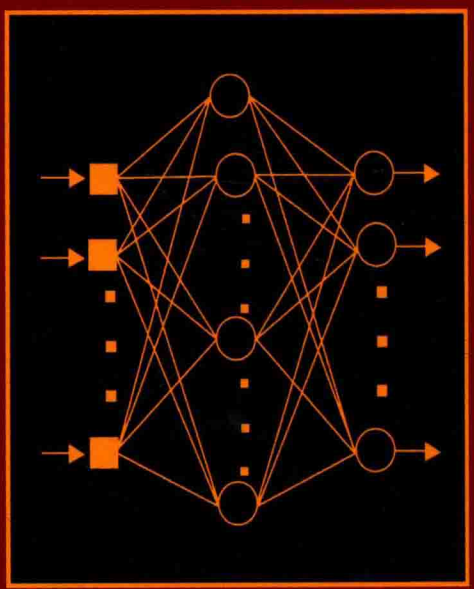
Advances in COMPUTERS

Volume 92



Edited by
ALI HURSON

Series Editors
Ali Hurson and Atif Memon

ADVANCES IN COMPUTERS

Edited by

ALI HURSON
Department of Computer Science
Missouri University of Science and Technology
325 Computer Science Building
Rolla, MO 65409-0350
USA
Email: hurson@mst.edu

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO
Academic Press is an imprint of Elsevier

Academic Press is an imprint of Elsevier
225 Wyman Street, Waltham, MA 02451, USA
525 B Street, Suite 1800, San Diego, CA 92101-4495, USA
The Boulevard, Langford Lane, Kidlington, Oxford, OX5 1GB, UK
32 Jamestown Road, London NW1 7BY, UK
Radarweg 29, PO Box 211, 1000 AE Amsterdam, The Netherlands

First edition 2014

Library of Congress Cataloging-in-Publication Data
A catalog record for this book is available from the Library of Congress

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library

ISBN: 978-0-12-420232-0
ISSN: 0065-2458

For information on all Academic Press publications visit our web site at *store.elsevier.com*

Printed and bound in Singapore by Markono Print Media Pte Ltd
Transferred to digital print 2013

PREFACE

Traditionally, *Advances in Computers*, the oldest Series to chronicle the rapid evolution of computing, annually publishes several volumes, each typically comprising of five to eight chapters, describing new developments in the theory and applications of computing. The theme of this 92nd volume is inspired by the advances in information technology. Within the spectrum of information technology, this volume touches a variety of topics ranging from software to I/O devices. The volume is a collection of five chapters that were solicited from authorities in the field, each of whom brings to bear a unique perspective on the topic.

In Chapter 1, "Register-Level Communication in Speculative Chip Multiprocessors," Radulović *et al.* articulate the advantages of having register-level communication as a part of the thread-level speculation mechanism in Chip Multiprocessors. In addition, this chapter presents a case study addressing the Snoopy Inter-register Communication protocol that enables communication of the register values and synchronization between the processor cores in the CMP architecture over a shared bus. This chapter covers issues such as *thread-level speculation mechanism*, *thread identification*, *register communication*, and *misspeculation recovery*.

In Chapter 2, "Survey on System I/O Hardware Transactions and Impact on Latency, Throughput, and Other Factors," Larsen and Lee survey the current state of high-performance I/O architecture advances and explore its advantages and limitations. This chapter articulates that the proliferation of CPU multicores, multi-GB/s ports, and on-die integration of system functions requires techniques beyond the classical approaches for optimal I/O architecture performance. A survey on existing methods and advances in utilizing the I/O performance available in current systems is presented. This chapter also shows how I/O is impacted by latency and throughput constraints. Finally, an option to improve I/O performance is presented.

The concept of "Hardware and Application Profiling Tools" is the main theme of Chapter 3. In this chapter, Janjusic and Kavi describe widely acceptable hardware and application profiling tools along with a few classical tools that have advanced in the literature. A great number of references are provided to help jump-start the interested reader into the area of hardware simulation and application profiling. The discussion about application

profiling is interleaved with terms that are, arguably incorrectly, used interchangeably. Thus, this chapter makes an effort to clarify and correctly classify tools based on the scope, interdependence, and operation mechanisms.

In Chapter 4, "Model Transformation Using Multiobjective Optimization," Mkaouer and Kessentini propose the application of genetic algorithm for model transformation to ensure quality and to minimize the complexity, two important conflicting parameters. Starting from the source model, randomly a set of rules are generated and applied to generate some target models. The quality of the proposed solution (rules) is evaluated by (1) calculating the number of rules and matching metamodels in each rule, and (2) assessing the quality of generated target models using a set of quality metrics. This chapter reports on the validation results using three different transformation mechanisms.

Finally, in Chapter 5, "Manual Parallelization Versus State-of-the-Art Parallelization Techniques: The SPEC CPU2006 as a Case Study," Vitorović *et al.* attempt to articulate the importance of manual parallelization of applications. This chapter studies various parallelization methods and contemporary software and hardware tools for extracting parallelism from sequential applications. It also attempts to identify typical code patterns amenable for parallelization. As a case study, the SPEC CPU2006 suite is considered as a representative collection of typical sequential applications. The automatic parallelization and vectorization of the sequential C++ applications from the CPU2006 suite are discussed, and since these potentials are generally limited, it explores the manual parallelization of these applications.

I hope that you find these articles of interest and useful in your teaching, research, and other professional activities. I welcome feedback on the volume and suggestions for topics for future volumes.

Ali R. Hurson
Missouri University of Science and Technology
Rolla, MO, USA

CONTENTS

CHAPTER ONE

Register-Level Communication in Speculative Chip Multiprocessors

Milan B. Radulović, Milo V. Tomašević, Veljko M. Milutinović
School of Electrical Engineering, University of Belgrade, Belgrade, Serbia

Contents

Abstract

The advantage of having register-level communication as a part of the thread-level speculation (TLS) mechanism in chip multiprocessors (CMPs) has already been clearly recognized in the open literature. The first part of this chapter extensively surveys the TLS support on the register level in CMPs. After the TLS mechanism is briefly explained, the classification criteria are established, and along them, the most prominent systems of this kind are elaborated upon focusing on the details about register communication. Then, the relevant issues in these systems such as thread identification, register communication, misspeculation recovery, performance, and scalability are comparatively discussed. The second part of the chapter represents a case study that describes the snoopy interregister communication (SIC) protocol that enables communication of the register values and synchronization between the processor cores in the

Advances in Computers, Volume 92
ISSN 0065-2458
http://dx.doi.org/10.1016/B978-0-12-420232-0.00001-5

CMP architecture over a shared bus. The appropriate software tool, which creates and annotates the threads from a sequential binary code of the loop-intensive applications, is described. Also, the states of registers are defined and the protocol actions during producer-initiated and consumer-initiated communication among the threads. Finally, the ESIC protocol, an enhancement of the SIC protocol with more aggressive speculation on the register values, is also presented and compared to the SIC.

LIST OF ABBREVIATIONS

AMA atlas multiadaptive
CMP chip multiprocessor
CRB communication register buffer
CSC communication scoreboard
ESIC enhanced SIC
EU execution units
FOPE fork-once parallel execution
FU functional unit
FW final write
GRF global register file
HW hardware
INV invalid
INVO invalid-others
IPC instructions per cycle
IRB intermediate register buffer
LC last-copy
LRF local register file
MAJC multiprocessor architecture for Java computing
MRF multiversion register file
MUCS multiplex unified coherence and speculation
NFW nonfinal write
PE processing element
PFW possibly final write
PRO propagated
PU processing unit
RAW read after write
RC ready-CRB
RR ready-released
RVS register validation store
RVT register versioning table
SH shared high
SIC snoopy interregister communication
SISC speculation integrated with snoopy coherence
SL shared low
SM speculative multithreaded
SPEC standard performance evaluation corporation
SRB store reservation buffer

SS synchronizing scoreboard
SW software
TD$ thread descriptor cache
TLS thread-level speculation
TU thread unit
UPD updated
VLIW very-long-instruction-word
VPS valid-possibly-safe
VPSF valid-possibly-safe-forwarded
VS valid-safe
VSO valid-safe-others
VU valid-unsafe
WAR write after read
WAW write after write

1. INTRODUCTION

In the previous decade, chip multiprocessors (CMPs) have emerged as a very attractive solution in using an ever-increasing on-chip transistor count, because of some important advantages over superscalar processors (e.g., exploiting of parallelism, design simplicity, faster clock, and better use of the silicon space). Furthermore, nowadays, the memory wall, the power wall, and the instruction-level parallelism (ILP) wall have made the CMP architecture inevitable. In order to attain a wider applicability and being a viable alternative to the other processing platforms, besides running parallel workloads, CMPs also have to be efficient in executing the existing sequential applications as well. The technique of thread-level speculation (TLS) is a way to achieve this goal. In the TLS, even possibly data-dependent threads can run in parallel as long as the semantics of the sequential execution is preserved. A special hardware support monitors the actual data dependencies between threads in run time and, if they are violated, misspeculation effects must be undone. The application threads can communicate between themselves through registers or through shared memory. This kind of system is known as speculative chip multiprocessor [1–7].

The rest of this chapter is organized as follows. The short explanation of the TLS technique illustrated with appropriate examples is given in Section 2. An extensive survey of the most representative speculative commercial and academic CMPs with the TLS support on the register level is presented in Section 3 along the established classification criteria (register file organization and interconnection topology). Section 4 brings a comparative analysis of

relevant issues in these systems such as thread creation and speculation scope, register communication and synchronization mechanisms, misspeculation recovery, and performance and scalability. The case study in Section 5 presents the snoopy interregister communication (SIC) protocol and its enhancement, the enhanced SIC (ESIC) protocol, that enables the communication of register values and synchronization between processor cores in a CMP over a shared bus. Some conclusions are drawn in Section 6.

2. TLS IN CMPs

Multithreading is a technique that partitions a sequential program into a number of smaller instruction streams (threads) that can be executed in parallel keeping different processor cores in a CMP simultaneously busy. If these cores are simple superscalars, the small amount of the ILP can still be exploited on top of the multithreading, since the ILP and multithreading are orthogonal to each other. The best candidates for threads are basic blocks (sequence of instructions with no transfers in or out except at the start and the end), inner- or outer-loop iterations, subprogram calls, etc.

There are two multithreading approaches: *explicit* and *implicit*. The main differences between explicit and implicit multithreading relate to thread dispatch, execution, and communication mechanisms, while the underlying processor architecture and memory hierarchy are similar.

In *explicit* (nonspeculative) multithreading, the threads can be either independent or interdependent but properly synchronized on each occurrence of data dependence, so they can be nonspeculatively executed concurrently in a correct order. The software imposes the fork primitives for thread dispatch to specify interthread control dependencies. The explicit threads in sequential applications can be identified by the advanced parallelizing compilers or manual parallelization [2,7,8]. However, the identification of explicit threads is not easy because of the problems with pointers, conditional execution, etc. Even for numerical applications, the parallelizing compilers have been successful only to a limited extent in directly explicit multithreading. The parallelizing compilers are very conservative during thread identification, because they assume the existence of interthread dependencies whenever they cannot guarantee their independence even where interthread dependencies are not very likely. As illustrated in Fig. 1.1, if the values in arrays L and K are dependent on input data, the compiler cannot determine whether or not loop iterations access distinct array elements, and hence, it marks the loop as serial.

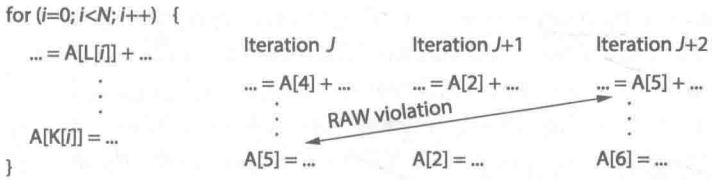


Figure 1.1 Example of the possibly dependent loop iterations. *This figure is taken from [2] with permission from the copyright holder.*

Consequently, CMPs cannot efficiently handle general-purpose sequential applications even with the sophisticated compilers in explicit multithreading. The problem can be solved by exploiting the *implicit* (speculative) multithreading. The implicit threads are identified in the application either during the compile time or during the run time with hardware support and can be (but less likely) interdependent. These threads can be executed speculatively in the CMP with some software or hardware support that can detect and correctly resolve interthread dependencies [2,7,9].

In such a system, the threads run in parallel on the different processors speculatively as long as their dependencies are not violated. If no violation occurs, the thread finishes and commits. In case of misspeculation, the thread that violates the dependence and all its successor threads are squashed and later reexecuted with correct data. The speculation hardware (thread identification, dependence prediction and detection, and data value prediction) guarantees the same result of execution as in a uniprocessor. This technique is referred to as TLS. The speculative thread architecture for mostly functional languages was first proposed in [1] where hardware is used to ensure the correct execution of parallel code with the side effects. Later on, the TLS technique was employed in a number of different CMP architectures (e.g., the Multiscalar is one the earliest tightly coupled multiprocessors fully oriented towards speculative execution [7]).

The speculative parallelism can be found in many sequential applications (e.g., [10]). The TLS provides a way to parallelize the sequential programs without a need for a complex data dependence analysis or explicit synchronization and to exploit the full potentials of CMP architecture in execution of general sequential applications. The threads are committed in the order in which they would execute sequentially, although they are actually executed in parallel. However, the TLS and synchronization are not mutually exclusive. In order to improve the performance, explicit synchronization can be used when the interthread dependencies are likely to occur.

The ideal memory system within hardware support for speculative execution should consist of the fully associative, infinite-size L1 caches intended to keep the speculative states. They should operate in write-back mode to prevent any change in the sequential state held in the L2 cache unless the thread is committed. When the speculative thread i performs a read operation, the speculative hardware must return the most recent value of data. If it is not found in the L1 cache of the processor that executes the speculative thread i, then the most recent value is read from a processor executing a thread j that is less speculative than thread i. If the value is not found in the caches of the processors executing the less speculative threads than thread i, the data are read from the L2 cache or from memory.

More precisely, an adequate hardware TLS support imposes five requirements: (a) data forwarding between parallel threads; (b) detecting read after write (RAW) hazards; (c) safe discarding of speculative state(s) after violations; (d) retiring speculative writes in the correct order, write after write (WAW) hazards; and (e) providing the memory renaming, write after read (WAR) hazards.

Firstly, in case of true data sharing between threads, when a later thread needs shared data, an earlier thread has to forward the most recent value. Sometimes, in order to minimize stalling, the producer thread sends updated data in advance on nondemand basis to the successor threads.

The RAW hazard occurs when a later thread prematurely reads data. Therefore, a situation when more speculative thread first reads a value that is later updated by some predecessor thread must be detected. It is usually resolved by squashing a thread that caused the violation and executing it again with valid data.

In case of violation, the permanent state must not be corrupted and all changes made by the violating thread must be made void. Usually, speculative state is held only in the private L1 cache and permanent state in the L2 cache, so the effects of misspeculation are easy to discard.

When a later thread writes to the same location before an earlier thread updates it (WAW hazard), this write must be seen by other threads in the correct program order. This can be achieved by allowing the threads to update the permanent state after committing strictly in the program order.

Finally, an earlier thread must never read a value produced by a later thread (the WAR hazard). This is ensured by keeping the actual data in the L1 cache that cannot be reached by earlier threads. An illustration for some of described situations is presented in Fig. 1.2.

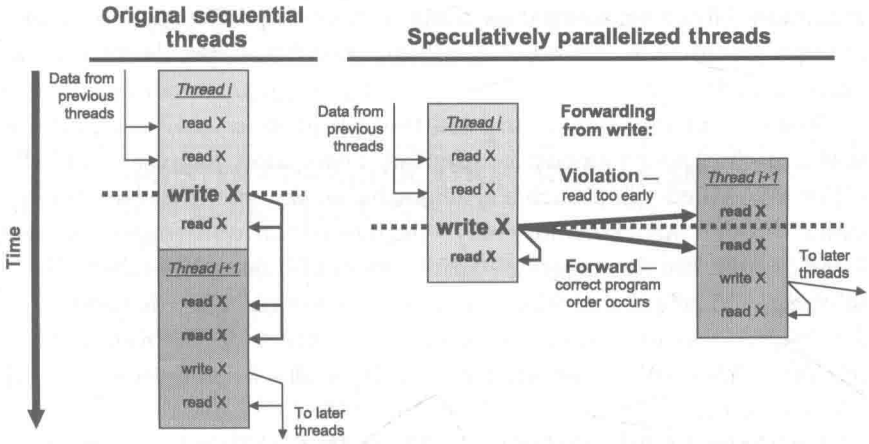


Figure 1.2 An example of data dependencies between the speculative threads. *This figure is taken from [7] with permission from the copyright holder.*

CMP with the TLS support is a high-performance and cost-effective alternative to complex superscalar processors. It has been shown that a speculative CMP of comparable die area can achieve performance on the integer applications similar to the superscalar processor [7]. However, the hardware and software support for speculative execution is not sufficient to ensure that a CMP architecture performs well for all applications. The notorious potential problems with the TLS are the following: (a) a lack of parallelism in the applications, (b) hardware overheads (a large amount of hardware remains unutilized when CMP runs a fully parallel application or a multiprogrammed workload), (c) software overheads in managing speculative threads, (d) an increased latency of interthread communication through memory, and (e) a wasted work that must be reexecuted in case a violation occurred.

Although the speculative parallelism can be exploited by software means only (e.g., [11–13]), most of the TLS systems employ the hardware support usually combined with the software support. There are three major approaches in the design of the speculative CMPs with the TLS hardware support.

The first one is related to the CMP architectures completely oriented towards exploiting speculative parallelism, for example, Multiscalar [14], Multiplex [8], Trace [15], speculative multithreaded (SM) [16], Multiprocessor Architecture for Java Computing (MAJC) [17], MP98 (Merlot) [18], and Mitosis [19]. These systems support interthread communication through both registers and shared memory and they usually have significant and effective hardware and software support for efficient speculative

execution. However, when these CMPs run a true parallel application or a multiprogrammed workload, a large amount of that speculative support remains ineffective.

The second approach is oriented towards generic CMP architectures with a minimal added support for speculative execution, for example, Hydra [7] or STAMPede [20]. Such a system achieves the interthread communication through shared memory only. The limited hardware support in these CMPs is sufficient for correct speculative execution since they rely on compiler support and software speculation handlers to adapt the sequential codes for speculative execution. However, the necessity of source code recompilation can be a serious problem, especially when the source code is not available.

IACOMA [9], Atlas [21], NEKO [22], and Pinot [23] belong to the third approach that tries to combine the best of previous two approaches. They still enable the threads to communicate through both registers and memory as in the first approach, but they have mainly generic CMP architectures with modest hardware (HW)/SW support for speculative execution as in the second approach. Consequently, they can be considered as more cost-effective in running the true parallel or multiprogramming workloads.

Some studies about the impact of communication latency on the overall performance of the speculative CMP argued that a fast communication scheme between the processor cores may not be required and that interthread communication through the memory is fast enough to minimize the performance impact of the communication delays [7,9,24,25]. The limitation of the interthread communication through the memory simplifies the overall design but the need for source code recompilation is still a disadvantage for this group of CMPs.

The support for register-level communication introduces an additional complexity in the system (fast interconnect for exchanging the values, relatively complex logic, etc.). However, earlier studies have shown that register-level communication pays back by avoiding overhead of memory-level communication that requires the instructions to (a) explicitly store and load the communicated values to and from memory and (b) synchronize the communicating threads. It was concluded in [26] that register-based communication is 10 × faster and synchronization is 60 × faster than corresponding memory counterpart mechanism. The impact of having memory-level communication only on the overall performance degradation is evaluated in [27]. It was demonstrated that the communication through L2 cache (and not through registers) in a CMP with four superscalar cores (up to

four-issue) incurs performance degradation of up to 50%. This is a strong support for employment of interregister communication mechanisms in the most of representative CMPs.

3. REGISTER COMMUNICATION MECHANISMS IN SPECULATIVE CMPs

The main goal of this chapter is to present an overview of the various CMP systems with TLS support on register level found in the open literature. There is a variety of issues in reviewing the hardware and software support for interthread register communication such as organization and implementation of the register file, interconnection topology, register communication protocol, thread identification, recovery from misspeculation, and compiler and other software tool support. The organization of register file(s) has a profound impact on the mechanisms of synchronization and communication of the register values between on-chip processor cores in speculative CMPs. Therefore, it is adopted as the main classification criterion for this presentation. Three different approaches can be recognized: distributed (local) register files, unified shared (global) register file, and hybrid design that combines both local register files (LRFs) and a global register file (GRF) (Fig. 1.3).

Traditional microprocessors have been mostly designed with a GRF shared by multiple functional units (FUs), which in turn increases the

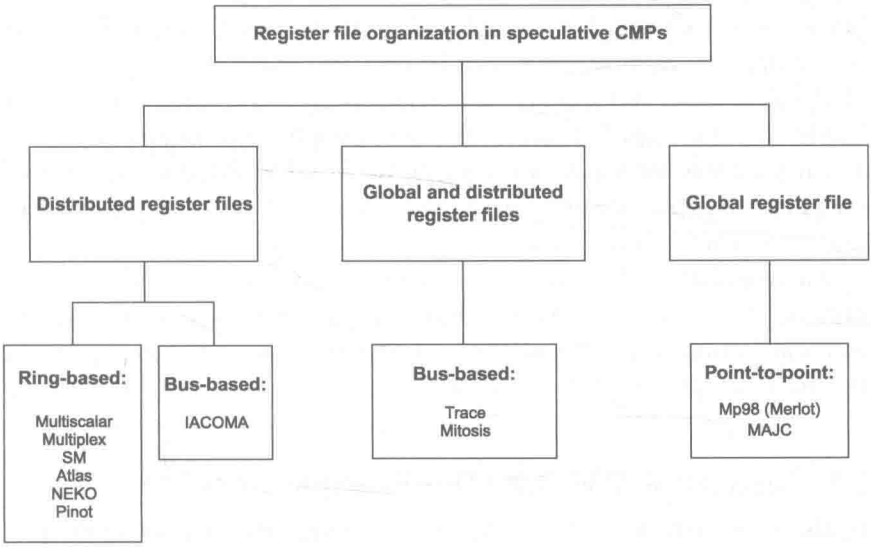


Figure 1.3 Register file organization in speculative CMPs.

number of register file read/write ports and, hence, leads to complex wiring, affects the cycle time, and increases latencies. In order to support higher degrees of instruction-level parallelism (ILP) and thread-level parallelism (TLP), the hardware implementation cost of a design with a central register file and a large and complex operand bypass network grows rapidly. In extremely small feature sizes and highly parallel processor designs like a CMP, a shared register file must be replaced with distributed file structures with local communication paths to alleviate the problems with a large number of long interconnects between the register file and operand bypass structure. This is the reason why the exclusive use of shared register file is scarce. Distributed register organizations scale efficiently compared to the traditional shared register file organization since they significantly reduce area, delay, and power dissipation. Consequently, almost all systems either employ distributed approach only or combine it with small shared register file [28–31].

Another influential design parameter is interconnection topology between cores on register level and it is adopted as a secondary classification criteria. The ring is a main design choice for the operand bypass network on register level in Multiscalar, Multiplex, SM, Atlas, NEKO, and Pinot (see Fig. 1.3). It is a natural choice for interconnection since processor cores during speculative execution primarily communicate with their two nearest neighbors—produced register values are sent to the more speculative core and consumed register values come from the less speculative core. The bus is another choice that is employed in Trace, IACOMA, and Mitosis. The simple bus architecture would be sufficient to handle a small number (4–8) of processor cores, but more cores or faster ones would require higher bandwidth, which in turn demands either more buses or hierarchy of local and global buses. Finally, in systems with GRF, MP98 (Merlot), and MAJC, the point-to-point networks are used as an operand bypass network for register value communication.

General data, architecture details, and register communication mechanisms of the speculative CMPs with the support for register-level communication, grouped by organization of register files and interconnection topology, are presented in this section.

3.1. Speculative CMPs with Distributed Register File

In almost all systems of this kind, LRFs are interconnected by a unidirectional or bidirectional ring except IACOMA, which uses shared bus for register communication.