collective MPI calls. In: Kranzlmüller, D., Kacsuk, P., Dongarra, J.J. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users' Group Meeting. LNCS, vol. 3241, pp. 18–27. Springer, Heidelberg (2004)
32. Vetter, J.S., de Supinski, B.R.: Dynamic software testing of MPI applications with Umpire. In: Proceedings of SC2000: High Performance Networking and Computing (November 2000), pp. 70–79 (2000)

Implicit Transactional Memory in Kilo-Instruction Multiprocessors

Marco Galluzzi[1], Enrique Vallejo[2], Adrián Cristal[3], Fernando Vallejo[2], Ramón Beivide[2], Per Stenström[4], James E. Smith[5], and Mateo Valero[1,3]

[1] Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya
[2] Grupo de Arquitectura de Computadores, Universidad de Cantabria
[3] Barcelona Supercomputing Center
[4] Dept. of Computer Science and Engineering, Chalmers University of Technology
[5] Dept. of Electrical and Computer Engineering, University of Wisconsin-Madison

galluzzi@ac.upc.edu, enrique@atc.unican.es, adrian.cristal@bsc.es, fernando@atc.unican.es, mon@atc.unican.es, pers@ce.chalmers.se, jes@ece.wisc.edu, mateo@ac.upc.edu

Abstract. Although they have been the main server technology for many years, multiprocessors are undergoing a renaissance due to multi-core chips and the attractive scalability properties of combining a number of such multi-core chips into a system. The widespread use of multiprocessor systems will make performance losses due to consistency models and synchronization styles of popular programming models even more evident than they already are. Known architectural approaches to combat these losses are generally too complex, too specialized, or not transparent to software. In this article, we introduce *implicit transactional memory* as a generalized architectural concept to remove unnecessary performance losses caused by consistency models and synchronization styles. We show how the concept of *implicit transactions* can be implemented with low complexity by leveraging the multi-checkpoint mechanism of the Kilo-Instruction Processor. By relying on a general speculation substrate, this method supports even the strictest consistency model – sequential consistency – potentially as effectively as weaker models and it allows multiple threads to speculatively execute critical sections, beyond barriers and event synchronizations.

1 Introduction

Over the last decades, multiprocessors have played a key role in servers for a wide range of computational problems ranging from scientific/engineering tasks to information processing tasks, for example in database and web servers. Their importance will likely increase further as the available transistors on a chip will soon allow a midrange server of the 90s to fit on a single chip. Building large-scale multiprocessors will be a matter of connecting a limited number of such chips, for example in a non-uniform-memory-access (NUMA) configuration.

One reason for the success of shared-memory multiprocessors, as opposed to other parallel architecture styles, is their intuitive programming style (and relatively low

© Springer-Verlag Berlin Heidelberg 2007

parallelization effort). For example, parallel threads can be managed via a work queue, with inter-thread communication taking place through shared data structures. While this approach greatly reduces the effort required to achieve good load balance, the challenge for the programmer is to assure that data dependences among threads are respected. This is achieved by using *synchronization primitives*, e.g., locks and barriers. Regardless of the primitives chosen, synchronization typically leads to two orthogonal sources of performance loss: *memory access ordering* and *serialization* losses.

Regarding *memory access ordering*, event synchronization in many legacy codes may use regular loads and stores that cannot be distinguished from other non-synchronizing loads and stores. This approach will work correctly as long as the underlying machine supports a strict consistency model such as sequential consistency. Unfortunately, supporting a strict consistency model has performance and/or complexity implications because the architecture has to guarantee that individual loads and stores are *performed* in the order specified by the program of each thread to establish a consistent global execution order. While naïve implementations may stall the processor until each load or store completes, research [10][23] has shown that it is possible to allow a thread to have multiple outstanding memory accesses as long as reordering violations can be detected and their effects can be "undone". Conventional speculation support in wide-issue processors can be extended to accomplish this, but this approach will suffer from scalability problems. Specifically, the length of speculation must be extended to cover the entire memory access latency, approaching hundreds of processor cycles, so it would be necessary to support a thousand or more instructions simultaneously in flight.

It is well-known that memory access ordering losses can be reduced or eliminated by relaxed memory consistency models. Indeed, if synchronization primitives can be identified by the programmer or compiler and if this information can be passed to the hardware via special synchronization operations, one can allow reordering of non-synchronizing memory accesses without any speculation support. Apart from not being compatible with some legacy codes, such relaxed consistency models cannot eliminate the second source of performance loss: serialization loss. Serialization loss occurs when threads stall at a synchronization point, waiting for other threads to arrive. For example, a thread cannot enter a critical section until another thread releases the lock, or, a thread cannot execute beyond a barrier until all other threads have reached the barrier. This *serialization loss* can sometimes be avoided if the programmer spends more effort exploiting the inherent concurrency in the program. An alternative approach that removes the burden from the programmer is to let threads speculatively execute past synchronization points [13][19][22][24]. For example, it is semantically correct to allow multiple threads to execute a critical section as long as there are no data races, i.e., accesses from multiple threads to the same variable with at least one access being a store. Apart from relying on special-purpose speculation mechanisms, these proposals assume that critical section boundaries can be identified at the hardware level, giving the approach limited appeal.

In order to avoid memory access ordering and serialization losses, there is a need for a general speculation mechanism that has attractive scalability properties. Further, in order to be applicable to a broad software base, it should not require that synchronization primitives be explicitly identified by the programmer or compiler. In this article, we propose *implicit transactions*, i.e. sequences of memory operations executed in hardware as atomic units, without any software support and transparent to the programmer. Within an implicit transaction, a processor's memory operations can be performed in any order (subject to sequential program semantics). Each transaction begins with a checkpoint, i.e. a point to which architected state can be rolled back, if necessary. Then a thread executes speculatively beyond a checkpoint until it successfully arrives at the next checkpoint, which ends the implicit transaction. Then, the thread commits all its changes to the memory state by making its stores visible to the other threads (e.g. via a snoopy bus). At that time, if another thread detects a data race with the committing thread, then the current transaction of the conflicting (non-committing) thread will fail. At that point, the conflicting thread must roll back to its previous checkpoint and begin re-execution.

While other recent transactional memory proposals also rely on a speculation mechanism, they are either specialized for certain software idioms or otherwise rely on software to mark the start and end of a transaction explicitly [12][19][24], or they are restricted to a specific programming task, e.g. critical sections [22]. Many of them rely on a checkpointing mechanism to recover the state of the machine in case of a data race. Some superscalar architectures have based their operation in a checkpointing mechanism rather than a tracking ROB, such as CFP [25], or Kilo-Instruction Processors [3]. We will show that implicit transactions can be naturally integrated into superscalar processors by leveraging the multi-checkpoint mechanism of these innovative architectures. As it was shown in previous work [7], the complexity with which Kilo-Instruction Processors can manage thousands of in-flight instructions is remarkably low and is sufficient to hide long memory access latencies in multiprocessors. Thus, we will focus on the mechanisms that have been proposed for Kilo-Instruction Processors, though our schema will be equally applicable to other architectures such as CFP, which also makes use of multiple checkpoints on flight. There have been different proposals focused on multiprocessor aspects for checkpointed processors [16], but not on these precise architectures.

The remainder of the article proceeds as follows. In Section 2, we present the basic mechanisms in Kilo-Instruction Processors that support implicit transactions. Then, Section 3 explains how checkpointing is used in our design to create implicit transactions. Section 4 shows the implications of the selected design for cache coherence and memory consistency. Finally, in Section 5, we describe the application of speculation mechanisms for executing beyond a locked critical section or a closed barrier.

2 Kilo-Instruction Processors

The implicit transactional memory concept leverages some of the key mechanisms in the Kilo-Instruction Processor, especially its multi-checkpoint mechanism. Therefore, we first provide a summary of the Kilo-Instruction Processor.

Kilo-Instruction Processors [3] have a demonstrated ability to hide large latencies, specifically due to memory accesses, because the processor allows thousands of instructions to be in-flight simultaneously. In order to increase the number of in-flight instructions, however, the designer must increase the capacities of several resources, the most important ones being the re-order buffer (ROB), the instruction issue queues, the load/store queues and the physical registers. Unfortunately, simply up-sizing these structures is not feasible with current or near-term technology.

In order to overcome the difficulties of up-sizing critical structures, Kilo-instruction processors employ a number of techniques to arrive at a complexity-effective implementation. Such an approach is possible because critical resources are underutilized in present out-of-order processors [2],[15]. The main technique consists of multi-checkpointing long latency instructions instead of using a very large ROB. This way, instructions can release resources in an out-of-order fashion [4], thereby requiring fewer resources than in a ROB-based implementation. The second technique is the Slow Line Instruction Queue that employs a secondary instruction queue to which long-latency instructions can be off-loaded. This mechanism allows the regular instruction queue to remain small and fast. The third technique is called Ephemeral Registers [20], which is an aggressive register recycling mechanism that combines delayed register allocation and early register recycling and, in conjunction with multi-checkpointing and Virtual Tags, allows the processor to non-conservatively de-allocate resources.

During program execution, a checkpoint is taken at certain instructions, generally at branches that depend on a load miss or load instructions that are likely to miss in the L2 cache. The checkpoint is a snapshot of the processor state. If an exception or branch misprediction occurs, the state is rolled back to the closest checkpoint prior to the excepting instruction. Using a relatively small set of checkpoints for long flight-time instructions assures safe points of return and reduces ROB requirements considerably. Although effective, this may lead to a longer recovery time than with a conventional ROB. Therefore, to minimize the exception and misprediction penalty a reduced structure called a pseudo-ROB can be used. The pseudo-ROB only maintains the youngest in-flight instructions and allows quick misprediction recovery for these instructions in a manner similar to a conventional ROB. Because exceptions and branch mispredictions occur most frequently within these youngest instructions, the average recovery time is effectively reduced.

Given that multiple checkpoints are taken during program execution, the mechanism works as follows. As the instructions in the pipeline advance, in-flight instructions corresponding to the different checkpoints are executed. When these speculatively executed instructions finish, memory operations remain in the processor queues; otherwise they are completed by the processor, leaving results in their destination registers. When all the instructions corresponding to the oldest checkpoint are finished, the processor commits the checkpoint atomically, by removing memory operations from the load and store queues and committing all the results speculatively calculated. With this action, the results of speculative instructions may be observed by other processors and become globally performed.

3 Implicit Transactions

Given the multi-checkpoint mechanisms in the Kilo-Instruction Processors, we will now see how they naturally support implicit memory transactions. To simplify the discussion, we assume a multiprocessor system with Kilo-Instruction Processor nodes interconnected by a bus using a snoopy cache protocol. Other interconnection architectures will be briefly discussed later. Specifically, implicit transactions cause the instructions between two checkpoints to appear to the rest of the system as a single memory transaction. In particular, memory updates (the *store* instructions) associated with a transaction are managed as a group and are globally and atomically performed when the corresponding checkpoint commits.

We call these *implicit transactions* because they are automatically and transparently hardware delimited, by means of the checkpoint mechanism; a key point is that *every* instruction belongs to some implicit transaction. The programmer does not need to know that the system provides transactional behavior. Therefore, both the programming model and the instruction set are unaffected, and legacy binaries can be directly executed. Note that this idea differs from the concept of transaction that known transactional memory systems rely on [12][13][19][24], where transactions are considered as programming constructs that normally depend on instruction set support. At the same time, implicit transactions can support TCC-based programming models [12] where all the code is executed using transactions, by detecting a few special transaction boundary-marking instructions. This property allows the execution of traditional, as well as TCC-based programming models, using the same underlying hardware.

Now that the close relationship between a checkpoint and an implicit transaction in our system has been established, we will henceforth use the term *checkpoint* and the term *implicit transaction* (or just *transaction*) interchangeably. In the following we first describe a straightforward embodiment that implements implicit transactions and then discuss possible performance/complexity optimizations.

3.1 Basic Scheme

As in the multiple checkpoint mechanism for a single Kilo-Instruction Processor, all the in-flight instructions remain speculative until their corresponding checkpoint commits. This means that memory instructions remain in the processor queues and do not modify the local cache or the global memory, and they may be correctly rolled back. In a snoopy bus based system, after a checkpoint commits, all the stores are broadcast over the bus as a packet, thereby validating all the speculative memory updates in the checkpoint. The broadcast packet is snooped by remote processors, and if an address-match is found the remote processor rolls back to the checkpoint previous to its data use because it may have used a stale value.

Fig. 1 shows an example of the execution flow for four processors, P1 to P4, and their respective checkpoints. The processors execute different portions of code, taking different checkpoints as the execution advances. The oldest checkpoint for a given processor can commit when all of its corresponding instructions, i.e. the ones that come after the checkpoint and before the next checkpoint, are finished. In Fig 1, for example, P3 can commit checkpoint Chk31, when all the instructions up to Chk32 have been completed. Then, part of the commit process is an atomic bus broadcast of all stores, and in particular the cache tags that the processor has modified during the checkpoint execution; these are, the pending memory updates. “Atomic” simply means that after the processor gets access to the bus it does not release it until all the tags have been broadcast.

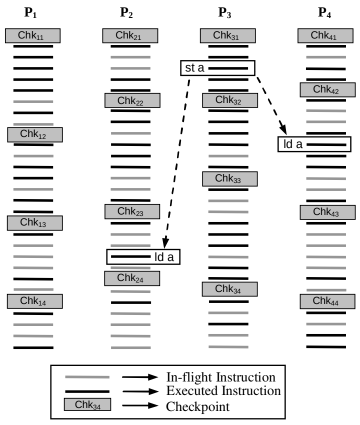


Fig. 1. Execution flow for 4 processors

Remote processors snoop the memory updates searching for a conflict with load instructions in their load queues; by definition these are speculatively executed because they are not yet committed. In case of an address match, the remote processor is forced to roll back to the appropriate checkpoint because it has used data from a shared location that, logically speaking, has been over-written by a “previous” store instruction. Conversely, if no such address matches are found, the remote processor continues execution, uninterrupted. In the example from Fig.1, the broadcast of a store to a given memory location “a” conflicts with two other processors that have already speculatively loaded from location “a”, and the loads have not yet committed. In this example, P2 is rolled back to Chk23, causing instructions from Chk24 to Chk23 to be discarded. Also P4 rolls back to Chk42, forcing its newest instructions to be discarded.

With such a checkpoint-based system, forward progress of the parallel application is always guaranteed because roll-backs, other than normal exceptions or branch mispredictions, occur only when one processor is committing a checkpoint and another one has a conflict. Therefore, at least one processor, the one committing, must make forward progress. The system is not concerned with conflicts that can exist during the execution of a transaction because they are not globally visible until the transaction commits.

This completes the explanation of how our design provides a correctness substrate, although it may be non-optimal from a performance point of view. The design is comprised of speculative execution, atomic validation, and the roll-back mechanism, which collectively ensure that the code is executed correctly.

3.2 Performance/Complexity Optimizations

The length of a transaction has a significant effect on the performance/complexity tradeoff. On the one hand, longer transactions allow a higher degree of memory operations re-ordering (as we will show in section 5), but on the other hand longer transactions may cause more roll-backs due to data races. We have used Simics [18] to obtain some initial statistics of the projected performance of our proposal, in terms of the proportion of rollbacks against the overall number of committed transactions. Fig. 2 shows an estimation of the proportion of rollbacks in different SPLASH-2 applications.

The results in Fig. 2 are not surprising: the proportion of rollbacks increases with the number of dynamic instructions that are associated with each checkpoint. It also increases with the number of processors that execute the same application, as each memory update can force more sharers to rollback. However, it is clear that for the checkpoint sizes that have been previously used, often under 512 dynamic instructions, the overhead that the protocol imposes will be negligible. With this figures in mind, we propose that transaction length should be adaptively adjusted in order to give good performance and avoid frequent roll-back scenarios. One could start from a fixed transaction length and adaptively shorten or lengthen it as more/fewer data races are observed. In case of frequent consistency violations, the length of the transactions decreases, also decreasing the probability of violations recurring. We are currently investigating heuristics to strike a good compromise between performance and resources needed. Note that having a mechanism to adaptively adjust the transaction length will also promote fairness among threads.

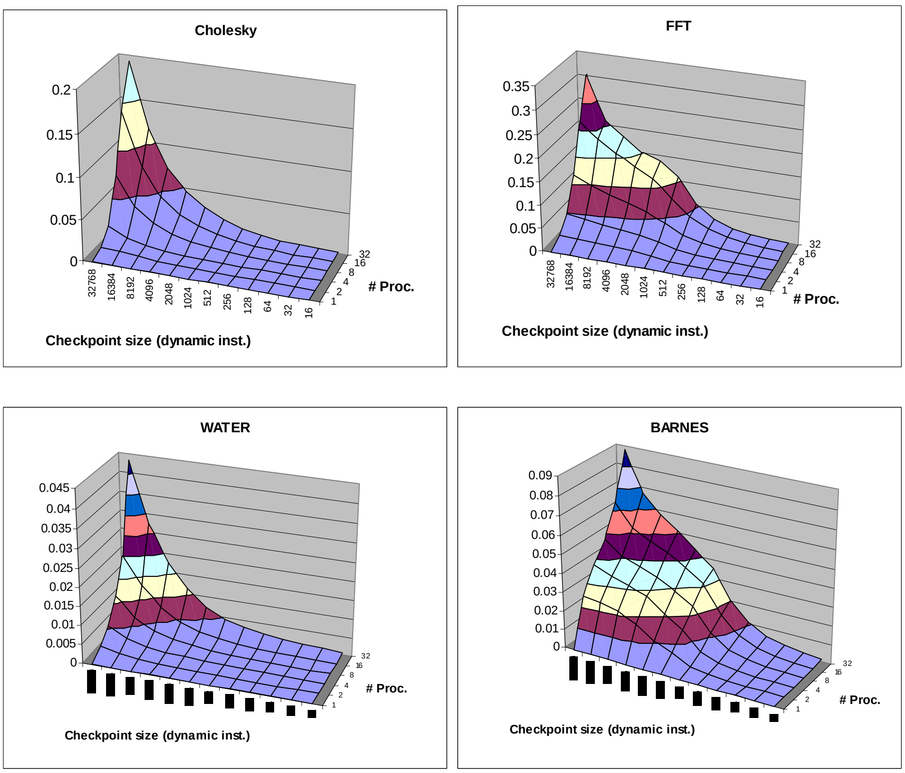


Fig. 2. Rollback proportion for different number of processors and transaction size

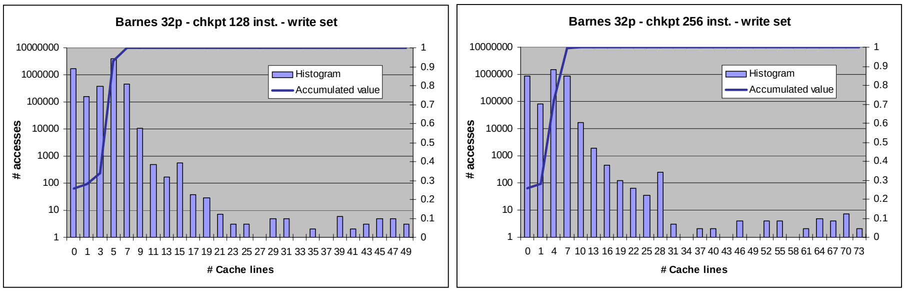


Fig. 3. Set size for the Barnes application with 32 processors

Another limiting parameter is the capacity of the cache, which has to store the speculative state of the transition read-set and write-set, similarly to [1]. Fig. 3 shows the capacity needed for the write set of two different checkpoint sizes on a simulation of Barnes on 32 processors: 128 and 256 dynamic instructions per checkpoint.

We can observe that in the first case merely 9 64-byte cache lines are enough to hold the entire write set of the transaction on the 99.8% of the times. With 256 instructions per checkpoint, just 10 cache lines are needed to maintain the write set on the 99.4%. A lack of resources on the first level cache would make the processor stall, waiting for previous checkpoints to commit and free their resources. However, the previous graphs suggest that, with current cache sizes, on most cases storage won't be a limit.

Our proposed baseline implementation of implicit transactions assumes that one can handle a very large number of in-flight memory operations. While the Kilo-Instruction Processor aims at providing this capability, more attention is needed with regard to the scalability of load/store queues. While encouraging solutions exist ([8],[21]), we are currently exploring solutions that better match our implicit transaction framework.

Finally, it is interesting to remark that we always refer to the size of the checkpoint in terms of dynamic instructions, not static ones. This is important because, otherwise, the system might block if two processors kept waiting for data that the other one had written but not committed on current transaction. The use of dynamic instructions makes both processors eventually take a new checkpoint while spinning and commit the previous one, guaranteeing forward progress.

4 Implicit Transactional Cache Coherence and Memory Consistency

This section demonstrates how cache coherence is maintained and that strong memory consistency models can be supported at high performance under the implicit transactional model.

4.1 Cache Coherence

A baseline snoopy cache coherence protocol works as follows. During the execution of a transaction, the local cache is not modified by local stores, only speculative loads are performed including loads that miss in the caches. Once a transaction commits, the pending stores are atomically performed by broadcasting the changes to the rest of processors, updating or invalidating the remote cache lines. When the stores from a transaction need to be broadcast, the corresponding processor will be granted access to write on the bus and release it only when all its memory updates are globally performed. This operation prevents other processor from broadcasting memory updates simultaneously and guarantees that a transaction is globally performed in an atomic fashion. Finally, if there is an update or invalidation of a remote cache line, all remote cache lines matching the snooped address will be updated or invalidated and the associated processors will roll back to the previous checkpoint. Note, however, that while transactions have to be committed one by one, individual nodes can still gain access to the bus to service cache misses. While a processor is validating the different stores of a transaction, other processors can interleave read requests without affecting correctness, as no update is performed on a read.

With implicit transactions, cache coherence protocols can be greatly simplified. For example, the basic protocol does not need to maintain cache lines in "shared" or "exclusive" state, as dictated by MESI-like cache coherence protocols. While this inevitably can result in higher traffic, simulation results in [12], assuming a similar transactional framework supported in software, show that the traffic is manageable. Further, there is room for improvement using a *write cache* [5] to coalesce multiple modifications of individual words in a cache line and using *silent store elimination* to be discussed later.

Since the basic protocol exploits the atomicity that a bus provides, generalizing it to a non-broadcast environment appears problematic. However, we are exploring extensions to scalable protocols. For example, a directory protocol should be extended with an arbitration mechanism. Arbitration would decide which processor can send its atomic update or invalidate packet to the rest of processors, avoiding the interleaving problem. Arbitration could be implemented, for instance, using a token-based mechanism. Briefly, the simplification of a normal coherence protocol under this approach would be the same as for a bus: no coherence state is potentially needed for each memory address. However, it would be good for performance to maintain the list of sharers on each directory entry in order to reduce the number of messages.

4.2 Memory Consistency

The memory consistency model employed in a shared-memory multiprocessor establishes the rules that must be followed when the system overlaps or reorders memory operations performed by the processors. The different consistency models offer a trade-off between programming simplicity and performance. At one end of the spectrum is Sequential Consistency (SC) which is generally considered to be the most natural and desirable programming model [14], but it is also the most restrictive. It guarantees that interleaved memory operations from different processors appear to execute in program order. A basic implementation of SC normally requires a processor to delay each memory access until the previous one is completed, while simple this approach clearly leads a low performance. More relaxed models, such as Release Consistency (RC) [9] provide higher performance, at the cost of not ensuring strict ordering of memory operations in hardware; this places more burden on the programmer to assure correct shared memory accesses. Other consistency models, lying between SC and RC, provide intermediate performance and ordering restrictions, such as Processor Consistency [11].

One of the most important properties of implicit transactional memory is that sequential consistency is simply enforced, potentially without performance loss due to memory ordering constraints and in a manner that is transparent to the software. The key observation is that a globally consistent transaction execution order is established, where transactions from a single thread respect the program order of that thread.

In Fig. 4 we give an example of a sequentially consistent global order of memory operations from two different processors; the operations are labeled {A1, A2, A3} for processor A, and {B1, B2, B3} for processor B. The third column shows a global order that respects the program orders from both processors.

Our proposed implementation groups memory accesses from each processor into *implicit transactions* by taking checkpoints. Thus, we can extend the definition of sequential consistency to such transactions, and require only transactions from each processor to be in order. The resulting global order will be an interleaved sequence of transactions that meets the basic definition of SC because it corresponds to one of the possible sequentially consistent global orders. Fig. 5 provides an example where we group instructions into transactions, labeled {TR_A1, TR_A2} for processor A, and {TR_B1, TR_B2} for processor B. The third column shows that respecting program order for these transactions will also respect program order for memory operations.

It is important to note that memory operations can be executed out-of-order, and therefore in Fig. 5 it would be possible for instructions in transaction "TR_A1" to be reordered, for example, as "A2, A3 and A1" instead of the order "A1, A2 and A3" as

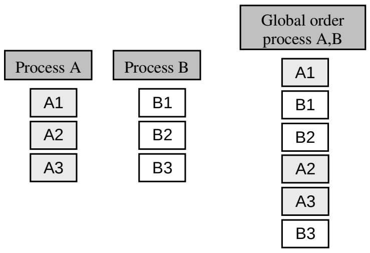


Fig. 4. Sequentially consistent reordering of memory operations from 2 processors

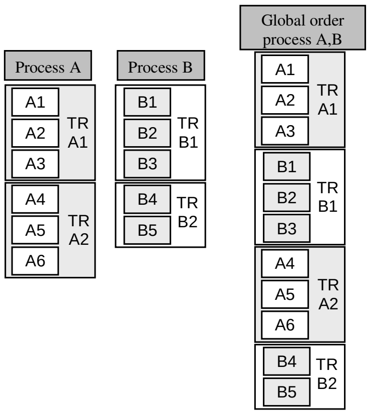


Fig. 5. Sequentially consistent reordering of checkpoints from 2 processors

shown. This avoids memory ordering performance losses without violating sequential consistency, as the speculative character of these operations ensures their validity, and they are committed atomically in a block. Because the Kilo-Instruction Processor can support hundreds of in-flight memory instructions with a constant checkpoint recovery time, this approach is expected to be significantly more scalable than the proposed SC++ implementation [10] that relies on a tracking buffer with a variable recovery time whose size increases with the number of instructions tracked.

5 Synchronizations and Implicit Transactions

We now turn our attention to ways that implicit transactions can avoid costly serialization losses by allowing multiple threads to speculatively execute beyond synchronization points. We first show a basic scheme that focuses on correct concurrent execution of critical sections and then discuss opportunities for optimizations of the basic scheme.

5.1 Basic Scheme

Fig. 6 shows an example in which a processor executes a lock-protected critical section under three checkpointing scenarios A, B, and C. When a processor reaches the lock, it checks the value of the lock and acquires it if it is free; otherwise it spins on the lock variable. If the lock is free, the store to the lock variable, i.e. the acquiring operation, remains in a speculative state in the processor queues until the store can commit. Meanwhile, a new checkpoint can be taken inside the critical section (Fig. 6a). Committing the transaction that finishes inside the critical section will validate the lock store instruction, thereby globally acquiring the lock, and forcing remote processors inside the critical section to roll back due to the invalidation of the lock variable. After that, remote processors will be inhibited from entering the critical section, due to the acquired lock. The validation of transaction T1 will force any other thread executing inside the critical section to roll back, as the lock variable is in the remote processor read set. When the processor validates transaction T2, the critical section gets unlocked and remote processors can start executing it.

The same invalidation happens if no checkpoint is taken until the critical section is unlocked (Figures 6b and 6c). In this case, the lock variable is also written. Thus, the validation of a transaction that has entirely executed a critical section will cause any other processor that is speculatively executing it to roll back, which means that only a single valid processor stays inside a critical section. This should be the most frequent case, due to the short nature of critical sections.

Finally, it is obvious that the system behaves correctly in the presence of flags and barriers. In effect, processors that reach a closed flag keep spinning on it, ensuring that no code is executed past the flag. These examples show that the correctness substrate ensures correct execution in presence of critical sections and barriers, independently of where checkpoints are taken.

5.2 Performance Optimizations

Roll-backs will sometimes happen even if lock variable accesses are the only sources of data races. In theory, this leads to an unnecessary performance loss. When one

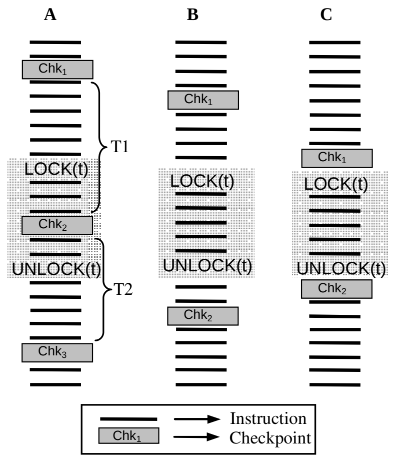


Fig. 6. Different checkpointing scenarios

processor has acquired a lock, the release the lock variable will write the same value to the lock variable that is already in this memory position (a *temporally silent store* [17]). We propose the use of two optional mechanisms that allow the system to detect these cases and naturally speculate through critical sections, thereby improving performance and reducing network traffic.

Store merging and silent store removal. The first mechanism detects and removes silent stores by detecting that a store does not globally modify memory, and removes that update from the update packet. An ordinary *store merging* mechanism (similar to that implemented in the Alpha 21164 [6]) reduces the number of stores to the same address within a transaction to only one, and *silent store removal* avoids broadcasting the remaining stores. Store merging therefore removes a store when a younger store is to the same address. If we apply this mechanism when all the instructions of the checkpoint are finished, we need not bother with race conditions such as load instructions between two merged stores. In such a case, the load instruction would have received the correct value via store forwarding mechanisms internal to the processor. The next step is to remove silent stores by leveraging the store-forwarding logic used in current processors. Store-forwarding searches the older stores before executing a load, and if an address match is found, the value of the store is forwarded to the load. In contrast, our mechanism searches older loads before executing a store, and if an address and value match is found the store is removed because it is silent. The mechanism is simple because it is only necessary to look for silent stores within a single transaction, with all the loads queued waiting to be committed and the stores queued waiting to be broadcast to the memory hierarchy. This mechanism, when applied to the checkpoint scenario B in Fig. 6, would remove the lock variable from the update packets and allow different instances of the same critical section to run in parallel avoiding processor serialization stalls, in the absence of other data races. For typically small critical sections, scenario B is likely to be the common case.

Lock detection mechanism. If locks are detected at run-time, we can ensure that no checkpoint is taken within a critical section, to enforce scenario B in Fig. 6. We make use of a *lock detection mechanism*, which dynamically detects typical Test&Set lock constructs. To maximize concurrency, the lock detection hardware forces a new checkpoint to be taken just before the lock, so that the lock remains open for the rest of the processors at the beginning of the new transaction. In addition, the hardware could detect the lock release, which is a simple store to the lock variable, and take a new checkpoint just after it. This makes the transaction length equal to the critical section, and avoids unnecessary roll-backs due to data races on data outside the critical section as in scenario C in Fig. 6.

Finally, we note that the proposed method preserves correctness independently of the length of the critical section. In contrast, TCC [12], which is a similar transaction approach, locally buffers all the memory updates corresponding to a certain transaction. In case of a buffer overflow, the processor in TCC must acquire the bus and not release it until the end of the transaction, thus blocking the rest of the system. In case of such an overflow, the system takes a new checkpoint and waits for the resources to free from previous checkpoints before continuing execution. Thus, in such a case, the behavior would be similar to scenario A in Fig. 6, which is correct but does not enable parallel execution.

5.3 Speculating Beyond Flags and Barriers

The proposed design can be adapted to speculate after barriers, in a similar fashion as [19]. Here, too, there is a need to detect the barrier code and take a new checkpoint just prior to it. Speculative execution starts after the barrier. All the transactions after the barrier remain fully speculative, meaning that none of them can be validated, as long as the barrier remains closed. This is ensured by setting a "pure speculative mode" in the processor.

To determine the time at which the barrier opens, the processor tracks the cache line containing the barrier variable, waiting for a cache event (an invalidation or an update of the line) before checking the value again. When the barrier opens, the "pure speculative mode" is disabled, and the processor can start committing all the transactions in the pipeline. Note that a remote invalidation of the speculatively marked line does not force a roll-back, but makes the processor re-check the variable. Of course, all the speculative execution done before opening the barrier variable has no effect on the consistency model because the commit only happens after the real barrier opens up, and the correctness substrate ensures that the cache contents remain valid up to the commit instant.

The expected performance improvement of this scheme depends on the average time the processors wait at a barrier, while in the previous case the processor can commit the critical section and continue execution. Thus, if the waiting time does not exceed the time needed for the pipeline to fill up and stall, performance will improve. As Kilo-instruction Processors are designed to support thousands of in-flight instructions, this good-case scenario will occur frequently. Furthermore, in case of a data race forcing a roll-back, this mechanism will prefetch the needed data, possibly reducing memory latencies encountered later on.

6 Concluding Remarks

This paper introduces a framework that makes Kilo-instruction Processors capable of executing parallel code in a transactional fashion, similar to the TCC model, but assumes no modification of the code nor the programming model. Our model maintains Sequential Consistency with a low hardware cost, a high performance potential, and a reduced bus overhead. The hardware requirements are low, as most of the mechanisms are already proposed for Kilo-Instruction Processors, and the processor model is simplified thanks to the transactional behavior.

Our model also enables speculative execution in critical sections and beyond barriers, reducing performance losses that these constructs cause in parallel programs. This, together with the advantages of transactional behavior, will provide high performance with no required code modifications.

Acknowledgments

This work has been supported by the Ministry of Education of Spain under contracts TIN-2004-07739-C02-01 and TIN2004-07440-C01-01 and grants AP2003-0539 (M. Galluzzi) and AP-2004-6907 (E. Vallejo), the HiPEAC European Network of Excellence, and the Barcelona Supercomputing Center. J. E. Smith is partly supported by the NSF grant CCR-0311361. Per Stenström is partly supported by the Swedish Research Council under contract VR 2003-2576.

References

1. Chung, J.W., et al.: The Common Case Transactional Behavior of Multithreaded Programs. In: Proc. of the 12nd HPCA (2006)
2. Cristal, A., Martínez, J.F., Llosa, J., Valero, M., Case, A.: for Resource-conscious Out-of-order Processors. In: IEEE TCCA Comp. Architecture Letters, 2, (October 2003)
3. Cristal, A., et al.: Kilo-instruction Processors: Overcoming the Memory Wall, In IEEE Micro Magazine 25(3), 48–57 (2005)
4. Cristal, A., Ortega, D., Llosa, J., Valero, M.: Out-of-Order Commit Processors. In: Proc. of the 10th HPCA (February 2004)
5. Dahlgren, F., Dubois, M., Stenström, P.: Combined Performance Gains of Simple Cache Protocol Extension. In: Proc. of 21st ISCA, pp. 187–197 (1994)
6. Edmondson, J.H., et al.: Internal organization of the Alpha 21164, a 300-MHz 64-bit quad-issue CMOS RISC microprocessor, Digital Technical Journal, vol. 79(1) (1995)
7. Galluzzi, M., et al.: A First Glance at Kilo-Instruction based Multiprocessors. In: Proc. of the 1st Conf. on Computing Frontiers, pp. 212-221 (April 2004)
8. Gandhi, A., et al.: Scalable Load and Store Processing in Latency Tolerant Processors. In: Proc. of the 32nd Int'l Symp. on Computer Architecture (ISCA'05), pp. 446-457 (June 2005)
9. Gharachorloo, K., et al.: Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In: Proc. of the 17th ISCA (1990)
10. Gniady, C., Falsafi, B., Vijaykumar, T.N.: Is SC + ILP = RC? In: Proc. of the 26th Int'l Symp. on Computer Architecture (ISCA'99) (1999)

11. Goodman, J.R.: Cache Consistency and Sequential Consistency, Technical Report no.61, SCI Committee (March 1989)
12. Hammond, L., et al.: Transactional Memory Coherence and Consistency. In:Proc. of the 31st Int'l Symp. on Computer Architecture (ISCA'04), pp. 102-113 (June 2004)
13. Herlihy, M., Moss, J.E.B.: Transactional Memory: Architectural Support for Lock-Free Data Structures. In: Proc. of the 20th ISCA, pp. 289–300 (1993)
14. Mark, D.: Multiprocessors Should Support Simple Memory-Consistency Models. IEEE Computer journal 31(8), 28–34 (1998)
15. Karkhanis, T., Smith, J.E.: A Day in the Life of a Data Cache Miss. In: Proc. of the 2nd Workshop on Memory Performance Issues (WMPI 2002) (May 2002)
16. Kırman, M., et al.: Cherry-MP: Correctly Integrating Checkpointed Early Resource Recycling in Chip Multiprocessors. In: Proc. of the 38th MICRO (November 2005)
17. Lepak, K.M., Lipasti, M.H.: Temporally Silent Stores. In: Proc. of the 10th ASPLOS, (October 2002)
18. Magnusson, P.S., et al.: Simics: A Full System Simulation Platfor. IEEE Computer 35(2), 50–58 (2002)
19. Martínez, J., Torrellas, J.: Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications. In: Proc. of 10th ASPLOS (October 2002)
20. Monreal, T., et al.: Delaying Physical Register Allocation Through Virtual-Physical Registers. In: Proc. of the 32nd MICRO, pp. 186–192 (November 1999)
21. Park, I., Ooi, C.-l., Vijaykumar, T.N.: Reducing design complexity of the load-store queue. In: Proc. of the 36th MICRO, pp. 411–422 (December 2003)
22. Rajwar, R., Goodman, J.R.: Transactional Lock-Free Execution of Lock-Based Programs. In: Proc. of the 10th ASPLOS (October 2002)
23. Ranganathan, P., Pai, V.S., Adve, S.: Using Speculative Retirement and Larger Instruction Window to Narrow the Performance Gap Between Memory Consistency Models. In: Proc. of the 9th SPAA, pp. 199–210 (June 1997)
24. Rundberg, P., Stenström, P.: Speculative Lock Reordering. In: Proc. of Int'l Parallel and Distributed Processing Symp (IPDPS'03) (April 2003)
25. Srinivasan, S.T., et al.: Continual flow pipelines. In: Proc. of the ASPLOS, pp. 107–119, Boston, MA (October 2004)

Design of a Low–Power Embedded Processor Architecture Using Asynchronous Function Units

Yong Li, Zhiying Wang, Xuemi Zhao, Jian Ruan, and Kui Dai

National University of Defense Technology, School of Computer,
Changsha, Hunan 410073, P.R. China

Abstract. Efficiency and flexibility are crucial features of processors in the embedded systems. The embedded processors need to be efficient in order to achieve real-time requirements with low power consumption for specific algorithms. And the flexibility allows design modifications in order to respond to different applications. As the superset of traditional very long instruction word (VLIW) architecture, Transport Triggered Architecture (TTA) offers a cost-effective trade-off between the size and performance of ASICs and the programmability of general-purpose processors. The main advantages of TTA are its simplicity and flexibility. In TTA processors, the special function units can be utilized to increase performance or reduce power dissipation. In this paper, we design a low-power processor architecture using asynchronous function units based on TTA. The processor core is globally synchronous and locally asynchronous implementation using not only synchronous function units but also asynchronous function units. We solve the problem that use asynchronous circuits in TTA that is only synchronous design environment. The test result shows that this processor has lower power dissipation and higher performance than its pure synchronous version that only uses synchronous function units.

1 Introduction

In recent years, special-purpose embedded systems have become one very important area of the processor market. Digital signal processor (DSP) offer flexibility and low development costs, but it has limited performance and typically high power dissipation. Field programmable gate arrays (FPGA) combine the flexibility and speed of application specific integrated circuit (ASIC), but it cannot compete with the energy efficiency of ASIC implementations. Application Specific Instruction Processor (ASIP) is designed to perform certain specific tasks as efficiently as possible to solve this problem [1]. But during the ASIP design process, there are some difficulties such as the instruction set and the architecture, development of software retargetable compilation and so on.

To solve these problems, This paper studies an embedded low-power processor architecture based on Transport Triggered Architecture according to the ASIP design flow. For the specific application, TTA can provide both flexibility and configurability during the ASIP design process. In order to exploit operation parallelism as much as possible and reduce the dynamic power consumption of the processor, custom asynchronous function units are studied.

© Springer-Verlag Berlin Heidelberg 2007

Since the early days, asynchronous circuits have been used in many interesting applications. There were many successful examples of asynchronous processors, which were described in [2], [3], [4], [5], [6], [7] and [8] et al. The results show that asynchronous circuits have advantages of low power consumption and high performance. In the embedded systems that are sensitive to power dissipation, there is a problem for us to solve that how to make our processors have lower power consumption without performance loss. There were many ways but no one used asynchronous circuits in TTA. In this paper, we attempt to use asynchronous function units in TTA in order to implement a low-power embedded processor architecture taking advantage of asynchronous circuits and TTA. This novel processor architecture may be viewed as globally synchronous locally asynchronous implementation. The evaluation results show that it has higher performance and lower power dissipation than TTA that only uses synchronous circuits.

The rest of this paper is organized as follows. Section 2 briefly describes the Transport Triggered Architecture. Section 3 describes the implementation of custom asynchronous function units. The target architecture is designed in Section 4. Next, two processor core are simulated, performance and power dissipation results are presented. The last section gives the conclusion and the future work.

2 Transport Triggered Architecture

Transport Triggered Architecture that proposed by Henk Corporaal et al can be viewed as a superset of traditional VLIW architecture [9]. A TTA processor consists of a set of function units and register files are connected to an interconnection network, which connects the input and output ports. As compared to conventional processor architectures, in the TTA programming model, the program specifies only the data transports (MOVE) to be performed and the interconnection network is visible to the software level when developing TTA applications. Each function unit of TTA may have one and more operand registers (O) and result registers (R), but only one trigger register (T). For example, we show how a three address registers ADD instruction translates into MOVE operations:

```
ADD R3, R2, R1 =>
         R1 -> O; R2 -> T; R -> R3.
```

First, the values in R1 and R2 have been transported to the operand register and trigger register of the adder respectively. When the value of R2 is transported to the trigger register, the. function units will begin to work. After some time (depending on the latency of the function unit) the result in result register will be moved to R3. In addition, one TTA instruction always contains several parallel data move operations after software optimization.

The structure of the TTA processor is very simple, as shown in Fig. 1 (a). The data path part consists of Register Files (RF), Load Store Units (LSU) and Function Units (FU), which are connected to an interconnection network. The control path part consists of Instruction Fetch Unit (IFetch) and Instruction Decode Unit (IDecode). The Instruction Decode Unit can generate the control signals of the interconnection network and

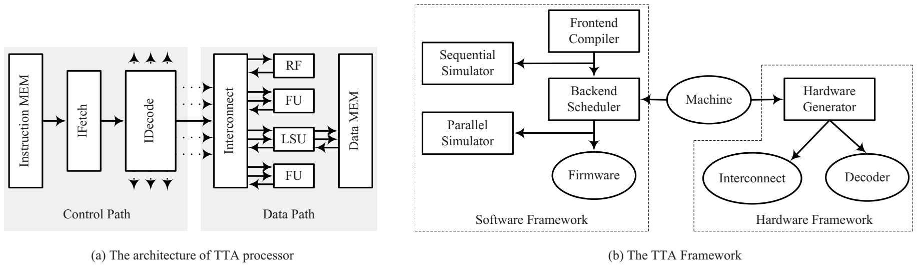


Fig. 1. (a) The architecture of TTA processor. (b) The TTA framework.

immediate. According to different applications, the differences of the application specific TTA processors are the number and type of the function units, register files, buses and buses connections. This brings the flexibility of the architecture design.

As shown in Fig. 1 (b), the TTA framework contains the software framework and hardware framework [10]. In TTA framework, the machine description language is defined. The application specific processor can be described in this language and the Machine file will be generated. Based on the generic TTA architecture, the frontend compiler can generate the sequential code. According to the Machine file, the backend scheduler can convert the sequential code into efficient parallel code for a given TTA target processor. The sequential simulator and parallel simulator can simulate the sequential code and parallel code respectively. These simulators are used to verify the code and evaluate the performance. In hardware framework, the HDL (hardware description languages) description of the decoder and interconnect network can be generated. These HDL description can be provided to other EDA tools.

Because of the flexibility and simplicity of the TTA architecture, the designer can customize special operation into the instruction set by designing special function unit to the architecture. Thus, some bottleneck operations in the applications can be implemented by special function unit so as to increase the performance. In the same way, our asynchronous function units can be utilized easily to reduce power dissipation in the TTA architecture.

3 Application Analysis and Asynchronous Function Units

In CMOS circuits, power dissipation is proportional to square of the supply voltage [11]. Therefore, a good energy-efficiency can be achieved by aggressively reducing the supply voltage [12] but unfortunately this results in low circuit performance. Clock gating technique can be used to reduce the power consumption of non active function units. Significant saving can be expected on units with low utilization But the clock gating technique also suffers from performance loss [13].

Asynchronous circuits have characteristics that differ significantly from those of synchronous circuits, such as high operating speed, less emission of electro-magnetic noise and low power consumption. Asynchronous circuits can be used to reduce the power consumption of the processor due to fine-grain clock gating and zero standby power consumption [14].

Asynchronous circuits only operate when data are ready, and realize the instant change between idle state and busy state, that is to say, the unused components will be power-down automatically. This characteristic of asynchronous circuits is propitious to low-power integrated circuits design.

According to application analysis, the statistics of major operations will be presented. If we will implement the asynchronous function units to execute the operations that are frequently used in these applications, the power consumption may be reduced.

3.1 Application Analysis

Because the architecture is determined by the characteristics of the application set, the first step of the architecture design is to analyze the application. By the analysis of different digital signal processing applications, paper [15] finally chosen 6 kernel applications as the representative set, as shown in Table 1.

Table 1. The DSP kernel application set

No.	Name	Brief Description
1	FFT	Fast Fourier Transform
2	FIR	Finite Impulse Response filter
3	IDCT	Inverse Discrete Cosine Transform
4	MAT-MUL	Multiply of two matrix
5	RECIP	Get the reciprocal of one integer
6	MAXIDX	Get the index of the maximum value of a vector

These kernel applications are frequently used in various embedded applications. The type and amount of the major operations in the application determines those of the function units in the TTA processor. The operations are divided into several types:

– add/sub: add operation and sub operation
– mul: multiply operation
– memory: load and store operation
– shift: shift operation
– logic: logic operation, such as and, or compare operation
– misc: some trivial miscellaneous operations

The proportions of the six major operations are shown in Fig. 2.

According to the type of the major operations, designer can quickly decide what function unit to implement; similarly, the amount of the function units is decided to the proportion of the equivalent operations. As shown in Fig. 2, we find that add/sub operation and multiply operation are common in the DSP applications, so the asynchronous adder and multiplier are implemented in order to reduce the power dissipation.

3.2 Asynchronous Adder

In this work, we implemented a 32-bit asynchronous adder. This asynchronous function unit employ micropipeline architecture that advocated by Sutherland in his paper [16].

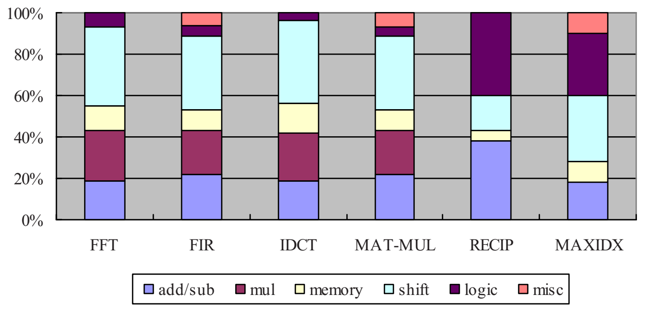


Fig. 2. The proportions of the major operations

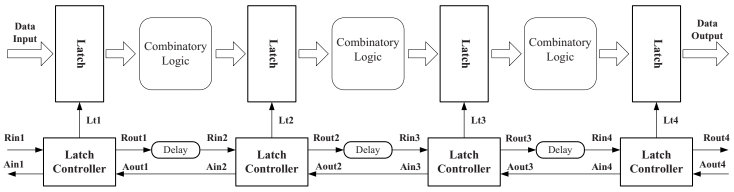


Fig. 3. The micropipeline architecture of asynchronous adder that has four stages

In asynchronous micropipeline, the global clock is replaced by a local communication protocol. The communication protocols employed in a micropipeline can be either a 2-phase or 4-phase signaling. The control circuits, called handshake circuits, realize the communication protocol locally inside a micropipeline between adjacent stages. The control circuits can be described as STG (signal transition graphs) [17] and synthesized by Petrify [18].

This asynchronous adder employs four micropipeline stages, and the architecture can be illustrated in Fig. 3.

This adder is composed of a traditional combinatorial circuit and a control circuit. To the designer the trade-off is between the performance and area. We employe ripple-carry adder, which performance is limited but this implementation can satisfy the requirement and save layout area.

In Fig. 3, each latch controller can generate the local clock, such as Lt1, Lt2, Lt3, Lt4, to control the latches of each stage. The matching delay elements provides a constant delay that matches the worst case latency of combinatory logic in each stage. In addition to the combinatorial circuit itself, the delay element represents a design challenge. In our procedure, the timing analysis has been done and custom delay cells is implemented to provide the constant delay.

3.3 Asynchronous Multiplier

In [19], a 32-bit asynchronous multiplier based on radix-2 Booth algorithm has been presented. The asynchronous multiplier used booth decoding and has the same

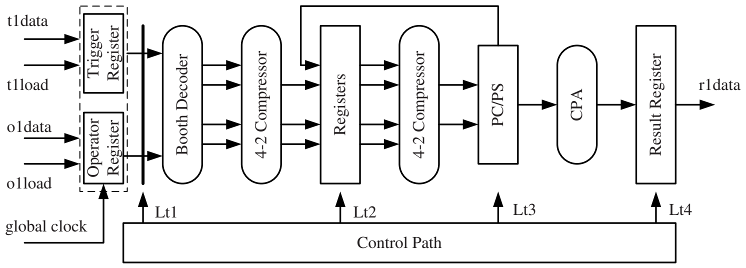


Fig. 4. The architecture of asynchronous multiplier

performance to the synchronous version but lower power dissipation. Similar to the asynchronous adder, the multiplier employs four stages micropipeline architecture and matching delay elements. The architecture of the asynchronous multiplier is shown in Fig. 4.

As shown in Fig. 4, the asynchronous multiplier has one operator register (O), one trigger register (T) and one result register (R). The operator register and trigger register are both controlled by global clock but the internal data path is controlled by local clock. When data is transferred to the trigger register, the multiplier will be triggered to work. The latency of the asynchronous multiplier need to be converted to cycles of the global clock. For example, if the latency of the asynchronous function units is equal to 20ns and the global clock period is 5ns, we should define that the latency of this asynchronous function units is 4 cycles in TTA Machine file. To TTA scheduler, it will take 4 cycles for this function unit to complete the operation. After 4 cycles, the scheduler can read the result register and send the result to other units that need it. But, if the latency of asynchronous unit is 16ns, we also have to define the latency is 4 cycles. It may brings performance loss, but it is a worthy trade-off between performance and power consumption to some systems that are sensitive to power dissipation.

Using global clock to control the input registers and trigger signal to start the operation can solve the problem that how to use asynchronous function units in synchronous TTA environment. This means we should only modify the interface of asynchronous function unit in order to use it quickly in TTA framework.

4 Implementation of the Architecture

According to the parallel simulator of the TTA framework, the parallelism upper bound is determined. It means how many buses should be used. The amount of average active registers shows how many registers should exist at the same time both to save the hardware cost and to keep the performance.

In this work, for example, we select application of MAT-MUL that multiplies of two matrix for test. Because in MAT-MUL, the add, mul, shift are the major operations, we can implement the processor core quickly using simple integer function units. Of course, the TTA processor core can execute the other applications by quickly adding some function units, such as some floating point function units. Taking into account

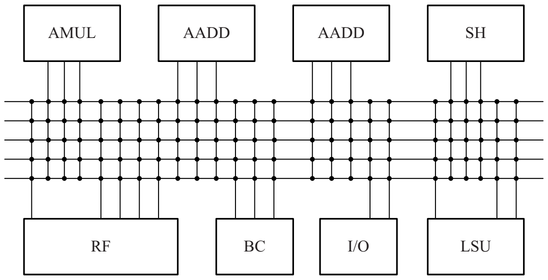


Fig. 5. The architecture of the processor core

that the function unit executing the shift operation is simple, we implement it as a synchronous cell. So the processor is composed of seven separate function units and one register files (RF) containing 128 general-purpose registers. The function units include one asynchronous multiplier (AMUL), two asynchronous adders (AADD), one Boolean comparison unit (BC), one Load/Store unit (LSU) , one shift unit (SH) and one I/O unit (I/O). The function units and register files are full connected by interconnection network consisting of 5 buses and 30 sockets. The 32-bit buses are used to transport data. In addition, the processor contains control unit (CNTRL), instruction memory and data memory.

This processor core (HTTA) is a hybrid implementation using both asynchronous function units and synchronous function units. The general organization of the proposed processor core is shown in Fig. 5.

The structural description of the processor core was obtained with the aid of the hardware subsystem of the TTA framework, which generated the Verilog description. The structures of the Boolean comparison unit, Load/Store unit, shift unit and I/O unit were described manually in Verilog. Based on the asynchronous circuit design flow that was presented in [19], the asynchronous multiplier and adder were implemented. These asynchronous function units should be defined in the TTA framework.

5 Test and Results

In order to compare the power consumption, we implemented synchronous multiplier and adder that used the same data path as their asynchronous versions. The synchronous processor core (STTA) replaced the asynchronous function units with their synchronous versions in Fig. 5. These two processor cores were implemented in $0.18\mu m$ CMOS standard cell ASIC technology. The layout was implemented in standard cell automatic place and route environment. The Mentor Graphics Calibre was used for the LPE (Layout Parasitic Extraction) and the Synopsys Nanosim was used for the performance and power analysis. In performance and power analysis, the simulation supply voltage was 1.8V, the temperature was 25°C, and the device parameters used the typical values that come from the foundry. The clock frequency of these processor cores was 200MHz.

The application MAT-MUL was executed by two processor, the optimization is purely done by the compiler. It should be noted that only performance and power

Table 2. Comparison of two processor cores

Design	Clock Frequency [MHz]	Execution Time [μs]	Power [mW]	Area [mm^2]
STTA	200	192.785	32.9503	0.782768
HTTA	200	170.555	24.7657	0.835012

dissipation of processor cores are compared, and the power dissipation of instruction and data memory are not taken into account. The obtained results are listed in Table 2.

Table 2 shows the test results of two processor cores. The metric is the execution time of the applications and power dissipation. Due to characteristics of fine-grain clock gating and zero standby power consumption, the total power of hybrid processor core is less than pure synchronous one. When switch to idle state, the asynchronous function units can save the power but synchronous function units cannot do.

The execution cycles of hybrid processor core is less than pure synchronous one. The reason is that the performance of asynchronous adder is higher than its synchronous version. The architecture of asynchronous adder is very simple and the latency of the combinatory logic in each stage is very low. According to timing analysis, The matched delays of each stage are 1.79ns, 1.82ns and 1.82ns, which is less than one clock period (4ns). After doing post-layout timing analysis and trimming of the delays, we define the matched delays are both 3ns. In non-pipelined mode, the synchronous adder needs five cycles (20ns) to complete the whole add operation. But the asynchronous adder can complete the whole operation in 16ns. So we define the latency of the synchronous adder is 5 cycles and the latency of asynchronous one is only 4 cycles in TTA framework. On the other hand, the latencies of asynchronous multiplier and its synchronous version are both 7 cycles in TTA framework. So the total performance of HTTA is higher than STTA.

Because of the area cost of the control circuits and independent power rings layout, the area of asynchronous function unit is lager than its synchronous version. The area cost is a disadvantage of the asynchronous circuits implementation without any area optimization. Designers should seek the trade-off between power consumption, performance and area. In different applications, optimization techniques for different targets should be used [20]. Taking into account of the improvement in performance and power dissipation, it is worth to pay attention to the design and application of asynchronous circuits.

6 Conclusions and Future Work

TTA is very suitable for embedded systems for its flexibility and configurability. Supported by application characteristics analysis and special function units, the architecture is easy to be modified to adapt to different embedded applications. TTA is configurable, the special function units can be easily added and used in hardware/software co-design environment in order to improve the performance or reduce the power consumption. In this paper, we modified the interface of two asynchronous function units and used them in embedded TTA processor successfully.

Here two asynchronous function units tailored for TTA architecture were used, the simulation result shows that the hybrid processor core using asynchronous and synchronous function units has higher performance and lower power dissipation than the processor core using synchronous function units. The results proves that this low-power embedded processor architecture taking advantage of asynchronous circuits and TTA is effective. Although there is some area cost, it is worth to pay attention to the design and application of asynchronous circuits in embedded systems that are sensitive to power dissipation and performance.

In this work, we found that the utilization ratio of the function units were not very high. The scheduler in TTA framework is far from perfect and the low utilization will cause the performance loss. In the future work, the software optimization will be researched further, including compiler optimization and handle scheduling.

Acknowledgments. This work has been supported by the National Natural Science Foundation of China (90407022). Thank Andrea Cilio and his colleagues in Delft Technology University of Netherlands for their great help and support to our research work.

References

1. Keutzer, K., Malik, S., Newton, A.R.: From ASIC to ASIP: The next design discontinuity. In: Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'02), pp. 84–90 (2002)
2. Werner, T., Akella, V.: Asynchronous processor survey. Computer 30(11), 67–76 (1997)
3. Furber, S.B., Garside, J.D., Temple, S., Liu, J., Day, P., Paver, N.C.: AMULET2e: An asynchronous embedded controller. In: Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems, pp. 290–299 (1997)
4. Furber, S.B., Edwards, D.A., Garside, J.D.: AMULET3: a 100 MIPS asynchronous embedded processor. In: Proceedings of the 2000 IEEE International Conference on Computer Design, pp. 329–334 (2000)
5. Garside, J.D., Bainbridge, W.J., Bardsley, A., Clark, D.M., Edwards, D.A., Furber, S.B., Lloyd, D.W., Mohammadi, S., Pepper, J.S., Temple, S., Woods, J.V., Liu, J., Petlin, O.: AMULET3i - an asynchronous System-on-Chip. In: Proceedings of the 6th International Symposium on Advanced Research in Asynchronous Circuits and Systems, pp. 162–175 (2000)
6. Plana, L.A., Riocreux, P.A., Bainbridge, W.J., Bardsley, A., Garside, J.D., Temple, S.: Spa: A synthesisable amulet core for smartcard pplications. In: Proceedings of the 8th International Symposium on Asynchronus Circuits and Systems, pp. 201–210 (2002)
7. Garnica, O., Lanchares, J., Hermida, R.: Fine-grain asynchronous circuits for low-power high performance DSP implementations. In: IEEE Workshop on Signal Processing Systems, pp. 519–528 (2000)
8. Kawokgy, M., Salama, C.A.T.: Low-power asynchronous viterbi decoder for wireless applications. In: Proceedings of the 2004 international symposium on Low power electronics and design, pp. 286–289 (2004)
9. Corporaal, H.: Microprocessor Architecture: from VLIW to TTA. John Wiley & Sons Ltd, Chichester (1998)
10. Corporaal, H., Arnold, M.: Using Transport Triggered Architectures for embedded processor design. Integrated Computer-Aided Engineering 5(1), 19–37 (1998)

11. Weste, N.H.E., Eshraghian, K.: Principles of CMOS VLSI design: a systems perspective. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA (1985)
12. Chandrakasan, A., Sheng, S., Brodersen, R.: Low-power CMOS digital design. IEEE Journal of Solid-State Circuits 27(4), 473–484 (1992)
13. Pitkanen, T., Makinen, R., Heikkinen, J., Partanen, T., Takala, J.: Low–power, high–performance tta processor for 1024–point fast fourier transform. In: Int. Workshop SAMOS, pp. 227–236 (2006)
14. Nielsen, L.S.: Low-power Asynchronous VLSI Design. PhD thesis, Technical University of Denmark, Department of Information Technology (1997)
15. Yue, H., Lai, M.C., Dai, K., Wang, Z.Y.: Design of a configurable embedded processor architecture for dsp functions. In: In: Proceedings of the 11th International Conference on Parallel and Distributed Systems - Workshops (ICPADS'05) vol. 02, pp. 27–31 (2005)
16. Sutherland, I.E.: Micropipelines. Communications of the ACM 32(6), 720–738 (1998)
17. Piguet, C., Zahnd, J.: STG-based synthesis of speed-independent CMOS cells. In: Workshop on Exploitation of STG-Based Design Technology (1998)
18. Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L., Yakovlev, A.: Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. IEICE Transactions on Information and Systems E80-D(3), 315–325 (1997)
19. Li, Y., Wang, L., Gong, R., Dai, K., Wang, Z.Y.: Research and implementation of a 32-bit asynchronous multiplier. Computer Research and Development 43(12), 2152–2157 (2006)
20. Zhou, Y., Sokolov, D., Yakovlev, A.: Cost-aware synthesis of asynchronous circuits based on partial acknowledgement. In: Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design, pp. 158–163 (2006)

A Bypass Mechanism to Enhance Branch Predictor for SMT Processors*

Yongfeng Pan[1], Xiaoya Fan[1], Liqiang He[2], and Deli Wang[1]

[1] Department of Computer Sciences, Northwestern Polytechnical University, China
{pan_yong_feng,wdl900}@mail.nwpu.edu.cn,
fanxy@nwpu.edu.cn
[2] Department of Computer Sciences, University of Cyprus, Cyprus
liqiang@cs.ucy.ac.cy

Abstract. Unlike traditional superscalar processors, Simultaneous Multithreaded processors can explore both instruction level parallelism and thread level parallelism at the same time. With a same fetch width, SMT does not fetch instructions from a single thread as deeply as in traditional superscalar processors. Meanwhile, all the instructions from different threads share the same Function Units in SMT. All the characteristics make it possible to enhance the performance of SMT by reducing the branch mis-predictions. Based on the fact that about 15% of branch instructions directions can be definitely known at predicting cycle, a simple and effective bypass mechanism is proposed. This scheme doesn't depend on any existing branch predictors, and it can be used as an effective enhancement to any one of them. Execution-driven simulation results show that the branch miss prediction rates of our predictor decrease by more than 15% on average compared to a simple base line (g-share) predictor and improve the instruction throughput by about 2.5%.

1 Introduction

Simultaneous Multithreaded processors [1, 2] increase the instruction throughput by allowing fetching and running instructions from several threads simultaneously at a single cycle. In SMT processors, functional units that would be idle due to instruction level parallelism (ILP) limitations of a single thread are dynamically filled with useful instructions from other threads. An SMT processor can hide both long latency operations and data dependencies in one thread effectively. These advantages help increase both processor utilization and instructions throughput.

With the pipeline deepening and issue widths increasing, the branch predictor plays an important role in improving the performance of an SMT processor [3]. At the same time, according to Matt Ramsay et al. [4], high accuracy of branch predictors are not always needed for a SMT processor because the SMT processors can hide the penalty effectively. So a simple and effective predictor is more suitable for SMT processors.

* Supported by National Natural Science Foundation of China (NSFC) No.60573107.

© Springer-Verlag Berlin Heidelberg 2007

It is well known that a conditional branch instruction uses the result from previous instructions to make a branch decision. In our experiments, we observe that most times the distance between instructions that produce results and a branch instruction that uses the results is not too large. In some programs, there is a high percentage of conditional branches whose direction decisions can be definitely known at the predicting cycle, and for these branch instructions the predictor should not miss-predict, and thus the wrong path instruction fetching should also be avoided for them. When this feature is considered in SMT processors, the saved fetch slots can be used to fetch more useful and correct instructions in the threads, therefore improving the overall performance.

In this paper, we are proposing a simple and effective bypass mechanism that exploits the above feature by combining a Writing-Register-Table (WRT) and a base line (g-share) branch predictor together. It is easy to implement, and needs less hardware than many existing dynamic predictors. Compared with our base line Execution-driven simulation results show that the branch prediction miss rates will be further reduced. Although we use a simple branch predictor as our base line predictor, our scheme doesn't depend on it, and can be used with any existing branch predictor.

The paper is organized as follows. First, we introduce the related work on branch predictors. Then, we present a study on the characteristics of branch instructions and their relied registers, and give our proposed bypass mechanism. In the third part, we give the simulation results and analysis, and finally conclusions.

2 Related Work

Till now, a lot of branch predictors have been proposed. Yeh and Patt [5] show that a two-level branch predictor can achieve high levels of branch prediction accuracy. And S. McFarling [6] proposed the g-share branch predictor. To solve the problem of branch interference, Chih-Chieh Lee et al [7], introduced the Bi-modal Predictor, Eric Sprangle et al. introduced the Agree Predictor [8]. Additionally, Skewed Branch Predictor [9], the Filter Mechanism [10] and the YAGS predictor [11] are introduced in traditional superscalar processors. Moreover, value predictors using the concept of value locality to improve branch predictor accuracy [12, 13, 14, 15, 16] are always used as a component of a modern combined predictor.

The past few years, some new methods are introduced such as Lucian N. Vintan's pre-computed branches [17] which compute the destination of conditional branches as early as the first operand is ready for superscalar processors, Robert S. Chappell's Difficult-path branch prediction that uses subordinate micro-threads [18], Craig Zilles's Execution-based prediction that uses speculative slices [19], Lucian Vintan introduced the Neural Branch Prediction [20], Renju Thomas et al studied dynamic dataflow-based identification of correlated branches from a large global history [21], Steven Swanson et al. evaluated the importance of branches in modern deep pipelined processors [22], and David Tarjan introduced the hashed perceptron predictor, which merges the concepts behind the g-share, path-based and perceptron branch predictors [23].

Though these methods are suitable for superscalar processors, they do not take advantage of SMT processors characteristics. In this paper, we propose a simple and effective bypass scheme for SMT processors that tries to decrease the wrong path instruction fetching when a branch can be predicted correctly for sure. Our scheme outperforms traditional simple predictors and can be used with any other traditional predictors. Although it is based on previous work [25], there is a big difference between them. Our scheme does not predict all conditional branches; instead, those branches whose operands are not being written by in flight instructions will not go through the branch predictor. In the next section we will describe this in detail.

3 A Bypass Scheme to Improve Branch Predictor

In this section we present our basic idea, and give the proposed bypass scheme in detail.

3.1 Basic Idea

Traditionally, high performance processors employ complex predictors and fetch deeply to find more independent instructions and therefore increasing ILP. However, in SMT processors, as there are enough instructions that can be fetched from different threads, it is not necessary for SMT to fetch as deeply as traditional processors. Besides, SMT processors can hide mis-prediction penalties effectively, therefore making a simple predictor more suitable for them.

The outcome of a conditional branch requires the datum values from two previously computed source operands. As soon as these operands are known, then by using data-value prediction techniques the branch outcome can be speculated. Previous studies have shown that the greater the distance branch source operand computation is from the branch instruction then the greater the degree of data-value prediction. In our simulations, using an SMT processor, we show that a significant proportion of the computation of source operands (about 15%) required by conditional branches are far enough from their associated branch instruction to render data-value prediction appropriate. Processor performance and thread utilization is, therefore, improved by the resulting earlier pre-fetching of instructions from the appropriate conditional branch path.

3.2 Architecture of Our Scheme

The architecture of our scheme is shown in Figure 1. It includes three components: a Writing Register Table (WRT), a base line predictor, and an update engine.

The WRT. Using Alpha ISA as an example, and consists of 64 entries that represent 64 physical registers. Each entry includes two fields: the first one is a 9 bits counter to record the number of in flight instructions that will write the

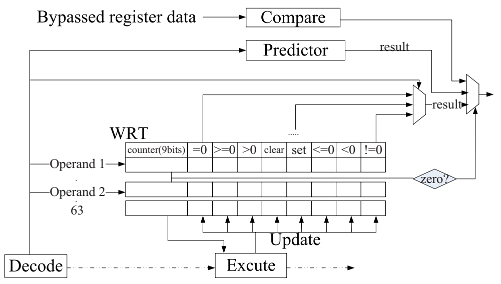


Fig. 1. The scheme of our predictor

register (here we set the maximum number of in flight instructions to 512), "0" means there is no instruction in flight will write the register. Whenever there is an instruction will write register, the corresponding counter is increased by one, and when the instruction finishes the counter is decreases by one. The second field is a flag field which indicates the flag of the register, that is branch on equal to, not equal to, greater than, greater than or equal to, less than, less than or not equal to zero, and the low bits of the register.

The Base Line Predictor. The base line predictor component can be any predictor. Here we use a simple g-share/bi-modal predictor as an example. As an enhancement of branch predictors, our scheme must be used together with an existing branch predictor. When the fetch engine meets a branch instruction whose source register is not clean (the dirty bit in WRT is set), it looks up the base line predictor to get a branch prediction.

The Update Engine. The update engine is a bypass-logic, when one instruction is executed and its destination is a register, the result of this instruction will update the flag field and the counter of WRT.

3.3 How It Works?

To enhance branch predictor using our scheme, we need the detailed information (branch or non-branch instruction, source register numbers if it is branch) of instructions. Generally, this information can be known at the Decode stage. In the Alpha ev6 processor, a line predictor is used to help to determine the next fetch block at the fetch stage, and is updated by a branch predictor at the Decode stage if they disagree. If a SMT processor has similar architecture as the Alpha processor, we can get the instruction's information without any problems. But if the branch predictor is accessed in the fetch stage, our scheme needs some pre-decoding information from another structure; here we propose using pre-decoded Icache to provide the needed information.

With the provided instruction details, when the SMT processor meets a conditional branch instruction, it will look up the entry corresponding to the destination-register in the WRT, and check whether the "counter" is "0". If it is “0”, it means that this register has not been updated by the in flight instructions. In this case, in the next cycle, the Instruction Fetch (IF) stage will fetch instructions according to the value (target address) stored in the register. Otherwise the destination of the next instruction will be decided by the predictor (which can be any traditional predictor.) In this paper, we use g-share/bi-modal predictor as the traditional predictor.

If an instruction in flight needs to write a new value in a register, the “counter” in the WRT entry corresponding to the register will increase, and when the instruction is finished in the pipeline then the “counter” decreases.

In super-scalar processors, the fetch depth can be as far as 20 to be able to find enough instruction to issue, and in such situations, there is no need to use our scheme because few conditional instructions whose distance from its relied registers is bigger than 20.

However, in SMT processors, the fetch depth in a particular running thread is not so deep because of the parallel running mechanism in them, so there is more chance that conditional branches can get their operands and make a decision than in superscalar processors. These chances give us enough space to improve the branch prediction accuracy and thus improve the overall performance. In the next section, we will give the experimental framework and the simulation results.

3.4 The Cost of Our Scheme

We assume that every thread has its own such bypass structure. Each structure needs 64*18bits and the extra bypass logic. In some architecture, the clean bit has been implemented in many modern superscalar processors. Therefore, the overall cost of such bypass scheme is quite small.

4 Experiment Framework and Simulation Results

4.1 Experiment Framework

We modify the sim-safe tool of the simplescalar3.0 [24] simulator to calculate the distance between the branch instructions and the relied instructions. Firstly, we measure the length of both the double-operand-branch ISA (PISA) and the single-operand-branch ISA (Alpha). Then, based on the fact that double-operand-branch ISA and single-operand-branch have similar results, we do our further experiment on SMTSIM [2] to test our scheme in SMT processor.

The configuration of SMTSIM is given in Table 1. In our experiment, to compare with our predict scheme we modify the simulator to let every thread has its own predictor. We implemented our scheme in the simulator, and combined the g-share predictor and the WRT together.

We select some of SPEC2000 benchmarks (8 integers and 5 floats) to test our schemes. To get the different data of the same benchmarks, we test the same benchmark in different threads environment. As an example, we test the data of “gzip” in 1, 2, 4, 6, 8 threads. Consequently, we know that the different clean rate

Table 1. Configuration of SMT processor

Parameter	*Value*
Functional Unites	3 FP, 6 integer (including branch), 4 load/store, 2 synchronization
Pipeline	8 stages
Branch miss penalty	6 cycles
Instruction Queue	32-entry FP, 32 entry Int
Inst./Data Cache	64KB/64KB, 2-way, 64 byte
L2/L3 Cache	512KB/4MB, 2-way, 64 byte
I/D TLB, miss penalty	48/128 entry, 160 cycles
Latency (to CPU)	L2 6, L3 18, Mem 80 cycles
Fetch Police	ICOUNT.2.8 [2]
Fetch/Rename/Issue/Commit Width	8 instructions/cycle

Table 2. Information of Individual benchmarks

No.	*Name*	*Num.ofInst.*	*Num.ofBranch*
1	Mgrid	281 million	0.05 million
2	Crafty	498 million	42 million
3	Equake	640 million	88 million
4	Gcc	263 million	31 million
5	Gzip	614 million	36 million
6	Mesa	500 million	49 million
7	Art	213 million	29 million
8	Ammp	477 million	11 million
9	Mcf	543 million	78 million
10	Vortex	337 million	38 million
11	Bzip2	477 million	51 million
12	Twolf	445 million	59 million
13	Parser	200 million	36 million

of condition-branch in the same benchmark. From the thirteen benchmarks, we created eight two-thread, four four-thread, two six-thread and one eight-thread work-loads randomly. The combination of these benchmarks with their running instructions are listed in the Table 2 and 3. We use the reference inputs for these benchmarks and and fast forward 10 billion instructions before starting detailed simulation.

4.2 Simulation Results

In our experiments, we first measured the distance which is the basic of our bypass scheme, when the branch instruction has one operand (Alpha), we record the distance between the instruction produce this operand and the branch that uses this operand. When the branch instruction has two operands (PISA), we

Table 3. The combinations of different benchmarks

Num.ofThread	*Combinations*	
2	1+5, 1+2, 3+4, 3+5,	6+9, 8+11 7+10, 12+13
4	1+2+3+4, 6+7+9+10,	2+8+5+11 9+11+12+13
6	1+2+3+4+5+6,	7+8+9+10+11+12
8	5+6+7+9+10+11+12+13	

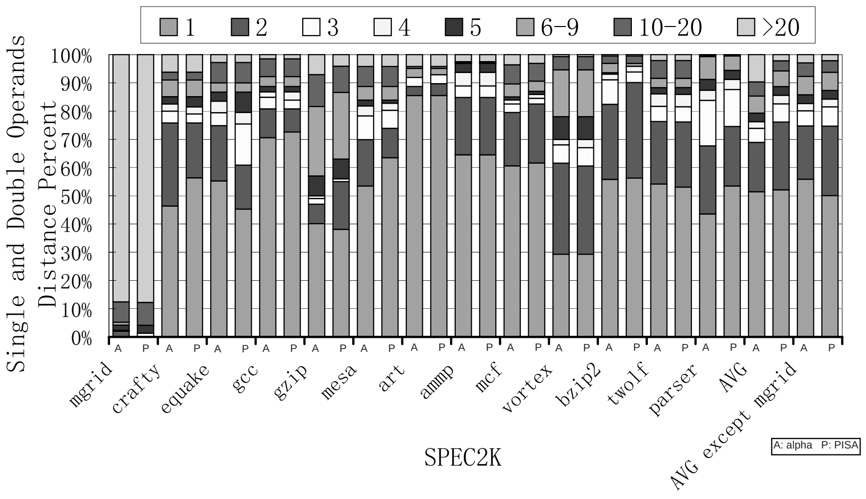


Fig. 2. The distance of Alpha and PISA

record the distance between the instruction that produces the last operand and the branch. The results are illustrated in Figure 2. In this figure, the 30 columns are divided into 15 groups. Each group stand for one benchmark and has two column: the left one and the right one. The left one means Alpha ISA, and the right one means PISA ISA.

Both of the left and right columns have similar results. That is because (1) both Alpha and PISA ISA are RISC, and therefore their branch destinations are determined by one or two registers, and (2) the distance has a close relationship with the characteristics of programs instead of the instruction sets. Therefore, to make things simple, in the rest of this paper, we focus on the study of Alpha ISA and conduct our other experiments on SMTSIM simulator.

In the Figure 2 we can find that the distance of about 15% of branch instructions and the instruction writing the register which will be used by this branch is more than 6. To illustrate this point more clearly, let us look a fragment of a program:

A. LDL R1, [R2+100]
B. MUL R1, R1, R3
C. MOV R3, R2
D. BEQ R1, Label

Instruction B writes register R1, and branch instruction D use the sign of R1 to determine whether it will go to Label. The distance mentioned above stands for the number of instructions between B and D. In this example, the distance is 2.

In a SMT processor, all threads use the same FU; there is a high probability that some instructions from other threads fit in the slots between instruction B and D, so the distance of B and D will be more than 2 in this example.

Therefore, in a fetch width is 8 instruction per cycle and 4 threads environments, if other threads can provide one independent instruction to insert between the branch and its relied instruction, then when the distance is more than 5 the destination of this branch can be sure before it is predicted. Similarly, when there are 8 running threads, when the distance is more than 2 then this branch is sure. To get the information precisely, we modify the SMTSIM simulator to record the clean rate in different thread work-loads. When a branch instruction is in the decode stage of the pipeline, and if the clean bit in WRT is 0, then this branch instruction is called a clean branch. The clean rate is the percentage of clean branch instructions to the total branches. We use ICOUNT 2.8 [2] fetch policy and the results are illustrated in Figure 3.

As the number of threads is increasing, the percentage of clean branches gets higher and higher. That is because in the SMT processor, some instructions from different threads that do not affect the operands of the current branch are inserted into the issue queue. So the distance between the branch and its relied instructions is enlarged. There are also more chances that the operands are ready when the branch needs to use them. Figure 5 shows that the percentage is quite optimistic.

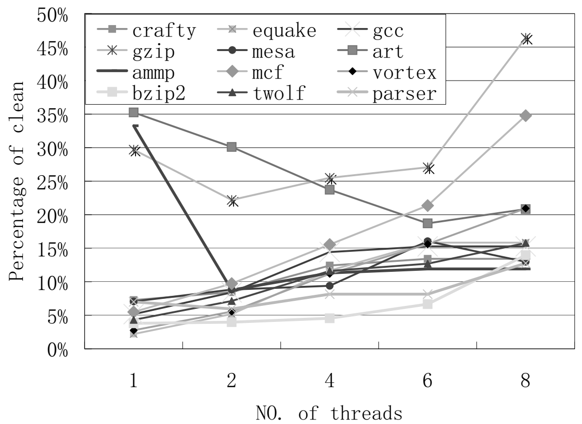


Fig. 3. The relation between clean rate and number of threads

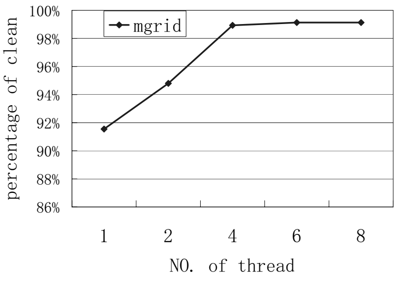


Fig. 4. The relation between clean rate and number of threads of mgrid

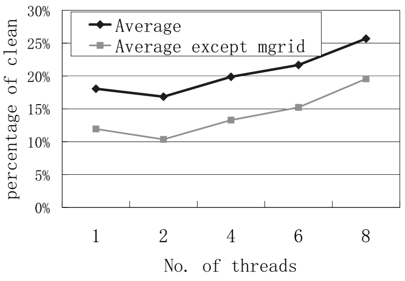


Fig. 5. The relation between clean rate and number of threads on average

There is an exception program, mgrid. Because there are not enough branches in this program, we get a very positive result (Figure 4). However, to be more accurate, we eliminate this program and the average result is illustrated in Figure 5.

Figure 5 shows that when there are 2 running threads, the clean rate is lowest, that is because the clean rates of some float benchmarks decline significantly. And when the number of running threads is more than 2, the clean rates get higher and higher as thread number is increasing. Particularly, when there are eight threads, the clean rate is 19.57%. Considerably, if we eliminate such branch instruction from mis-prediction, then the rate can be higher. For instance, when we use a predictor which is 90% correct, by using this scheme in Figure 1, we can increase the prediction to 91.96% (19.57%*(1-90%)+90%).

4.3 Branch Prediction Miss Rate

We implemented our scheme with g-share and bi-modal and called them g-share WRT and bi-modal WRT respectively. We compare the mis-prediction rate among 4K g-share, 4K g-share with WRT, and 4.5K bi-modal, and the results are as illustrated in Figure 6.

As mentioned before, WRT is not a predictor. It can be used as a complementary of any predictor to improve the prediction accuracy. In Figure 6, we know

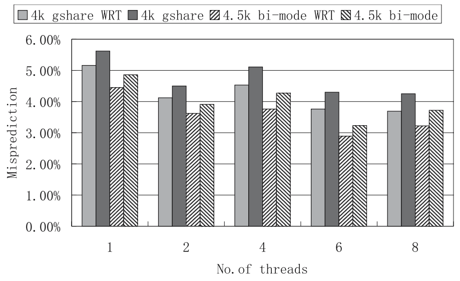


Fig. 6. Branch prediction miss rate

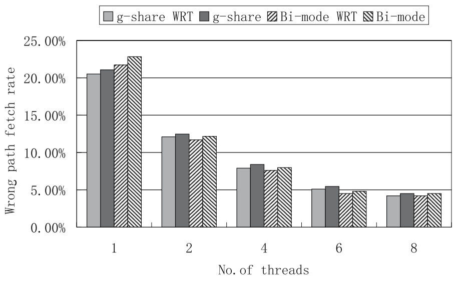


Fig. 7. Wrong Path instruction fetch rates

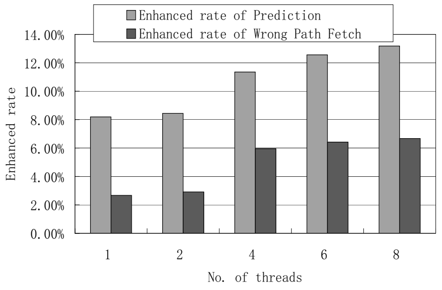


Fig. 8. Enhanced rate of prediction and wrong path fetch

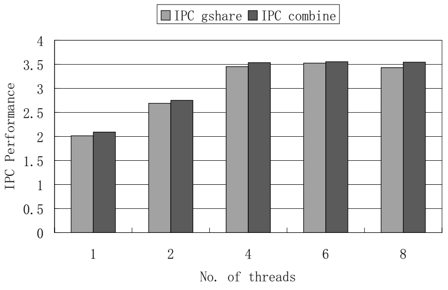


Fig. 9. Instruction throughput of our scheme and g-share predictor

that our scheme can improve the original predictor effectively no matter what the predictor is.

In Figure 8, we show the relationship between the number of threads and the enhancement of prediction. It is clear that the more running threads, the more benefit we can get from our bypass scheme. Furthermore, when the threads number increases from 2 to 4, the enhancement is higher than others. This illustrates that when there are only 2 running threads, the independent instructions are insufficient to insert between the conditional branch and the relied instruction. When there are 4 or 6 threads, the independent instructions from other threads are enough to show the efficiency of our scheme.

4.4 Wrong Path Instruction Fetch Rate

SMT processor can reduce the wrong path fetch rate effectively; in addition, our scheme can enhance this rate by 6% in 4 threads and 6.7% in 8 threads situations(Figure 7).

Figure 8 shows that in the 4 threads situation, the improvement of reducing wrong path fetch rate is most obvious. And the further increase of thread number has little influence on the enhancement. This indicates to us that we can obtain higher benefit by increasing from 2 threads to 4.

4.5 Processor Performance

Although the SMT can tolerate the degradation of performance caused by misprediction, our scheme still improves the overall performance by 2.55% on average. Figure 9 shows the instruction throughput of our scheme compared with the original g-share predictor.

5 Conclusion

In this paper, we studied the distance of conditional branches and its relied instructions, and present a bypass mechanism which may be not suitable in superscalar processors. With the help of WRT, the accuracy of the original prediction can be improved by 15% percent on average. The implementation of our predictor is simple, and the hardware cost is little. Execution-driven simulation results show that our predictor can be more effective as the number of threads increases.

Acknowledgement

We thank Panayiotis Charalambous in University of Cyprus for his kindly help of our English writing.

References

1. Tullsen, D.M., et al.: Simultaneous Multithreading:Maximizing On-Chip Parallelism. In: Proc. 22nd ISCA (June 1995)
2. Tullsen, D.M., et al.: Exploiting Choice:Instruction Fetch and Issure on an Implementable Simultaneous Multithreading Processor. In: Proc. 23rd ISCA (June 1996)
3. Swanson, S., et al.: An evaluation of speculative instruction execution on simultaneous multithreaded processors. ACM Transactions on Computer Systems archive 21(3) (2003)
4. Ramsay, M., et al.: Exploring Efficient SMT Branch Predictor DesignUniversity of Wisconsin-Madsin, Department of Electrical and Computer Engineering (2003)
5. Yeh, T.-Y., et al.: Two Level Adaptive Training Branch Prediction. In: 24th ACM/IEEE International Symposium on Micro architecture. pp. 51–61 (1991)
6. McFarling, S.: Combining Branch Predictors. Technical Report TN-36, Digital Western Research Laboratory (June 1993)
7. Lee, C.-C., Chen, I-C.K., Mudge, T.N.: The Bi-Mode Branch Predictor. In: Proc. MICRO30 (December 1997)
8. Sprangle, E., et al.: The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference. In: Proc. 24th ISCA (May 1997)
9. Michaud, P., et al.: Trading Conflict and Capacity Aliasing in Conditional Branch Predictors. In: Proc. 24th ISCA (May 1997)
10. Chang, P.Y., et al.: Improving Branch Prediction Accuracy by Reducing Pattern History Table Interference. In: Proc. ICPACT (October 1996)
11. Eden, A.N., et al.: The YAGS Branch Prediction Scheme. In: Proc. 31st MICRO (December 1998)
12. Heil, T.H., et al.: Improving Branch Predictors by Correlating on Data Values. In: Proc. 25th ISCA (June 1998)
13. Tuck, N., et al.: Multithreaded Value Prediction. In: Proc. of the 11th ISHPC(February 2005)
14. Calder, B., et al.: Selective Value Prediction. In: Proc. of the 26th ISCA (May 1999)
15. Lipasti, M.H., et al.: Value locality and data speculation. In: Proc. 7th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1996, pp. 138–147 (1996)
16. Sazeides, Y., et al.: The predictability of data values. In: Proc. 30th MICRO (1997)

17. Vintan, L., et al.: An alternative to branch prediction: pre-computed branches. ACM SIGARCH Computer Architecture News archive 31, 20–29 (2003)
18. Chappell, R., et al.: Difficult-path branch prediction using subordinate micro-threads. In: Proc. 29th ISCA (2003)
19. Zilles, C., et al.: Execution-based prediction using speculative slices. In: Proc. 28th ISCA (2001)
20. Vintan, L., et al.: Towards a High Performance Neural Branch Predictor. In: Proc. IJCNN '99, Washington DC, USA (1999)
21. Thomas, R., et al.: Improving branch prediction by dynamic dataflow-based identification of correlated branches from a large global history. In: Proc. ICCA (2003)
22. Swanson, S., et al.: An evaluation of speculative instruction execution on simultaneous multithreaded processors. ACM Transactions on Computer Systems 21 (2003)
23. Tarjan, D., et al.: Merging path and g-share indexing in perceptron branch prediction. ACM Transactions on Computer Systems, 2 (September 2005)
24. Austin, T., et al.: SimpleScalar: An infrastructure for computer system modeling. IEEE Computer 200235(2), 59–67
25. He, L., et al.: A new value based branch predictor for SMT processors. In: Proc. 16th IASTED International Conference on Parallel and Distributed Computing and Systems (2004)

Thread Priority-Aware Random Replacement in TLBs for a High-Performance Real-Time SMT Processor

Emre Özer and Stuart Biles

ARM Ltd.
110 Fulbourn Rd., CB1 9NJ Cambridge, UK
{emre.ozer,stuart.biles}@arm.com

Abstract. This paper proposes a novel random replacement method in fully or set associative structures such as TLBs to improve the performance of the main or high-priority thread running in an SMT processor along with other low-priority threads. The proposed random replacement technique considers the thread priorities when performing a random selection of evicted entries in the table. The replacement scheme increases the probability of evicting a low-priority thread entry by generating more than one random number index. We have shown that this simple and low-cost random replacement logic can boost the performance of the high-priority thread significantly with only minimal additional hardware support. Our results indicate that generating only 3 random numbers can increase the performance of the high-priority thread by about 9%, and provides the highest overall IPC for an 8-entry data TLB.

1 Introduction

Microarchitects prefer to use less costly schemes such as *round-robin, not-last-used (NLU)* or *random* replacement algorithms for fully or set associative tables/caches, particularly in embedded microprocessors because the replacement policies that provide higher performance such as least recently used (LRU) are hardware-intensive.

In an SMT processor, the set and fully associative structures such as caches, branch target buffers (BTBs) and TLBs are shared by many simultaneous threads. An entry belonging to one thread can be evicted by another thread. For instance, the least recently used entry is replaced no matter which thread the entry belongs to when the replacement policy is LRU. Similarly, the entry whose index is generated by a random number generator is chosen for eviction if the replacement policy is random.

The replacement policy without differentiating thread priority may not be problem for an SMT processor in which all threads have equal priority. However, the situation becomes different if the SMT processor has one highest priority and other low priority threads. This may be a typical case in a soft/hard real-time system where the highest priority thread or the real-time thread is given all resources and other threads are opportunistic in the sense that they use resources only when the highest priority thread stalls for a reason. Ideally, the real-time thread is expected to be delayed minimally by the non-real time low-priority threads. When a TLB is shared among all threads in a real-time SMT processor, a low priority thread replacing a TLB entry belonging to the highest priority thread can be detrimental to the performance of the highest priority thread.

© Springer-Verlag Berlin Heidelberg 2007

The current state-of-art random replacement algorithm, whether it is used in set or fully-associative caches or TLBs, selects an entry to be replaced randomly. The probability of an entry being replaced is equal for all entries, i.e. $1/N$ for N-entry table. However, this kind of random selection may not be appropriate in a real-time SMT processor in which one thread has priority over the others. Thus, the probability should be higher than $1/N$ in order to increase the chance of finding a low priority thread entry rather than a high priority thread one.

In this paper, we assume that we have a *real-time* SMT processor that has one *highest priority thread* which is the real-time task, and all other threads are all low priority non-real-time tasks. All threads in the SMT processor share fully-associative TLBs that use random replacement policy to evict an entry. We propose an improved random replacement policy for associative tables such as caches, TLBs, BTBs and etc. However, our focus will be on *thread priority-aware random replacement policy* for fully-associative TLBs. Using this novel random replacement policy, the probability of finding a low priority thread entry can be made as high as m/N where m is a design parameter.

The structure of the paper is as follows: *Section 2* discusses the related work. *Section 3* explains the rationale behind the thread priority-aware random replacement policy. *Section 4* compares the performance of the thread priority-aware random replacement policy to the traditional random replacement policy for fully-associative data TLB and attempts to reach a cost-effective design of this new policy. Finally, *Section 5* summarizes the paper with a discussion of the results and other potential applications of our technique.

2 Related Work

There is a plethora of studies on SMT where the focus is mainly on improving the overall throughput of the processor core [1, 2 and 3]. In these studies, the threads have equal priority to share resources and therefore much of the attention has been paid to improve the overall processor throughput.

Dorai et al. [5] investigates resource allocation policies to keep the performance of the high-priority thread as high as possible while performing low-priority task along with the high-priority thread. High-priority and low-priority thread model is also explored in *Raasch et al.* [4] in the context of prioritizing the fetch bandwidth among threads. Similarly, *Cazorla et al.* [6] discusses a technique to improve the performance of high-priority thread in an SMT processor under the OS control. None of these prior art study the performance impact of the replacement policies on shared associative tables such as TLBs.

3 Thread Priority-Aware Random Replacement

Fig. 1 shows a typical random replacement implementation using linear feedback shift register (LFSR). Although our scheme can work for any kind of pseudo-random generator, we use LFSR to illustrate how it works. At every cycle, the LFSR is shifted left by an input derived by XORing some of the bits in the register. The replaced entry

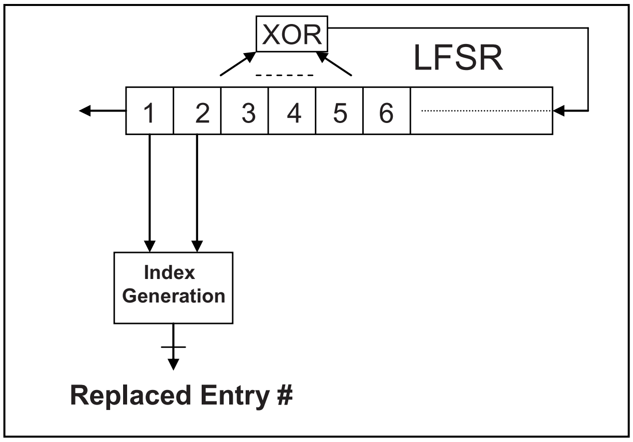


Fig. 1. Random replacement using a LFSR

number can simply be derived by reading out some MSB bits from the register. For instance, 4 MSB bits are read into the Index Generation logic to generate the replaced entry number if the fully associative table has 16 entries.

This sort of random replacement can be comfortably used in an SMT processor in which all threads have equal priority. On the other hand, it does not run efficiently when it is used in a real-time SMT processor having one high priority thread and all other threads having low priority because each entry in the table has equal probability of being replaced. What we really want is to devise a method of increasing the chance of picking an entry belonging to a low priority thread rather than the high priority one.

Generating only one random entry may give accidental replacement of a high priority thread entry from the fully associative table. However, it may be more likely to find a low priority thread for replacement if more than one random entry number is generated. In order to support such a mechanism, each entry in the table should be identified by its thread information. Fig. 2 shows the *Priority Bit Vector* (PBV) register in conjunction with the fully associative table. For each entry in the table, there is a priority bit in PBV. Initially, the PBV register has all zeros. When an entry is being written by a high priority thread, the associated bit in PBV register is also set. However, if the entry is being written by any of the low priority threads, the PBV bit is cleared.

The implementation of thread priority-driven random replacement (PRR) for an N-entry fully-associative table is shown in Fig. 3. The operation of the LFSR is the same as the one above but the number of entries read out of it is more than one. $log_2(N)$ bits need to be read from LFSR to form an index to N-entry table. m different log_2(N)-bit portions are read to increase the probability of finding a low-priority thread entry. The more such portions are read, the higher the chance of finding a low-priority thread entry becomes. Here, m is an implementation-dependent parameter. In order to generate

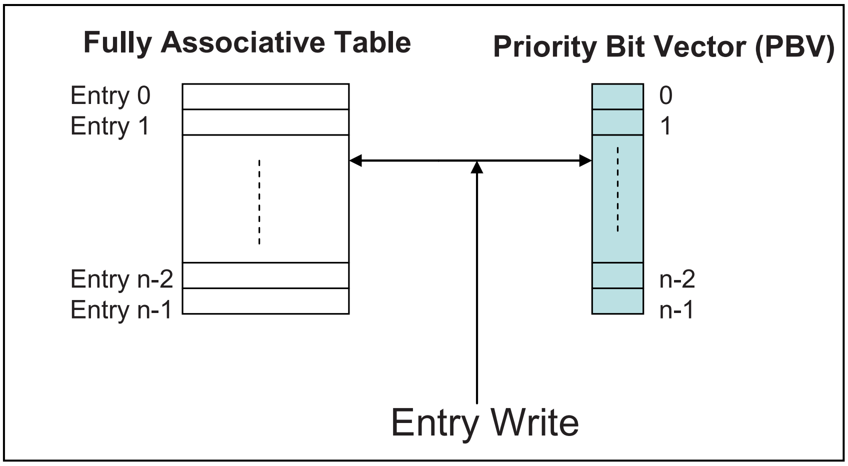


Fig. 2. Priority Bit Vector (PBV)

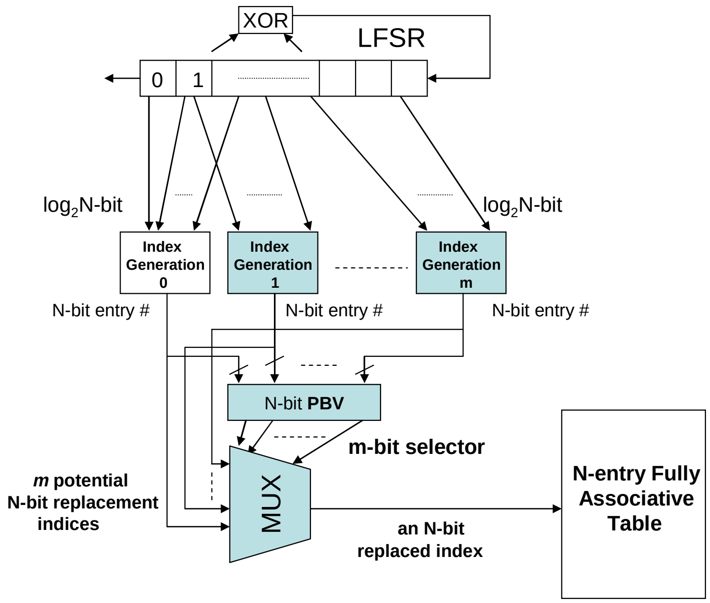


Fig. 3. Priority-driven random replacement (PRR)

m random numbers, we need *m* **index generation logic** (i.e. $\log_2$(N)-to-N decoder in this particular case). We also need an N-bit **PBV register** and a **multiplexer** to select the first low priority entry out of *m* entries.

m different indices are used to access the PBV register to read *m* 1-bit thread priority information associated with the randomly selected entry. Then, these *m* 1-bit information is fed into the multiplexer that selects the first low-priority entry number from left to right. The selected entry is the entry to be replaced from the table. If there is no low priority entry out of *m* randomly selected entries, then the first high priority entry from left will be the entry to be replaced.

Even though the table may have some entries belonging to a low priority thread, this scheme may still select an entry that belongs to a high priority thread. For instance, all *m* indices can point to entries that belong to a high priority thread. However, it can be very likely to find an entry belonging to a low priority thread for replacement if *m* is large enough.

4 Experimental Results

We have implemented this idea for a fully associative micro data TLB (DTLB) table in an **ARM** processor designed as a 2-thread SMT core in which one thread has the highest priority over the other one. Both threads share the same DTLB. The baseline or original TLB uses traditional random replacement policy. For PRR, we vary the number of randomly generated indices from 2 to 8.

Table 1. ARM-SMT processor parameters used in the simulation

Parameters	**Details**
Processor type	In-order superscalar
Issue width	Dual-issue
Fetch bandwidth	2 32-bit or 4 16-bit instruction fetch per cycle
Decode bandwidth	2 instructions per cycle
# of Threads	2
Virtually-indexed Physically-tagged On-chip L1 Instruction Cache	4-way 32KB with 1-cycle hit time
Virtually-indexed Physically-tagged On-chip L1 Data Cache	4-way 32KB with 1-cycle hit time
Physical On-chip L2 Unified Cache	8-way 256KB with 8-cycle hit time
Memory Access Latency	60 cycles
Data TLB size	**8-entry fully-associative with random replacement**
Instruction TLB size	32-entry fully-associative random replacement
Branch Predictor	4096-entry Global Branch Predictor

We have performed a cycle-accurate simulation of an SMT implementation of an ARMv7 architecture-compliant processor using the *EEMBC* [7] benchmark suite. The processor is modeled as a 2-thread soft real-time SMT core in which one thread is the dedicated HP thread, and the other one becoming the LP thread. The HP thread has priority using the fetch, decode and issue resources over the LP thread in the SMT model. We have used 18 benchmarks from the *EEMBC* benchmark suite covering a

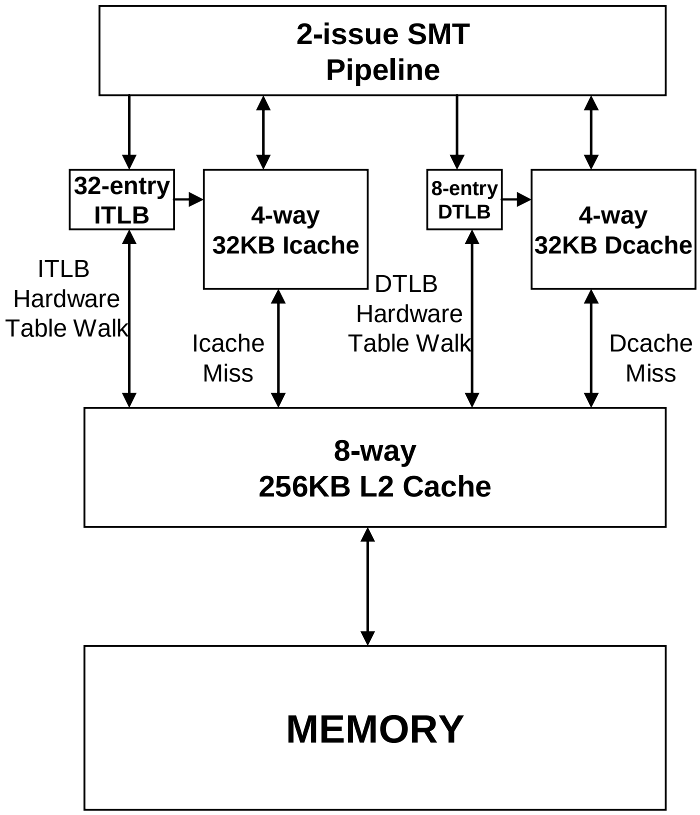


Fig. 4. The simulated ARM-SMT processor-memory model

wide range of embedded applications including consumer, automotive, telecommunications and DSP. We run all possible dual-thread permutations of these benchmarks. A dual-thread simulation run completes when the HP thread finishes its execution, and then we collect the required statistics. The ARM-SMT processor core and memory model parameters are shown in Table 1. The processor-memory model of the simulated ARM-SMT model is shown in Fig. 4. The instruction and data caches are virtually-indexed physically-tagged caches. While the set portion of the virtual address is indexing the instruction or data cache, the virtual tag portion is translated into the physical tag by the instruction or data TLB at the same time. Thus, the performance of the instruction and data TLB is critical to sustain a high HP thread performance.

We have measured 3 metrics to compare the performance of the PRR to the original random replacement: 1) Speedup of the highest priority thread, 2) CPI of the lowest priority thread, and finally 3) the total IPC. As the highest priority (HP) thread has the priority to use all processor resources over the other thread, any optimization on a shared processor resource will shorten its execution time because there will be some improvement in the DTLB hit rate of the HP thread. Thus, measuring the speedup relative to the baseline is a sensible metric. By nature, the PRR scheme will reduce the DTLB hit rate of the low priority (LP) thread as the scheme is inclined to evict an LP entry from the DTLB. Therefore, we need to measure how this decline in the DTLB hit rate in LP thread affects its overall performance. Thus, we measure the CPI of the LP thread under SMT for both baseline and the PRR scheme. Finally, the total IPC of the processor allows us to measure the total throughput of the processor with respect to the varying number of randomly-generated indices.

All measurements are done for an 8-entry fully-associative DTLB. We choose 8-entry data TLB to stress it out aggressively as the *EEMBC* benchmarks are, in general, kernel programs that do not put pressure on relatively large data TLBs (e.g. 32-entry).

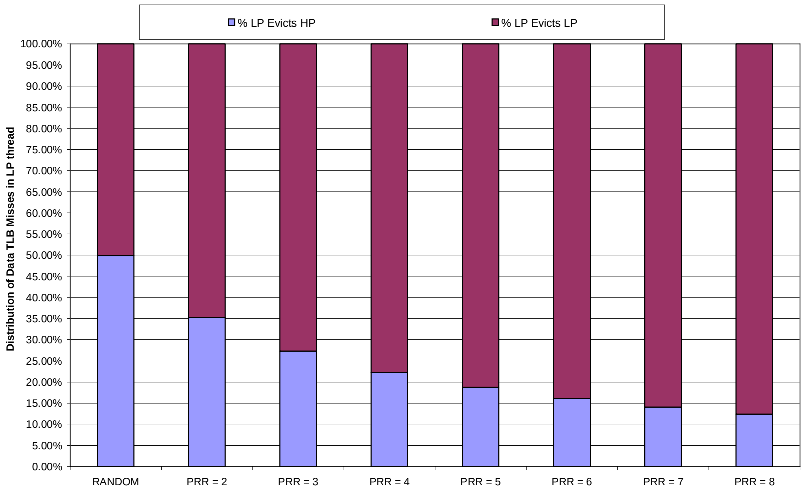


Fig. 5. Distribution of DTLB evictions by LP thread

We plot the distribution of DTLB misses caused by LP thread in Fig. 5. The first column in the x-axis is the baseline random replacement policy, and the randomly-generated indices vary from 2 to 8 for the PRR scheme. Each column consists of two portions. The bottom portion represents the percentage of DTLB evictions performed by LP thread in which the HP entry is replaced by an LP entry. The top portion represents the percentage of DTLB self-evictions by LP thread. In baseline or traditional random replacement policy, almost half of the DTLB misses caused by LP thread evicting HP thread entries. When the PRR scheme is used, the distribution is skewed in favour of the top portion, which is the percentage of self-LP evicts as the number of random indices increases. This means that the percentage of LP thread evicting HP TLB entries is decreasing when more random numbers are drawn from the LFSR to select a victim. The percentages of LP evicting HP TLB entries are **50%, 35%, 27%, 22%, 19%, 16%, 14% and 12%** for **1**, **2, 3, 4, 5, 6, 7 and 8 indices**, respectively.

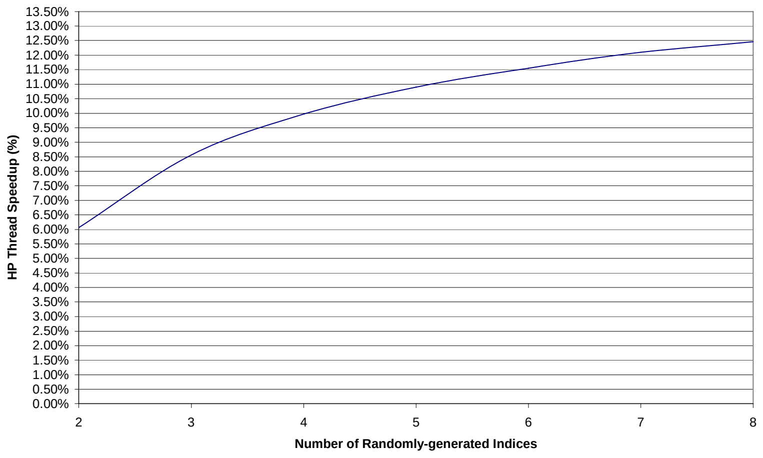


Fig. 6. HP thread speedup

Fig. 6 shows the speedup of the HP thread in a processor using the PRR scheme with varying number of randomly-generated indices relative to the same processor using the baseline random replacement scheme.

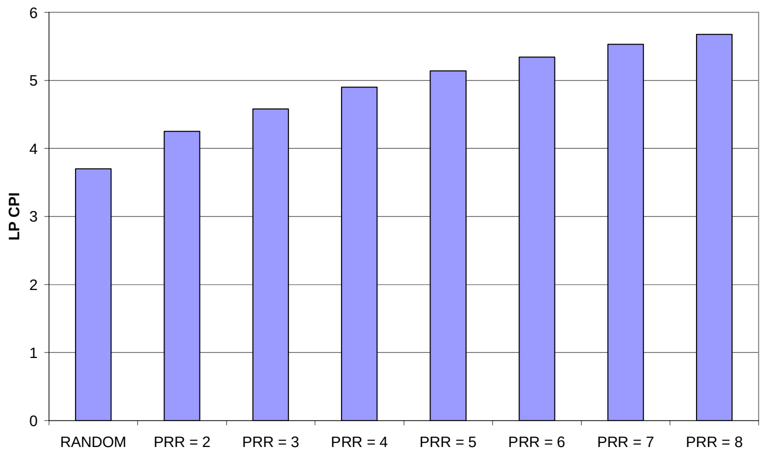


Fig. 7. CPI of the LP thread

The slope of the curve ramps up to 3 indices and gradually declines thereafter. This means that generating 3 or more random indices still improves the performance of the HP thread but the rate of improvement slows down. The reason for this drop in the rate of improvement is that the probabilities of finding a low-priority victim entry become very close after 3 or more random numbers. The actual speedup values are

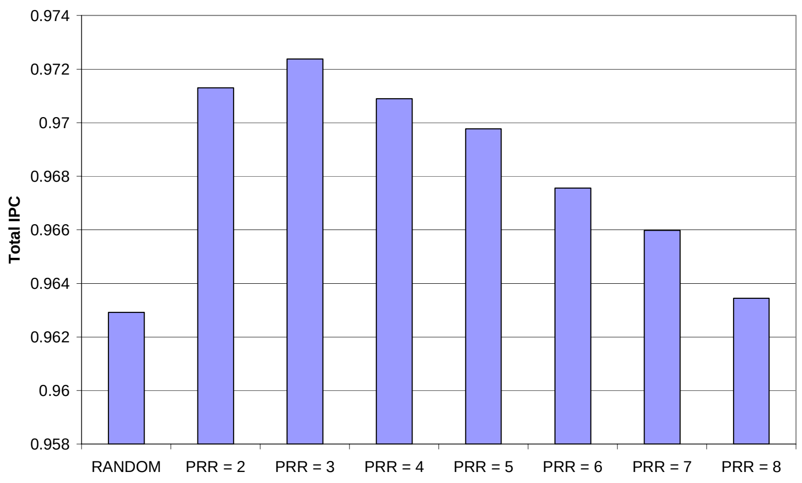


Fig. 8. Total IPC of the 2-thread SMT processor

6.1%, 8.7%, 10%, 10.9%, 11.6%, 12.1% and 12.5% for **2, 3, 4, 5, 6, 7 and 8 indices**, respectively.

Fig. 7 shows the CPI of the LP thread for the original random replacement as well as the varying sizes of the PRR scheme. The CPI of the LP thread for the baseline random replacement is about 4, which means that it can commit an instruction at every 4 cycles. In contrast, the LP thread in the PRR scheme commits an instruction at every 5 cycles up to 4 indices. This is only 1 cycle worse than the original random scheme due to decline in the DTLB hit rate of the LP thread. Thereafter, the CPI is increased by one more cycle and becomes 6 for 5 or more random indices.

Finally, Fig. 8 shows the total IPC or throughput of the SMT processor core for all schemes. Although the changes in IPC numbers are quite small, it is important to explain the underlying behaviour of this graph rather than providing IPC quantities. The baseline random replacement scheme has the lowest IPC. As the randomly-generated indices in PRR increase, so does the total IPC up to 3 indices. After 3 indices, the total IPC starts declining. With reference to Fig. 6, the slope of the HP thread speedup drops after 3 entries (i.e. the execution time reduction rate slows down), and also the CPI of the LP thread increases steadily as the number of randomly-generated indices increases (i.e. fewer number of LP instructions can be committed per cycle). The aggregate effect of these two factors is the decline of the total IPC as seen in Fig. 8 after 3 indices.

5 Discussion and Conclusion

We have shown that the optimal number of randomly-generated indices lies between 2 and 4 when considering the high-priority thread speedup, the total instruction throughput and the hardware cost of the replacement scheme. For instance, it accelerates the high-priority thread by about 9% and improves the total instruction throughput by 1% at a cost of 2 extra decoders, the priority vector register and a multiplexer for 3 randomly-generated indices shown in Fig. 9.

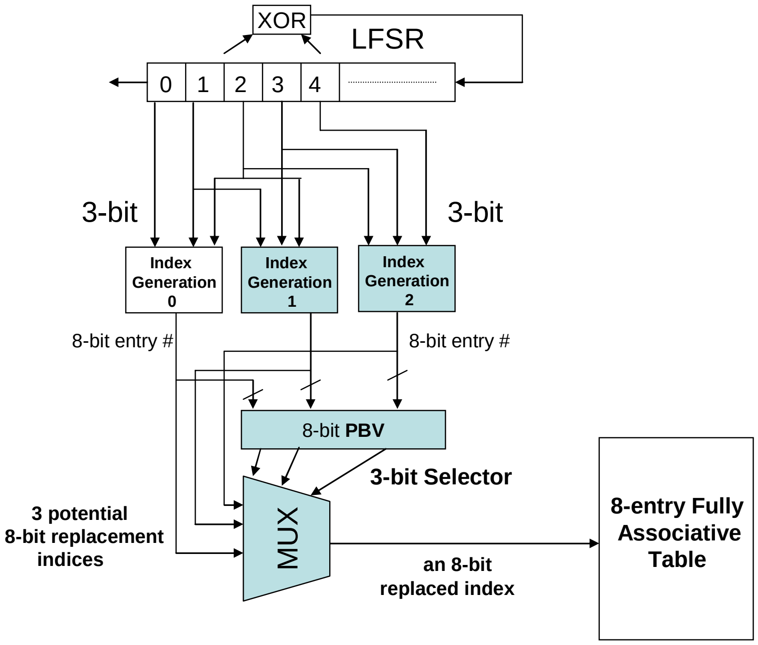


Fig. 9. Low-cost high-performance implementation of 3-index PRR scheme

In this paper, we have proposed a novel random replacement policy for fully or set associative structures such as TLBs to improve the performance of the main or high-priority thread running in an SMT processor along with other low-priority threads. The novel random replacement policy is thread-aware and picks up the first low-priority thread entry index out of many randomly-generated index numbers as a victim entry.

Although we apply this novel technique to TLBs, it could be just as well applied to any associative structures (e.g. L1 and L2 caches, branch target buffers and etc.) that use random replacement policy.

In this paper, we apply this technique in a dual-thread SMT processor but it can also be used in an SMT processor with more than 2 threads. Running several simultaneous LP threads increases the chance of replacing an HP thread entry if the traditional random replacement is used in a soft real-time SMT processor with one HP and multiple LP threads. However, the priority-driven random replacement could significantly improve the performance of the HP thread in the same processor model.

References

1. Tullsen, D.M., Eggers, S.J., Levy, H.M.: Simultaneous Multithreading: Maximizing On-chip Parallelism. International Symp. on Computer Architecture (ISCA) (1995)
2. Marr, D.T., Binns, F., Hill, D.L., Hinton, G., Koufaty, D.A., Miller, J.A., Upton, M.: Hyper-threading Technology Architecture and Microarchitecture. Intel Technology Journal 3(1) (February 2002)

3. Eskesen, F.N., Hack, M., Kimbrel, T., Squillante, M.S., Eickemeyer, R.J., Kunkel, S.R.: Performance Analysis of Simultaneous Multithreading in a PowerPC-based Processor. The Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD'02) held in conjunction with the International Symposium on Computer Architecture (ISCA) (June 2002)
4. Raasch, S.E., Reinhardt, S.K.: Applications of Thread Prioritization in SMT Processors. In: Proceedings of Multithreaded Execution, Architecture and Compilation Workshop (January 1999)
5. Dorai, G.K., Yeung, D.: Transparent Threads: Resource Sharing in SMT Processors for High Single-Thread Performance. In: Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques (2002)
6. Cazorla, F.J., Knijnenburg, P.M.W., Sakellariou, R., Fernández, E., Ramirez, A., Valero, M.: Predictable performance in SMT processors. In: Proceedings of the 1st Conference on Computing Frontiers (April 2004)
7. http://www.eembc.org/

Architectural Solution to Object-Oriented Programming

Tan Yiyu[1], Anthony S. Fong[2], and Yang Xiaojian[1]

[1] College of Information Science and Engineering, Nanjing University of Technology
NO.5 Xinmofan Road, Nanjing, China
tan_yiyu72@163.com
[2] Department of Electronic Engineering, City University of Hong Kong
Tat Chee Avenue, Kowloon Tong, Hong Kong
anthony.fong@cityu.edu.hk

Abstract. Object-oriented programming has become a major trend in software development for large-scale software systems. However, the classic von Neumann architecture machines have certain limitations for object-oriented computing, such as system security and overhead. To address these limitations, architectural support on object-oriented programming has been introduced. In this paper, an architectural solution to object-oriented programming in a Java processor named jHISC is described, where a new object representation model is mapped into hardware directly and the object-oriented programming features is implemented through controlling the related fields in the object context. Moreover, the object representation model is designed to access object information in parallel to speed up object-oriented operation. Compared with PicoJava II, JOP, JDK1.5.0_05 interpreter and HotSpot JIT compiler, it has a great improvement on execution of Java programs.

Keywords: Java, Java processor, Object-oriented programming.

1 Introduction

Ever since the introduction of computer, hardware has become increasingly smaller, faster, and cheaper, whereas software has become larger, slower, and more expensive to build and maintain. Especially, with the rapid development of network and Internet, there is a high demand for developing highly reliable and easily maintainable application programs in a wide range of application domains. Object-oriented programming (OOP) has become firmly established as the methodology of choice for developing new systems due to its advantages, such as reusability, maintainability, flexibility and modularity. Object-oriented programming facilitates its advantages through introducing data encapsulation, information hiding, object inheritance, and polymorphism, thus it enhances the quality of software and reducing the software development cost, especially when amortized over several iterations. Currently, object-oriented programming has become a major trend in software development for large-scale software systems.

© Springer-Verlag Berlin Heidelberg 2007

Object-oriented programming is supported through compilation or virtual machine in classic von Neumann architecture machines. In the compilation approach, an application written in an object-oriented programming language, such as C++, is compiled into the executable native instructions. At the same time, a process will be created for the execution of the program. Different applications are executed in their own addressing spaces, and they are invisible from each other by using virtual memory system. Security protection mechanism is normally implemented with page or segment table, where access right information is maintained.

In the virtual machine approach, a virtual machine is built on the top of operating system. The related object-oriented applications are executed through software emulation in the virtual machine. During execution, the virtual machine executes all the object operations, such as object creation, object communication, dynamic object linking, class loading, and so on.

However, the classic von Neumann architecture machines have certain limitations for object-oriented computing. In the classic von Neumann architecture machines, a word can be treated as either an instruction or datum because no specific semantics associated with the contents of each word, therefore the system is easy to be attacked. To improve security, most object-oriented programming languages introduce data type and access right checks when data are accessed. However, the von Neumann architecture machines do not provide instructions to manipulate objects and simply retrieve the representations of objects from data. Therefore, although a secure object-oriented system is used, a machine-code programmer could directly access data and misinterpret or corrupt the secure objects. For example, viruses can access data directly through load/store instructions, cast them into memory addresses, and corrupt the host system by operating the data stored in the addresses.

On the other hand, because hardware does not support data type and access right checks provided by object-oriented programming languages, such checks impose considerable overhead. For example, in the compilation approach, some code will be inserted to perform these checks, which will slow down the program execution. Although in the virtual machine approach, these checks can be performed by the virtual machine, two layers of software: virtual machine and operating system, introduce large overhead to system. Moreover, the object-oriented operations are executed through software emulation, which will slow down the execution.

To address these limitations, architectural support on object-oriented programming has been introduced, where object operations were carried out directly by an object-oriented processor. Objects are managed by the object-oriented operating system with the protection features offered by the object-oriented processor. Therefore, object manipulation becomes more direct and secure. In this paper, architectural support on object-oriented programming in a Java processor is introduced. The rest of this paper is structured as follows. The previous related work on object-oriented processors is discussed in Section 2. Object representation model is described in Section 3. The implementation of object-oriented programming features is introduced in Section 4. The results of system performance estimation are presented in Section 5. Finally, conclusions are made in Section 6.

2 Related Work

Various solutions to architectural support on object-oriented programming have been provided in many previous machines. The Intel iAPX432 was the first commercial object-oriented architecture [1], which provided hardware support for data hiding, methods, inheritance, late binding and access protection. Despite these advanced features, it was sometimes from 2 to 23 times slower than an 8086 [2]. One reason is due to the architectural limitations, such as the lack of local data registers or a data cache, the fault-tolerant and asynchronous bus/memory interface which resulted in 25% to 40% of the access time consumed by wait state, and so on. Another main reason comes from object orientation, especially procedure calls and returns [3]. On the iAPX432, when an object is accessed, the capability specifier selects an access descriptor which contains access rights information and indices to retrieve the object descriptor from object tables. The object descriptor contains the base address and length of the referenced object [4]. The object-oriented operations are expensive because they need to maintain and traverse more complex addressing information obtained through table lookup, for example, a procedure call references memory 40 times and consumes 724 clock cycles altogether on the iAPX432 [3].

To improve the architectural support on object orientation, some techniques have been proposed. The SOAR architecture [5], which was based on RISC architecture and targeted Smalltalk programming language, employed register sets to hold data, and cached the destination addresses of objects to reduce table lookup during object-oriented operations. It also tagged words to distinguish integers and pointers to support generational garbage collection. The Caltech Object Machine (COM) [6], which was oriented to the late binding object-oriented programming languages, provided hardware method lookup and maintained addressing information in an associative context cache to speed up object-oriented operations. Moreover, an instruction translation look-aside buffer was used to translate a message name to a method address. The MUSHROOM architecture [7] absorbed some techniques proposed before, such as tagged memory and parallel tag-checking, register windows. In addition, it provided an object-based virtual memory system to support garbage collection and employed a novel object cache to maintain the real address information of object. REKURSIV processor [12] offered an object-oriented memory management unit to swap objects in and out of memory as needed because objects were represented directly and mapped into a persistent storage.

There are many object-oriented processors for certain object-oriented programming languages. PicoJava II, developed by Sun Microsystems, targeted Java programming language and mainly performed object-oriented operations by software traps or microcode [8]. Anthony Fong proposed HISC architecture [13], which extended typical computer architecture to support object-oriented programming at the hardware level by introducing 128-bit operand descriptors to describe both object references and variables. In this paper, architectural support on object-oriented programming in a Java processor named jHISC is introduced. In jHISC, object is represented and mapped into hardware directly. And the object representation model is designed to access object information in parallel to speed up object-oriented operation.

3 Object Representation Model

An object consists of many fields in object-oriented programming system. Object representation model is critical because of its significant impact on the speed of accessing object. In general, to minimize the storage overhead, the object header is as small as possible and contains sufficient information about the object. Moreover, system should locate object fields quickly through an object reference.

Inside an object, when a field is accessed, the base reference of the object firstly needs to obtain; then a field offset and some other information, such as access right, field type, etc., are needed. Once all the security and data type checks are passed, the field can then be accessed. All these depend on the object representation model and may be done serially or parallel. For example, in the stack-based implementation of a Java virtual machine, such as JDK1.0 and JDK1.1, they are done serially, which in turn affects the execution speed of Java programs. In jHISC, three kinds of contexts, namely instance, class, and method contexts, are mapped into the hardware architecture to represent different objects. The different object context structures and their relations are shown in Fig. 1.

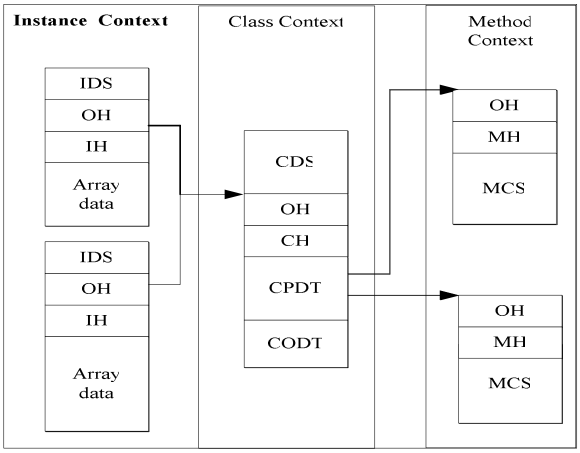


Fig. 1. Different object structures and their relations

Except the object header (OH), an instance context includes Instance Header (IH) and Instance Data Space (IDS); a class context consists of Class Header (CH), Class Operand Descriptor Table (CODT), Class Property Descriptor Table (CPDT) and Class Data Space (CDS); a method context contains Method Header (MH), Method Code Space (MCS) and Local Variable Frame (LVF) for local variable storage. When an instance context is used to represent and array, it contains the Array data, which locates under the Instance Header. Inside the class context, CODT and CPDT store the class operand descriptors and class property descriptors, respectively. Different objects are recognized by the object header which format is shown in Fig. 2.

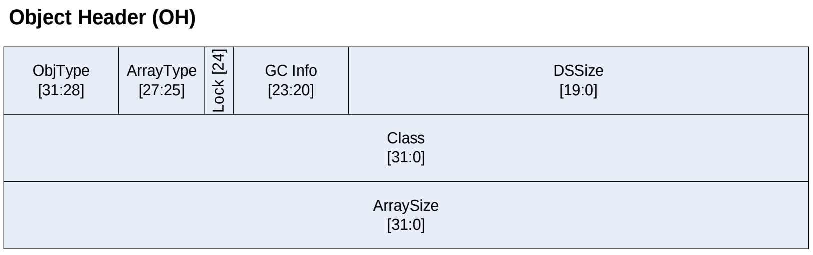


Fig. 2. Format of an object header

Inside the object header, the object type is stored in the field ObjType, such as method, instance, class and array; the field DSSize specifies the size of related data space, such as IDS, CDS; the field GCInfo stores information for hardware-based real-time garbage collections; the field Class holds a direct reference address to link an instance object with its affiliated class; the field Lock is used for multithreading; when the object is an array, the field ArraySize and ArrayType define the number and type of the elements in an array, respectively.

Each object has a unique object context and a reference always points to the base address of object header after the object is resolved. In an object context, all components are stored continuously with each having a constant address offset to the object header, thus allowing the access of some components in parallel to reduce the access overhead. When an object is accessed, the related operand descriptor is read from the operand descriptor table to verify whether the object is resolved or not, then the specific object header is accessed through the direct address pointer stored in the CDS of current class. Along with the object accessing, the bound control checks, such as access permission, boundary and data type, are also carried out by hardware. Moreover, both class variables and instance variables are stored in the related data spaces, therefore they are accessed by their references directly and not accessed through an intermediate object handle as Sun's JDK 1.0 and 1.1.

3.1 Descriptor Format

In jHISC, 32-bit operand descriptor stores information about variables or references. Its uniform format is shown in Fig. 3, which consists of Address Field, Type Field, Static Flag, Access Modifier, Read-Only Flag, and Resolved Flag. Address Field provides byte offset to locate data in the corresponding data space. Access Modifier is used for security control, such as public, private, protect, and so on. TypeField stores the type of described data and eight types are defined for both primitive and reference types. Static Flag indicates where data are stored. For the non-static fields, their values are stored inside the Instance Data Space (IDS) while they are stored inside the Class Data Space (CDS) for the static fields. Read-Only Flag represents whether the target can be written or not. Resolved Flag indicates whether the reference is resolved or not. If not, the system will be trapped to the operating system routines for dynamic reference resolution.

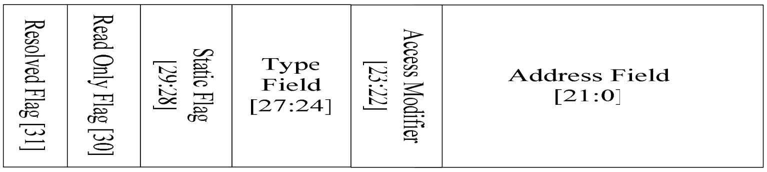


Fig. 3. Operand descriptor format

Two kinds of operand descriptors, class operand descriptor and class property descriptor, are defined to assert the resources accessed by the class and the properties owned by the class, respectively. Normally, a class operand descriptor contains the Address Field, Type Field and Resolved Flag while only the Resolved Flag are not included in a class property descriptor.

4 Implementation of Object-Oriented Programming Features

Object-oriented programming has four key features distinguished with other programming paradigms, namely data abstraction and encapsulation, inheritance, and polymorphism. In jHISC, the object-oriented programming features are implemented by mapping the object representation model into hardware and controlling the corresponding fields into the object context, such as CPDT, CODT, CDS, etc. In this section, how to implement the key features of object-oriented programming is discussed.

4.1 Data Abstraction and Encapsulation

Data abstraction denotes the essential characteristics of an object that distinguishes it from others. Data encapsulation hides all the implementation details of an object inside the class definition while presenting a well-defined interface to the outside world via the class's methods. Moreover, data encapsulation provides the additional access control mechanism to ensure data to be accessed legally and safely. In jHISC, data are described by operand descriptors and their values or reference addresses are stored in the related data space (i.e. IDS, CDS, MCS and Array data). In a class object, its properties and the accessed resources are described by the class property descriptors and class operand descriptors, respectively. The references of other objects or variable values are stored in the class data space (CDS). For an instance object, the instance data are stored in the instance data space (IDS) directly. For an array object, the values of array elements are stored in the array data area. For a method object, the bytecode instructions are stored in the method code space (MCS). In the operand descriptors of an object, the field Access Modifier defines the access rights. Before a program accesses an object, it needs to pass the bound access control and data type checks. Unauthorized or malicious accesses are prohibited.

4.2 Inheritance

Inheritance allows a subclass to share the properties of its superclass to provide a mechanism for code sharing and reuse so that the programming development effort is reduced. A subclass may select which properties of its superclass to inherit. It may also extend its superclass by adding new properties and selectively overriding the existing properties of its superclass, which allows the subclass to be specialized and the superclass to be generalized [9][10]. In jHISC, variables and methods belonging to a class are described by the property descriptors, which reside in the CPDT of the class context. The inheritance feature will be implemented by appending the inherited properties from the superclass and the related addresses into the CPDT and CDS of the subclass context, respectively.

In the example shown in Fig.4, the class ParentClass contains four integer variables, a, b, c, d and two methods, Method_A() and Method_B(). The class ChildClass extends from the class ParentClass. The corresponding object context structures are shown in Fig. 5.

```
class ParentClass
{
   public int a, b, c;
   public static int d;
   public void Method_A() {
         }
   public void Method_B() {
      }
}
class ChildClass extends ParentClass {
      public void Method_C() {
         }
    .
}
```

Fig. 4. A Java example about inheritance

In Fig. 5, all the methods and variables in the class ParentClass are inherited to the subclass ChildClass. The method code spaces of the inherited methods are shared between the two classes and only the method Method_C() is created for the subclass ChildClass specifically. For the static variable d, a direct address is stored in the CDS of class ChildClass, which points to the address where d is stored.

Two special cases, method overloading and overriding, are met in inheritance. Method overloading allows two methods to have the same names, but with different signatures. In jHISC, the overloaded method is treated as a new method added to the subclass. Thus the related operand property descriptor and direct address about the overloaded method are appended to the end of the CPDT and CDS in the object context of subclass, respectively.

In the case of method overriding, a subclass redefines the methods or variables inherited from its superclass. In jHISC, the overridden method reserves the descriptor in the same position in the superclass context, but the related descriptor in the subclass context is replaced by a new one. When a variable is declared as a different type in the subclass, the overridden variable is treated as a new variable in the subclass.

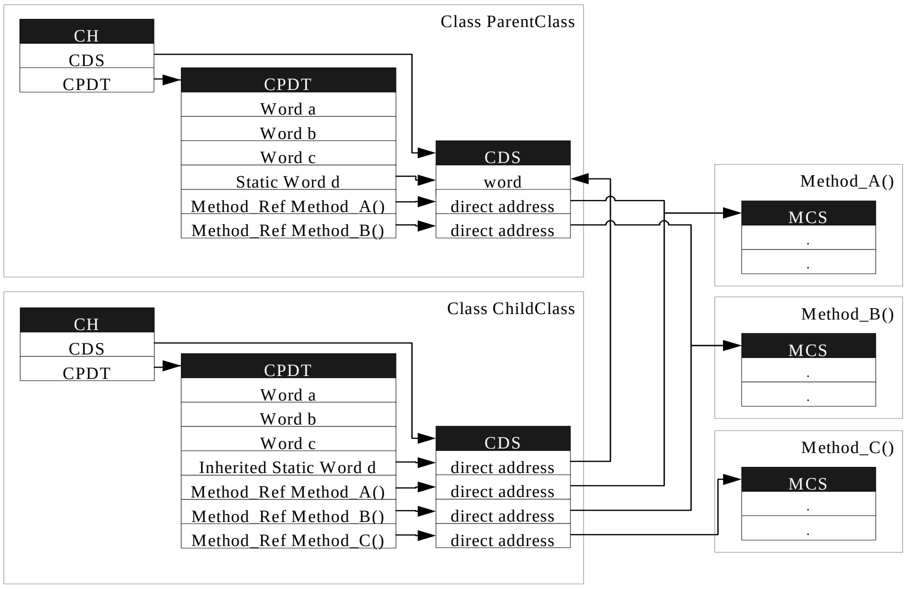


Fig. 5. Object context and relations in inheritance

4.3 Polymorphism

Polymorphism allows many types to be treated as if they were one type, and a single piece of code to work on all those different types equally. It is supported by dynamically binding an object to the appropriate method, but the actual binding occurs at runtime. In a Java program about polymorphism shown in Fig. 6, the class Shape establishes a common interface to any class inherited from it. The derived classes override these definitions to provide unique behavior for each specific type of shape, which causes that the method draw() is defined in the classes Shape, Circle and Square, and each of them has its own implementation. In the class Shapes, the polymorphic method draw() is invoked.

In jHISC, Polymorphism is performed by the resolution process and the corresponding object contexts and their relations are shown in Fig. 7. When the method main() is invoked, the instances a and b are created with the keyword new and the appropriate types (namely Square and Circle). But their instance references are upcasted to the Shape. When the method invocation a.draw() is executed, the instance a and the field reference Shape.draw() are needed. Since it is the first time to access the field reference Shape.draw(), system will trap to the dynamic resolution routine. From the instance a, the resolution process finds out that the reference Shape.draw() for instance a is bound to the method Square.draw(). The resolution process then sets up the related descriptors, field reference Shape.draw(), and the direct address pointing to the object header of class Square. After these, Invocation resumes with the resolved references and system executes instructions inside the invoked method.

When the second method invocation, b.draw(), is executed, system will use the same field reference Shape.draw() and the instance b. Because the operand descriptor is resolved, no resolution is needed for the field reference Shape.draw(). However, during invocation, the access checking finds that the affiliated class of the instance b is Circle, and is different with the resolved class reference Shape, which is actually pointing to the class Square. Thus it will cause a trap to the resolution routine to find whether the class Circle is a subclass of Shape. If yes, the resolution process finishes. And the direct address for the field reference Shape.draw() is set up to point to the object header of the class Circle, then the execution resumes with the resolved reference. If no, an error occurs and exception handler will be performed.

```
class Shape
{
    public void draw() {
   }
}
class Square extends Shape {
   public void draw() {      // <-- overridden method
   }
   .   // other methods or variables declaration
 }
class Circle extends Shape {
   public void draw() {      // <-- overridden method
   }
    .   // other methods or variables declaration
}
class Shapes {
   public static void main(String[] args) {
      Shape a = new Square();      // <-- upcasting Square to Shape
      Shape b = new Circle();      // <-- upcasting Circle to Shape
      a.draw();         // draw a square
      b.draw();         // draw a circle
   }
}
```

Fig. 6. A Java example about polymorphism

5 Performance Estimation

The performance of a processor can be defined as the time to execute a specific program, which is the product of three elements: the weighted average number of cycles per instruction (CPI), the cycle time and the number of instructions executed. We analyzed the distribution of bytecodes in the benchmark JVM98 [14] and clock cycles needed for the execution of each bytecode, and then normalized them to get the weighted average number of cycles per bytecode (CPI) to estimate the system performance. The cycles needed by some main object manipulation bytecodes in jHISC, PicoJava II and JOP are shown in Table 1. In Table 1, the cycle counts for jHISC are based on its RTL model, and for JOP, they are obtained from [11] by assuming the number of clock cycles to access memory is one. In PicoJava II, the cycles consumed by the original formats of object-oriented related bytecodes are estimated by totaling all the clock cycles taken by the relevant bytecodes in the software traps, and the cycles needed by the quick variants are quoted from its data sheet. We can find that the object-oriented bytecodes are executed much faster in jHISC than in PicoJava II and JOP.

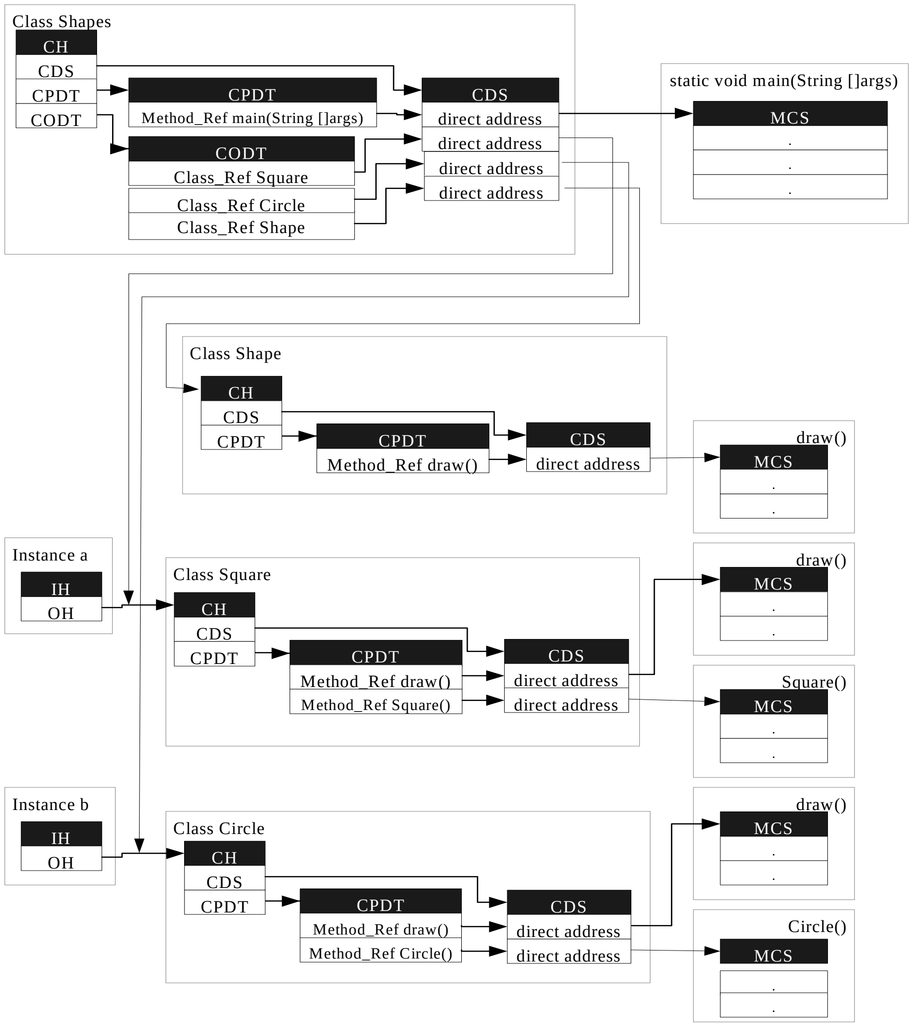


Fig. 7. Object contexts and relations in polymorphism

Table 1. Cycles needed by some object manipulation bytecodes in jHISC, PicoJava II and JOP

Bytecodes in PicoJava II	Cycles		JOP	Instruction in jHISC	Cycles
	Original format	Quick variant			
getfield	114	4	12	gfld	6
				gifld	2
putfield	130	4	15	pfld	6
				pifld	2
getstatic	103	3	6	gsfld	6
putstatic	103	3	7	psfld	6
invokestatic	86	11	67	ivkclass	9
invokevirtual	195	15	88	ivkintance	9
				ivkinternal	5
invokespecial	208	17	67	ivkintance	9
invokeinterface	203	184	96		
checkcast	97	6		checkcast	3
instanceof	100	7		instanceof	4
ireturn	8		19	oo_rvk	5
return			17		
areturn			19		
return			19		
iaload	5		24	arrayload	3
aaload					
caload					
iastore	7		26	arraystore	3

Table 2 shows the estimation results of CPI in PicoJava II, JOP, jHISC, and JDK1.5.0_05, which are obtained through normalizing the distribution of bytecodes in the benchmark JVM98 and clock cycles needed for the execution of each bytecode. In the table, we choose JDK1.5.0_05 (interpreter mode) as a speedup comparison. We observe that jHISC speeds up the overall performance from 0.74 (3.54/2.04-1) to 11.45 (25.4/2.04-1) times against PicoJava II, 3.18 (8.53/2.04-1) times against JOP,

Table 2. Estimation results of CPI in PicoJava II, JOP, jHISC and JDK1.5.0_05

	CPI	Relative Performance	Performance Improvement
JDK1.5.0_05 (interpreter mode)	28.58	1	--
PicoJava II(original format)	25.40	1.13	13%
JOP	8.53	3.35	235%
JDK1.5.0_05 (HotSpot mode)	4.15	6.89	589%
PicoJava II (quick format)	3.54	8.07	707%
jHISC	2.04	14.01	1,301%

13.01 (28.58/2.04-1) times against JDK1.5.0_05 interpreter and 1.03 (4.15/2.04-1) times over JDK1.5.0_05 HotSpot JIT compiler.

6 Conclusion

jHISC offers an attractive solution to speed up the Java program execution while enforcing the features of object-oriented programming. It is possible to provide hardware support on object-oriented programming by controlling the related fields in the object context because a new object representation model is used and mapped into hardware directly. Moreover, parallel accesses of fields in the object context contribute to the execution performance improvement of object-oriented operations.

Acknowledgment

This work is partially supported by the City University of Hong Kong under Strategic Research Grant 7001847.

References

[1] Intel Corporation: iAPX432 General Data Processor Architecture Reference Manual (1983)
[2] Hansen, P.M., Linton, M.A., et al.: A Performance Evaluation of the Intel iAPX432. Computer Architecture News 10(4), 17 (1982)
[3] Gehringer, E.F., Colwell, R.P.: Fast Object-Oriented Procedure Calls: Lessons from the Intel 432. In: Proceedings of the 13th Annual International Symposium on Computer Architecture, 1986, pp. 92–101 (1986)
[4] Budde, D.L., Colley, S.R., Domenik, S.L., et al.: The Execution Unit for the VLSI 432 General Data Processor. IEEE Journal of Solid-State Circuits 16(5), 514–521 (1981)
[5] Ungar, D., Blau, R., Foley, P., et al.: Architecture of SOAR: Smalltalk on a RISC. In: Proceedings of the 11th Annual Symposium on Computer Architectures, June 1984, pp. 188–197, 1984
[6] Dally, W.J., Kajiya, J.T.: An Object Oriented Architecture. In: Proceedings of the 12th International Symposium on Computer Architecture, June 1985 pp. 154–161, 1985
[7] Wolczko, M., Williams, I.: The Influence of the Object-Oriented Language Model on a Supporting Architecture. Proceeding of the 26th Hawaii International Conference on System Sciences 1, 182–191 (1993)
[8] Sun Microsystems: PicoJava-II: Java Processor Core, Sun Microsystems data sheet (April 1998)
[9] Eliens, A.: Principles of Object-Oriented Software Development. Addison-Wesley Publishing Company (1995)
[10] Eckel, B.: Thinking in Java, Third edn., Prentice Hall (2003)
[11] Schoeberl, M.: JOP: A Java Optimized Processor for Embedded Real-Time Systems, PHD thesis, http://www.jopdesign.com
[12] Harland, D.M.: REKURSIV: Object-oriented Computer Architecture. Ellis Horwood Limited (1988)
[13] Fong, A.S.: HISC: A High-level Instruction Set Computer. The 7th European Simulation Symposium, October 1995, pp. 406-410, 1995
[14] SPEC. JVM98 benchmark suits. http://www.spec.org/jvm98/

Author Index