Streamlined Atomic Execution for Java

by

Lixin Su

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Electrical Engineering)

at the

UNIVERSITY OF WISCONSIN-MADISON

2008

UMI Number: 3327732

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI®

UMI Microform 3327732

Copyright 2008 by ProQuest LLC.

All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest LLC
789 E. Eisenhower Parkway
PO Box 1346
Ann Arbor, MI 48106-1346

Committee's Page. This page is not to be hand-written except for the signatures

A dissertation entitled

Streamlined Atomic Execution for Java

submitted to the Graduate School of the
University of Wisconsin-Madison
in partial fulfillment of the requirements for the
degree of Doctor of Philosophy

by

Lixin Su

Date of Final Oral Examination: August 4, 2008

Month & Year Degree to be awarded: **December** **May** **August** 2008

****

Approval Signatures of Dissertation Committee

Signature, Dean of Graduate School

Committee's Page. This page is not to be hand-written except for the signatures

Abstract

In the era of billion-transistor microprocessors and large, complex software systems, this thesis makes a successful attempt in a co-design in future microprocessors and Java Virtual Machines (JVMs). We propose the concept of hardware atomicity and a hybrid speculative execution model for Java. In our model, hardware exposes the transistors in the form of atomic regions to the JVM. The JVM in turn applies lightweight, speculative, and powerful optimizations to speed up Java applications. The speculative optimizations are within the scope of an atomic hardware region, which is either committed in the end or rolled back during its execution depending on if any contract (or agreement) between the microprocessor and the JVM has been violated. We apply this approach to relax the optimization constraints imposed by Java's precise exception model, a popular software design decision in the safe programming language trend. The developed speculative null pointer check elimination and bounds check elimination algorithms did achieve good speedups of 15% on the studied benchmark suite.

We exploit register calling convention (SW), register dirtiness analysis (SW), and the physical register file free-list buffer (HW) to help reduce register checkpointing cost for hardware atomicity. We also use a stack write logging elimination algorithm (SW), a heap write logging elimination algorithm (SW), region shrinking (SW), and write buffering (HW) to help reduce write logging cost for hardware atomicity. With these techniques, we can remove 98% of register checkpointing and 94% of write logging and preserve 95% of the speedups due to speculative optimizations in the studied benchmark suite.

Acknowledgements

First and foremost, I would like to thank my parents, Gongquan Su and Hongying Zhang, for their upbringing and constant support. They are the most optimistic and calm people that I have ever known and they have deeply influenced my living philosophy. The greatest thing that has ever happened to me is to be their son. My father is a very inspirational person. His quote, "Technology advancement is through the efforts of each individual person even though one's contribution might be very incremental.", has been the motivation that has driven me to finish graduate school and obtain a PhD degree. It will continue to be the drive for my future career.

I would also like to thank my academic advisor, Prof. Mikko Lipasti, for his fiscal and emotional support, for mentoring me how to conduct computer research, and for showing me how to live a balanced life between work and life. His insights on the overall computer research have taught me how to think big and be brave. Coming to graduate school, I was told by prior students to find a nice person and work for him. It was fortunate for me to find a nice person and a great advisor in Mikko.

One can never imagine how one class could change one's PhD journey. The graduate level compiler class CS701 taught by Prof. Susan Horwitz did just that even though I did not realize that my journey was going to be completely altered then. I owe greatly to Prof. Horwitz for her great teaching. In addition, Susan's wonderful comments have made the thesis significantly better.

The interactions with other professors and graduate students have also benefited me by a great deal. I would like to thank Prof. Jim Smith, Prof. Mark Hill, Prof. David Wood, Prof. Guri Sohi, Prof. Mike Schulte, Prof. Parmesh Ramanathan, Prof. Katherine

Compton, Prof. Raghu Ramakrishnan, Prof. Remzi Arpaci-Dusseau, Prof Junichiro Kono (Rice University), Prof. Scott Rixner (Rice University), Prof. William Wilson (Rice University), and Prof. Joseph Cavallaro (Rice University) for all the things that I have leaned from them. I would also like to thank fellow PHARM students, computer architecture students, and computer engineering students for the great environment of peer learning. Among them, I would especially like to thank Eric Hill for help and friendship.

My journey of a computer architecture PhD actually started with a book, written by a former Intel chief architect -- Albert Yu, that I read as an undergraduate student at Fudan. I did not really think I was going to be a microprocessor designer at that time since there was no microprocessor industry in China. But I was greatly inspired by the book and somehow wanted to become a great computer architect. I am indebted to Konrad Lai for opening Intel's door to me four years later. I would also like to thank my mentors at Intel, Shih-Lien Lu, Kingsum Chow, and Youfeng Wu for their wonderful mentoring and the great learning opportunity at Intel.

My study-in-America experiences could not have been complete without the wonderful time I have spent living at the Wayland community. It is a very liberal, multicultural community where people range from very liberal non-christians to ultra-conservative christians who come from all over the world. During my stay at Wayland, I have made friends with people from many countries in all five humanly inhabited continents and learned from them all kinds of cultures. It was a truly unique American experience.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The evolution of computer systems is driven mainly from two distinct yet cooperative forces - hardware and software. On the hardware side, the number of on-chip transistors have increased many folds from the very first microprocessor to the latest chip multiprocessors (CMPs). The ample on-chip hardware resources have given computer architects tremendous opportunities to improve chip design, increase system performance, and add extra value.

Meanwhile, software systems are becoming more sophisticated and complicated [80]. Software system design strives to achieve several mostly compatible but sometimes conflicting goals - speed, usability, construction cost, and maintenance. Speed still remains the most important goal for modern software system designs. It is very common that the performance of a software system could be the major factor in its popularity. However, other goals start weighing more in design decisions. Advanced software system design has become a delicate art of balancing speed with other design goals such as usability, construction cost, and maintenance. Software designers and users (potential buyers) are paying more attention to usability, construction cost, and maintenance when speed is within an acceptable range. Usability could affect the number of users of the software and thus its success. Construction cost and maintenance could affect the software price and thus the software demand.

One particular software system is the compiler[3][26], which ranges from static to dynamic. A static compiler may have an offline profiler and compiles code based on the

profile information. A dynamic compiler may exist inside a virtual machine[90]. Compilers connect the processor Instruction Set Architecture (ISA) to software. They serve a vital role to the overall computer system performance. A static compiler compiles applications into machine-specific executables beforehand and then the executables can run on the target machine. A profile-driven static compiler runs an application in the profiling mode to collect application-specific information and then feeds the information to compilation to generate faster executables for a target machine. A dynamic compiler can detect hot spots in the code at runtime and apply more sophisticated compilations to them dynamically. On the dynamic compiler side of the spectrum, programming language based virtual machines are becoming popular. They usually include a dynamic compiler, an online profiler, and some other resource management parts typically found in an operating system.

The Java Virtual Machine [61] designed for the Java Programming Language [32] combines system resource management (a traditional role of an operating system), hot code detection, runtime profile information collection, and gradual dynamic recompilations. It represents a state-of-the-art compilation system and has established a new trend for future compilation systems. The JVM resides in a unique position in the computer system stack. It can be one hop away from the user via the Java applications. It is also directly connected to a microprocessor since it compiles or interprets code for a microprocessor. It is also a coherent virtual machine that performs many system management functions. Therefore, the interactions between a JVM and a microprocessor provide a unique opportunity to study the co-design between billion-transistor microprocessors and dynamic runtime systems (an example of large-scale software systems).

This thesis focuses on the interactions of Java Virtual Machines and underlying

microprocessors with ample transistors. It explores how to expose the ample transistor budget to the Java Virtual Machine to increase single threaded Java application performance. It also explores hardware and software techniques to reduce the consumption of hardware resources to enable such support.

1.1 Popularity of Java

Java has become one of the most popular programming languages since its invention at Sun Microsystems in the early 1990s. It is widely used in environments such as embedded systems, mobile computers, desktop PCs, and server systems. Many of Java's features have helped Java achieve the current popularity.

- **Portability:** Java code is first compiled into the Java bytecode by a static compiler. The bytecode is platform independent and can be run on any computer platform that has a JVM.
- **Internet Language:** Java was invented when the internet became big. From the very beginning, Java has been designed to facilitate internet programming. Variations of Java, such as the Java servlet [53], can ease internet programming significantly.
- **Safety:** Java is a safe language. It provides many safety checks such as null pointer checks and array bounds checks. It enforces a precise exception model which tracks the order of potential exceptions in a program and helps programmers and users reason about exceptions and programming mistakes.
- **Memory Management:** Java has its own garbage collector in a JVM and frees the programmer from the burden of memory management. It shields pointers from the

programmer and eliminates many bugs from poor pointer usage.

Above are just some major features of Java that have helped with its popularity. There are some other features such as strong types that may have also helped with Java's popularity.

1.2 Improving Java Performance

Java is slow compared with C or C++. Many of Java's popular features introduce overhead and hurt performance. Dynamic loading leads to incomplete class hierarchy information and prevents efficient optimizations based on exact class information. Memory management introduces garbage collection (GC) overhead and GC pauses can be significant and prevent Java from being used in a mission-critical environment. The safety feature introduces checking overhead and needs optimizations to reduce it. The precise exception model sometimes prevents simple and effective optimizations from being used since the rescheduling of potentially excepting instructions can break the precise exception model.

Most attempts to accelerate Java programs have been tried on the software side, especially in JVMs. The dynamic profiler has become more lightweight; the optimizing compiler has become more powerful; and the garbage collector has adopted more efficient algorithms. JVM performance tuning for certain classes of Java applications has also played a role; for example, there are server side and client side JVMs. The server side JVMs are usually tuned for the performance of server applications such as SPECjAppserver and SPECjbb while the client side JVMs are often tuned with applications such as SPECjvm and Java Grande.

Dynamic optimization with online profile information has become a standard Java compilation approach for state-of-the-art JVMs. In this environment, optimization/re-optimization cost due to compile time can also affect program performance. Heavyweight optimizations found in static compilers may not perform well for Java due to the intensive compile time usage. It can help improve Java performance to find the right combination of efficient optimizations for an optimizing compiler in a JVM.

1.3 Co-design for Java

The primary goal of this thesis is to investigate the synergy between a JVM (a large, complex software system) and a future state-of-the-art microprocessor (with abundant transistors onchip). We aim at exposing hardware resources in a quantifiable and atomic fashion to enable the JVM with better control over the abundant onchip transistor resource and an opportunity of better and more powerful optimizations. With the aid of the hardware, the JVM could utilize the hardware in an unprecedented way to relax some of the programming language constraints that are designed for software system implementation goals such as usability and cost but unavoidably introduce compiler optimization constraints.

More specifically as shown in Figure 1-1, we extend the microprocessor Instruction Set Architecture (ISA) to enable the JVM to identify atomic guarded regions in its execution stream. An atomic guarded region is a sequence of code that affects the processor/system state in an all-or-nothing fashion, whose execution is closely monitored by hardware to ensure that the executing code does not create wrong effects on the overall correctness or soundness of the program. In case anything goes wrong, the code sequence

could be completely discarded without affecting the processor/system state. After a region is formed, the JVM could perform aggressive, speculative, yet simple optimization within the region to relax the optimizing constraints imposed by the Java programming language's precise exception model, which could reduce both compile time and code size and further improve program performance.The precise exception model is imposed from the Java applications and the Java programming language for the purpose of a safe programming language and reduced application development and maintenance cost. It imposes a constraint on how the JVM can optimize Java code and prevents certain instructions from being ordered freely. We also look into using the JVM to form atomic regions based on static heuristics and to reduce hardware atomicity support cost.

1.4 Thesis Contributions

The research presented in this thesis makes the following contributions:

- **Hybrid speculative execution model for Java:** We propose a mechanism to allow the speculative optimization of Java code within an atomic region guarded by hardware. Hardware closely monitors the contract agreed between software and hardware and aborts the executing atomic region upon any violated contract. In case of a contract violation, software performs recovery and recompiles the code conservatively during recovery. With this model, the precise exception model of the Java programming language can be relaxed and lightweight speculative algorithms could be used to optimize Java code aggressively.
- **Atomic region formation:** We develop several static heuristics to create atomic regions in the JVM. The JVM relies on the ISA support exposed by hardware to

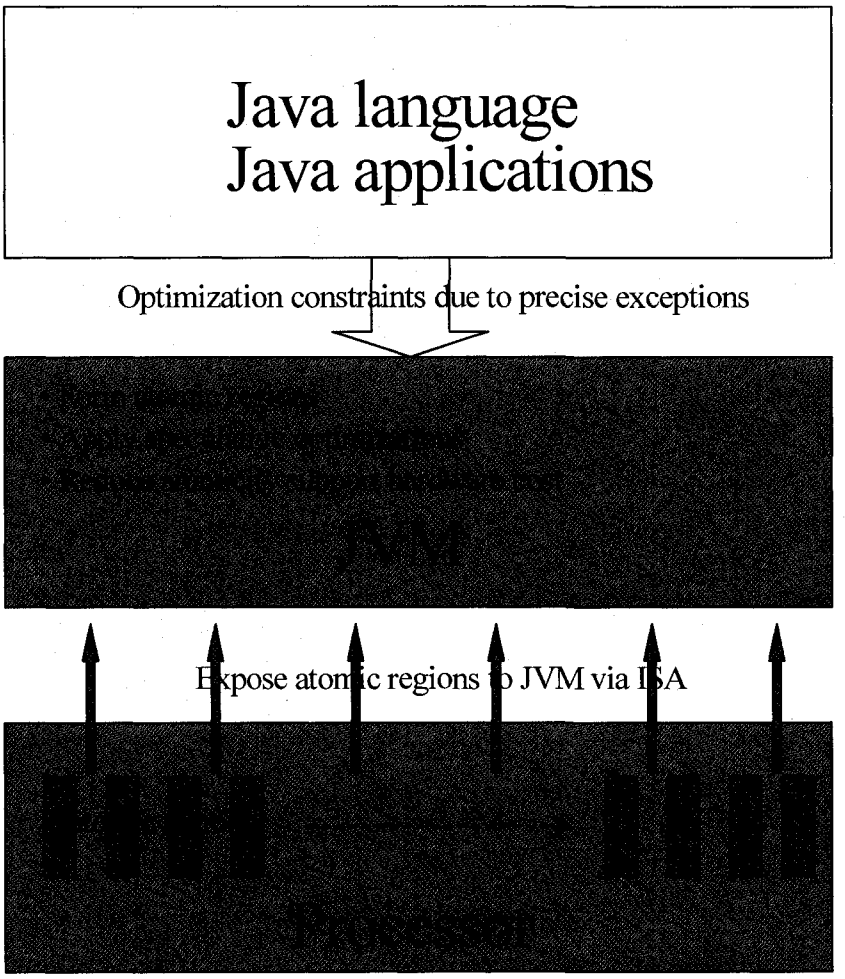


FIGURE 1-1. Thesis overview.

insert atomic region boundaries. We also extend the ISA to enable the flexible creation of atomic regions by software. We compare the pros and cons of these static region formation heuristics and also mention the possibility of dynamic region formation based on profile information.

- **Lightweight speculative algorithms:** We design a set of light-weight, easily

implementable speculative algorithms to relax the optimization constraints from Java's precise exception model. More specifically, we design and implement speculative null pointer check removal and bounds check removal algorithms to reduce compile time, decrease code size, and improve code performance. In our experimental evaluation, we statically compile Java code to the highest optimization level in our JVM and apply the speculative optimization early in the optimization flow. We report that the proposed speculative optimizations achieved on average a speedup of more than 14% for the studied benchmarks. The speculative optimizations could be easily employed in a higher optimization level if the JVM dynamically compiles the code from the lowest to the highest optimization level. Similar performance improvement could be observed in this true dynamic compilation environment.

- **Hardware-software techniques to reduce atomicity hardware support:** We propose a variety of hardware and software techniques to reduce the hardware atomicity support cost, namely register checkpointing and write logging. We implement and evaluate the proposed techniques in both the JVM and a simulator. We investigate techniques such as register calling convention utilization (SW), register dirtiness analysis (SW), and instruction window buffering (HW) to reduce register checkpointing. We also investigate techniques such as stack write logging removal (SW), heap write logging removal (SW), and store buffering (HW) to reduce write logging. We also look at a software technique called region shrinking (SW) to further reduce write logging. With all these techniques, we demonstrate that we can reduce 98% of register checkpointing and 94% of write logging, which translate to

the preservation of 95% of performance improvement from speculative optimizations.

1.5 Thesis Organization

This thesis is organized as follows: Chapter 2 presents a detailed overview of state-of-the-art Java Virtual Machines and atomicity supporting hardware research. Chapter 3 discusses the Java programming language's precise exception model and how it might impede the optimized code performance. Chapter 4 presents our solution -- a speculative execution model with hardware atomicity support and how it might help with speculative optimization design. Chapter 5 describes the speculative null pointer removal and bounds check removal algorithms to relax Java's precise exception model's optimization constraints with the proposed speculative exception model in Chapter 4. Chapter 6 discusses a variety of hardware and software techniques to lower hardware atomicity support cost and preserve speculatively optimized code performance. Chapter 7 presents an empirical study on both native machines and simulators to show speculative optimization's performance and the cost reduction techniques' effectiveness. Finally, Chapter 8 concludes this thesis and presents additional avenues for future research.

Chapter 2

Related Work

This chapter is divided into three sections. The first section surveys the recent progress in dynamic optimization, especially the Java Virtual Machines (JVMs) for the Java programming language. We focus on large software systems developed for dynamic optimization and their features. The second section presents the latest progress in atomic systems including Hardware Transactional Memory (HTM), Software Transactional Memory (STM), Hybrid Transactional Memory (Hybrid TM), and other atomicity supporting systems. The last section touches upon some of latest hardware-software co-designs and discusses their design philosophies.

2.1 Dynamic Optimization

2.1.1 Dynamic Optimization System Overview

There have been many dynamic optimization systems developed since the 1990s. Overall, dynamic optimization systems consist of software and hardware dynamic optimization systems. Software dynamic optimization systems include low-level native-to-native dynamic optimization systems such as HP's Dynamo [10] and Transmeta's Code Morphing Software designed for its Crusoe processors [29] and high-level programming-language-based dynamic optimization systems developed for programming languages such as C [34][9][30][64][78], Self [20][21], C#[67], and Java [5][75][56][94][54][11][88].

The low-level native-to-native dynamic optimization systems arise due to the need for binary translation from one processor ISA to another processor ISA. Computer sys-

tems advance at a fast pace. When the ISA is re-designed for a new generation of processors, it needs to provide a guarantee to run large amounts of legacy software in which billions of dollars has been invested. The low-level native-to-native dynamic optimization systems can address this problem easily. HP's Dynamo was a research prototype and it focuses on the translation of the same ISA (from HP PA-8000 to HP PA-8000) to investigate the ISA translation possibility and also the acceleration opportunity. Transmeta's CMS translates x86 binaries to its own VLIW ISA to achieve performance improvement and reduce power consumption.

High-level programming-language-based dynamic optimization systems focus on better compilation of the code written in a specific programming language. The dynamic optimization systems can gather runtime information (via profiling or not) and employ such knowledge to better compile the code and achieve better performance. In such systems, compile/re-compile time adds to the application execution time so that lightweight but efficient optimizations need to be used. Optimizations are usually introduced step by step as hot functions and code regions are identified during program execution. More advanced optimizations are applied to small regions of hot code which typically dominate the execution time of most applications.

Hardware dynamic optimization systems are an emerging trend compared with their software counterparts. One such system is Illinois' rePlay framework [76]. It relies on branch promotion to construct frames including multiple basic blocks when x86 binaries are being executed by a processor. It then applies dynamic optimizations to the code within each constructed frame. In a frame, every branch is converted to an assertion and the assertion is verified during the frame execution. By doing this, the basic block size is

extended and optimizers can easily extract more optimizations from the enlarged basic blocks.

2.1.2 Java Virtual Machines

There have been numerous Java Virtual Machines developed in both industry and academia since the dawn of the Java programming Language. Based on the purpose of the developed JVM, we classify all JVMs into two categories -- research JVMs and production JVMs. Research JVMs are developed for research purposes and their code is easily accessible. Production JVMs are developed for business applications and their code is typically strictly guarded (although until recently, Sun Microsystems open-sourced its Hotspot JVM code). We briefly describe two Java Virtual Machines -- IBM's Jikes Research Virtual Machine and Sun Microsystem's Hotspot in the following two subsections.

2.1.2.1 Jikes RVM

Jikes is a research JVM developed by IBM T. J. Watson research lab. It is an all compilation JVM as it does not have any interpreter. Jikes RVM supports two ISAs -- PowerPC and x86. It includes a baseline compiler (the simplest compiler that compiles the code without doing much optimization) and an optimizing compiler. The optimizing compiler has three optimization levels: opt0, opt1, opt2. The RVM is fully configurable and the user can specify how many compilation/optimization levels the RVM should support. Typically, a method is compiled first at either baseline or opt0. When a method is detected hot, it can be gradually compiled to opt1 and then opt2.

The optimizing compiler supports three levels of intermediate representations -- high-level IR (HIR), low-level IR (LIR), and machine-level (MIR). Optimizations are

made in each intermediate representation. HIR is quite similar to the Java bytecode; LIR introduces details about Jikes specific information such as the runtime information and the object layout; MIR introduces details about the target machine information. In Table 2-1, the optimizations supported by each optimization level are listed. Further details about the Jikes RVM optimization compiler can be found in [16][7][46].

Table 2-1: Jikes RVM optimizing compiler optimizations.

opt0	On-the-fly constant and type propagation, constant folding, branch optimizations, field analysis, unreachable code elimination, trivial inlining Instruction selection Register allocation and coalescing
opt1	Full inlining (including preexistence and other speculative inlining) Static splitting, tail recursion elimination Local redundancy elimination (common subexpression elimination, loads, checks) Flow-Insensitive: constant, copy, type propagation, sync removal, scalar replacement of aggregates, code reordering, dead code elimination
opt2	Loop normalization and unrolling Scalar SSA: dataflow, global value numbers, global common subexpression elimination, redundant conditional branch elimination Heap array SSA: load/store elimination, global code placement

2.1.2.2 Hotspot

Hotspot is a production JVM developed by Sun Microsystems. There are two versions of Hotspot -- one for server applications and the other for desktop/client applications. The server Hotspot compiler adopts more aggressive but efficient optimizations since the performance of server applications is often critical and servers can afford to spend more time on aggressive optimizations to achieve overall performance improvement in long-running applications. We only describe the server Hotspot compiler here.

The Hotspot JVM has an interpreter and methods are initially interpreted. When a method is detected hot, optimization (re-compilation) is applied to this method. The hotspot compiler uses Static Single Assignment (SSA) [25] as its intermediate representation upon which optimizations are based. According to [75], the compiler implements

optimizations such as dead code elimination, loop invariant hoisting, common subexpression elimination, constant propagation, null check and bounds check elimination, graph coloring register allocation, instruction scheduling, advanced inlining, instruction selection, global code motion, and peephole optimizations. The Hotspot server compiler also implements deoptimization, which creates safe points in the code to allow aggressive optimization of the code and rollback to the safe points in case any assumptions are violated.

2.1.3 Bounds Check Optimizations

ABCD [15] proposes the concept of eliminating bounds checks on demand for very hot functions in Java dynamic optimizations. It also gives an algorithm that can eliminate array bounds checks whose index values can be related to array lengths. However, only small percentages of array bounds checks can be related to array lengths that are constants in order for the algorithm to be safely applied. In static compilers, researchers have proposed three techniques to remove bounds checks in programs: theorem-proving style techniques [95][96][72], value-range analysis [45][77][84], and partial redundancy elimination[63][37][38][8][59]. These techniques are too heavyweight for a dynamic environment to implement.

2.1.4 Optimizations and Java's Precise Exception Model

There has been little research on the removal of the precise exception model's constraints on Java code optimizations. In [36], researchers propose to use software checks and recovery handlers to allow speculative code motion and significant speedups were reported on two very small kernel benchmarks due to removed precise exception constraints and the resulting loop transformations.

2.2.1 Transactional Memory

Transactions were originally a concept in the database community [35] to ease the database implementation of concurrent queries and operations. In databases, transactions need to be "ACID": A stands for atomic which means that a transaction is executed in an all-or-nothing way; C stands for consistency which means that the database remains in a consistent state before the start of a transaction and after the finish of a transaction; I stands for isolation which means that a transaction appears isolated to other ones; D stands for durability which means that a transaction cannot be undone once it is successful and notified to the user.

The concept of transaction was later adopted by the communities of computer architecture and compilers. The concept evolved into transactional memory (TM) which can be generally divided into three categories -- hardware transactional memory (HTM), software transactional memory (STM), and hybrid transactional memory (HybridTM). We will describe the three categories in detail in the following three subsections.

2.2.1.1 Hardware Transactional Memory (HTM)

Knight [58] first proposed hardware transactional memory (HTM) in 1986. In his paper, he described a transaction-based architecture for the execution of programs written in the LISP programming language. Later IBM built a transaction-like computer system called "801 Storage" [22]. In "801 Storage", the computer system implements a large virtual storage space for both temporary and permanent data and the access to the storage space is similar to that to a database storage (transactions). Herlihy and Moss [48] revisited the HTM concept and proposed an architecture to execute transaction-based pro-

grams. Their proposal is to remove locks in concurrent programming to ease parallel application development and improve its performance. In the proposed architecture, the programmer can define customized read-write-modify operations applied to multiple, independent words of memory. The correctness of this approach can be easily verified with the verification of the multiprocessor coherence protocols. This work is widely accepted as the foundation of modern HTM research in the multiprocessor research community.

The Stanford TCC paper [39] started the latest wave of HTM research. TCC stands for transactional memory coherence and consistency. In their proposed architecture, all the writes in a transaction will be put into one single packet and broadcast via the interconnection network to other processors for conflict detection. Conflicts are detected after a transaction finishes execution and rollbacks are performed by hardware if conflicts are detected. Since the seminal TCC paper, the Stanford TCC group has been at the forefront of modern HTM research. Their papers [24][17] give details about transaction opportunities in real workloads. Their work in [66] presents a comprehensive transactional memory ISA and describes in detail the architectural semantics for HTMs. In [18], they give an HTM based programming language called Atomos. Recently, they implemented a real chip [74] that was based on their HTM proposals. They also made an attempt to improve the debugging support for HTM programming in [23].

The Wisconsin Multifacet group started their HTM research with a seminal paper LogTM [69]. LogTM employs logging instead of buffering as the main methodology to temporarily retain the initial state during the execution of a transaction. By using logging, it makes in-place updates and saves the old value if the address is modified for the first

time. LogTM makes the common case (commits) faster while relying on software to handle rare cases (rollbacks). After the LogTM paper, the Multifacet group has published a series of papers on Log based HTM. In [70], it deals with how to support unlimited levels of nested transactions. Nested transactions can be handled in two ways -- closed nesting and open nesting. Closed nesting extends the isolation of a child transaction until the commit of the parent transaction. Open nesting ends the isolation of a child transaction when the child transaction commits. In [97][87], they propose to implement hardware transaction signatures to speed up conflict detection. Their work in [14] categorizes HTM performance issues into seven classifications and attempts to find solutions to performance pathologies.

Other notable HTM work includes VTM [83] and UTM/LTM [6]. VTM gives a proposal to make HTM transparent to the user and thus platform independent. UTM/LTM address the issues of unbounded transaction sizes.

2.2.1.2 Software Transactional Memory (STM)

One big disadvantage of STM compared with HTM is that it is relatively slow since software needs to buffer/log the initial state of a transaction. At the sacrifice of speed, it achieves flexibility. STM is easy to build and prototype and can suit many different hardware platforms.

The forefront of the STM research is in industrial research labs, namely Intel and Microsoft while the HTM research is mostly done in academia. At Intel [85][2], researchers implemented a STM prototype in a JIT environment [1]. [85] focuses on the implementation and correctness of their prototype while [2] discusses a variety of ways to reduce the cost of software constructed transactions. The optimizations they propose in [2]

include conventional code optimizations such as redundancy elimination, dead code elimination, inlining and loop transformations for code introduced by software transaction constructions. They also use some simple global optimizations to reduce transaction overhead. For example, they remove transaction support for reads from fields defined with the Java keyword "final" ("final" defines a runtime constant) and for reads/writes to transaction-local objects (they will not be seen outside a transaction). In [86], they further looked into ISA extension and architectural support to reduce the overhead of STM.

Microsoft's STM efforts started from Tim Harris' graduate student research. An attempt to add lightweight transactions to the Java programming language was made in [42]. A set of STM supporting operations were added to Java and modifications were made in both the source-to-bytecode and the bytecode-to-native compilers to recognize the STM data structures and operations. In [43], they look at an addition of a new concurrency model based on STM to the Haskell programming language. New modular forms such as "blocking" and "choice" are added. In [44], an attempt to optimize STM was made. In this work, a new "direct access" implementation was introduced to reduce the logging cost. Some optimizations were made to reduce the STM operation overhead. Duplicated logging records were also removed with runtime filtering. An overview of Harris' STM work can be found in [41].

2.2.1.3 Hybrid Transactional Memory (HybridTM)

The work distinctly labelled as HybridTM is far less than that in both HTM and STM. HTM needs some software support for correctness concerns and STM may need some hardware support for speed concerns. Therefore, HybridTM has some overlap with both HTM and STM.

Intel [60] first attempted an implementation of HybridTM. In their implementation, hardware managed the transaction if buffering did not exceed the hardware resource limit; otherwise, hardware fell back on software to gracefully handle large transactions. Sun Microsystems [28] also made an attempt at a prototype of HybridTM system to demonstrate its practicality. Stanford's SigTM [68] used hardware signatures to track a transaction's read-set and write-set but relied on software for all other transactional functionality including data versioning.

2.2.2 Illinois' Hardware Atomicity Work

Illinois published a paper [73] on hardware atomicity and its impact on Java compiler optimizations at the same time as our paper [93]. The Illinois work is very similar to ours. Instead of designing new algorithms to relax Java's precise exception constraints, their work forms atomic regions with cold paths converted to assertions, which is equivalent to increasing the atomic region size. The enlarged atomic regions can take advantage of existing compiler optimizations to achieve performance improvement. Their work is joint work with Intel and uses a commercial JVM [40]. Their benchmark suite is DaCapo [13].

2.2.3 Other Atomic Systems

Transmeta's Crusoe processor [29] exposes its hardware atomicity to its code morphing software (CMS) to help relax the optimization constraints imposed by x86 precise exceptions during the translation of the x86 binaries to its own VLIW binaries. This is the first working product that exposes hardware atomicity to software to help improve program performance.

Checkpoint processors are not strict atomic systems. They extend modern processors' speculative execution capability to recover states at a coarse level [4][27][65]. Modern processors implement a variety of speculation techniques and speculation is mostly correct. Checkpoint processors take a step further and do a better job to optimize recovery information management so that speculation can be performed on a larger scale.

Another type of system that is not atomic but similar to transactional memory in terms of the design goal is Rajwar's lock elision hardware [81][82]. This work also realizes the difficulty of writing scalable parallel programs with locks and the resulting performance losses. They propose a microarchitectural technique called Speculative Lock Elision (SLE) to remove unnecessary, performance-limiting locks in multithreaded programs.

2.3 Hardware Software Co-Design

The latest microprocessors have about one billion transistors on the chip and soon we will see tens of billions of transistors in microprocessors. The enormous amount of transistors integrated in a small chip has led to fast-growing power consumption. The ever shrinking semiconductor manufacturing technology makes the power problem even worse. With power as a big challenge, software cannot simply rely on faster clocked microprocessors to maintain the traditional performance improvement software has been enjoying. Therefore, the design has increasingly become a co-operative effort between hardware and software. Lately, hardware software co-design has become a hot trend.

The co-designed virtual machine project [49][50][51][52] at the University of Wisconsin - Madison combines hardware's performance innovations and software compatibly

and represents a future direction in which microprocessor design may be heading. In their design, the hardware fuses simple RISC operation into macro-ops to provide fast performance primitives. A binary translator automatically translates binaries in the original ISA to the ISA that reflects the hardware innovation. The design methodology is quite similar to the Intel Itanium design methodology in which old x86 binaries needed to be supported in the Itanium processors.

Chapter 3

Java's Precise Exception Model

This chapter starts by discussing the relationships of exceptions and general computer systems, proceeds to explain how the Java programming language handles its exceptions, then presents an upperbound empirical study to illustrate the potential performance losses due to Java's precise exception handling, and finally points out the possible performance impact of a safe programming language trend.

3.1 Exceptions and Computer Systems in General

Exceptions occur rarely but sometimes unavoidably in computer systems. There are exceptions in microprocessors, programming languages/compilers, operating systems, and databases, etc. Different systems may have different solutions to exception handling but they all need to address two questions. First, should the system handle an exception or simply ignore it? Second, should exceptions be handled in the order they occur and the exact system state be maintained at the occurrence of an exception? A safer system typically handles exceptions more conservatively to foster a better understanding (reasoning) of the system and reduce the implementation and maintenance cost.

3.2 Exceptions and Java

In a programming language, exceptions are violations of the semantic constraints. In contrast with normal executions, exceptions occur rarely but often surprisingly in a well designed software system. There are many reasons causing exceptions. One main reason

is the violation of runtime checks specified by a programming language. Other reasons could include resource limitation (e.g. out of memory), internal errors of a runtime environment, explicitly thrown exceptions from a programmer, etc.

Some programming languages such as C [57] choose to ignore exceptions and abruptly terminate the program when an exception is encountered. Java [32] explicitly deals with exceptions as a safe programming language. An exception can be caught and handled using a catch clause. If the catch clause cannot handle the caught exception, it can re-throw the same or a different exception along the call stack. If the exception cannot be handled eventually, an error message along with a stack dump will be given by the JVM.

The explicit exception handling in Java has many benefits. First, it separates normal code from exception handling code and makes the program easier to write and understand. Second, it can reduce surprises in program execution by handling expected exceptions. Third, it can help a programmer reason about a program. Fourth, it can help with the debugging process by providing exception handling and diagnosis information.

3.3 Java's Precise Exception Model

Java specifies that exceptions are precise. When an exception occurs and execution is transferred from the normal execution path to the exception path, all the statements before the excepting one should appear finished and their effect should have been committed to the system; all the statements after the excepting one should appear unexecuted and they should not affect the visible system state. Any code optimization should not make preceding statements appear unexecuted or following ones appear finished.

Precise exceptions have many advantages. By preventing code optimizations from

changing the order of potentially thrown exceptions, it helps programmers reason about the order of all possible excepting paths and makes debugging much easier. It also makes exception handling itself much easier since the number and types of possible exceptions at a particular program point are known at the source level and no more exceptions can be introduced by any code optimization.

3.4 How Can Precise Exceptions Hurt Performance

However, precise exceptions come with a performance hit via the constraints imposed on optimizations that may affect potentially excepting instructions (PEIs). A PEI is an instruction that usually executes normally but might throw an exception and cause an execution switch from a normal path to an excepting one. PEIs that need to follow the precise exception model are PEIs that throw runtime and checked exceptions in the Java programming language. PEIs that throw asynchronous exceptions do not need to follow the precise exception model.

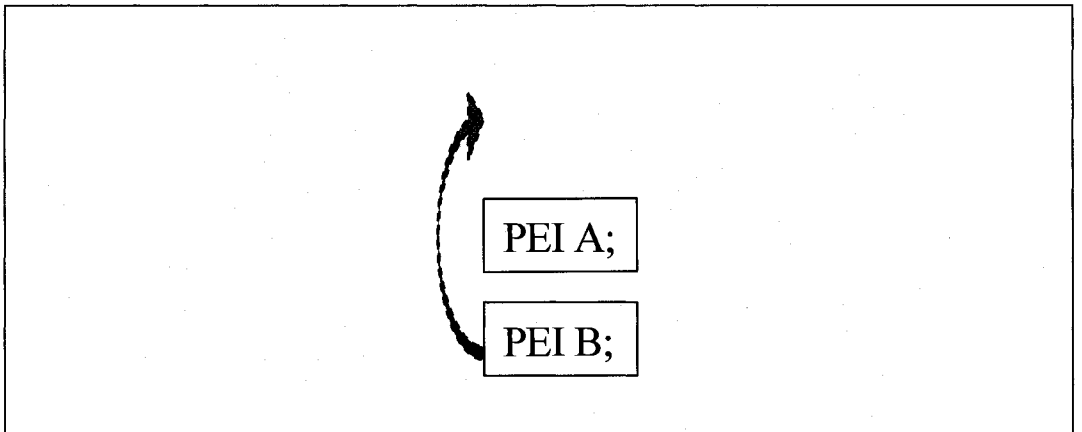


FIGURE 3-1. PEI ordering example. PEI B is immediately following PEI A.

The precise exception model imposes two constraints on code optimizations. The first one is about the ordering of PEIs. Code optimization can not freely move one PEI

before another since this could change the order of potentially thrown exceptions. As shown in Figure 3-1, code optimization can not move PEI B before PEI A unless it can statically prove that either or both PEIs will not throw an exception at runtime. For example, loop invariant code motion, a common code optimization technique, cannot be freely applied to PEIs inside a loop since it could potentially break the precise exception model by moving a loop-invariant PEI out of the loop.

The second constraint is about the program state visible at the entrance of an exception handler. The program state includes all variables or memory content that could be observed at a particular program execution point. Precise exceptions forbid any code optimization from changing the program state visible at the entrance of an exception handler. This constraint can make some seemingly easy optimizations difficult. In Figure 3-2,

```
try {
  dead code;
}
catch (...) {
}
```

FIGURE 3-2. Exception handler entrance program state example. Here dead code assigns some variables that will not be used in the normal execution flow; however, these variables could be observed in the catch clause (shown here) or even some other catch clauses (not shown here) in the call chain

it shows why dead code removal could be difficult for Java. In the figure, there is some dead code in the try clause and it is easy to prove that the code is dead in the normal execution paths. However, there are still catch clauses (visible or non-visible in the figure) that might observe the values assigned by the dead code. Therefore, it is not safe to simply

delete the dead code here. To make the matter worse, the catch clause could be anywhere in the call chain and it is very difficult to completely analyze all the catch clauses to prove that the code is "strictly" dead even in the excepting paths.

The PEIs that affect optimization and performance the most are check instructions that are introduced by Java to specifically validate certain conditions. Typical check instructions include null pointer checks (against null pointer dereference), array bounds checks (against array out of bounds memory accesses), zero checks (against division by zero), store checks (against an incompatible object reference saved in a reference array), and checkcast (against incompatible type cast). Among these check instructions, null pointer checks and array bound checks are the majority and affect the performance the most.

3.5 An Example: Jikes RVM's Performance

A high-performance research virtual machine such as Jikes RVM [5] only implements some limited optimizations to remove redundant null pointer checks and array bound checks and experiences visible slowdowns due to unremoved checks in the presence of Java's precise exception model.

The Jikes optimizing compiler's optimization flow is shown in Figure 3-3. The compiler converts the stack based Java byte code into a register based intermediate representation (IR). There are three levels of IR - high-level IR (HIR), low-level IR (LIR) and machine-level IR (MIR). The lower two levels implement more detailed operations. LIR implements unique operations in Jikes and MIR includes machine specific operations. HIR and LIR have tens of optimization phases and MIR has several optimization phases.

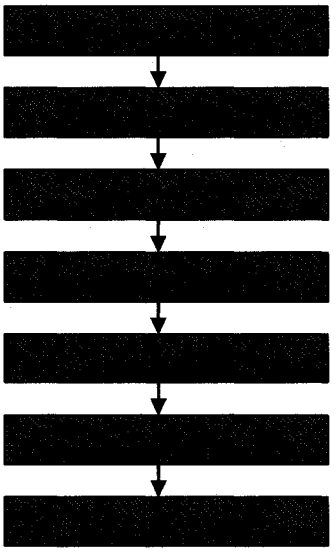


FIGURE 3-3. The optimization flow of Jikes RVM's optimizing compiler.

In HIR generation, check instructions are separately generated from their associated instructions. Their ordering is strictly maintained and it creates performance constraints for later optimization stages. Some checks can be statically proven by the compiler to be redundant and thus removed. Many checks simply propagate through the following optimization phases. They increase the IR size and thus the compile time. In one MIR optimization phase (NullCheckCombining), null checks are combined with loads/stores if they are in the same basic block and there are no PEIs between them. In the latest versions of Jikes, there are only limited local bounds check eliminations.

Next, an upperbound study using Jikes is presented to show the performance losses and the compile time increase due to the precise handling of array bounds checks and null pointer checks. The experimental methodology is described in Section 7.1.1.

The ideal case performance improvement due to bounds check elimination at the

beginning of the optimization flow is shown in Figure 3-4. Bounds checks impede perfor- 29

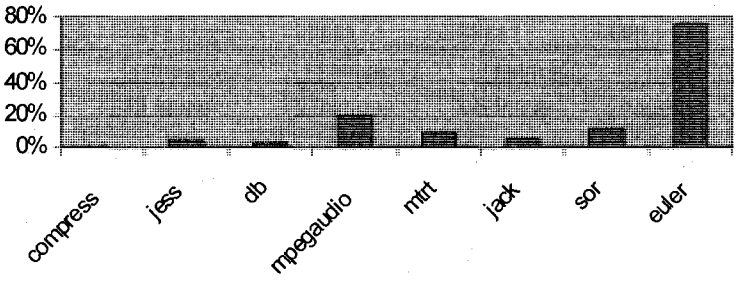


FIGURE 3-4. Performance gain due to avoiding bound checks. Simply removing the generation of bound checks at the beginning of the optimization flow when Java byte code is translated to Jikes HIR can cause substantial performance increases for some of the workloads we use. (one run with one input)

mance even more than null checks. We also tried to vary the mix of optimization phases in the optimization flow by the deletion of optimization phases in Static Single Assignment (SSA) and found that the performance losses due to bounds checks for our benchmarks fluctuated compared with the case where SSA optimizations existed. The performance impact for mpegaudio, mtrt, and euler lowered by about 30 per cent while they remained within 10 per cent for other benchmarks.

The presence of PEIs impedes other, seemingly unrelated optimizations and affects the overall effectiveness of the optimizing compiler. We modified Jikes to eliminate null checks at the beginning of the optimization flow instead of at the later NullCheckCombining stage. The performance improvement of early null check elimination is shown in Figure 3-5. The unmodified Jikes can combine about 88% of null checks with loads/stores at the NullCheckCombining stage in our benchmarks. In the baseline we went further to delete the remaining 12% after the NullCheckCombining stage (This deletion does not lead to any performance improvement in our benchmarks). However, the baseline still suf-

fers performance losses compared with null check elimination at the beginning of the opti- 30
mization flow. The losses are quite significant in some benchmarks. We tried to vary the

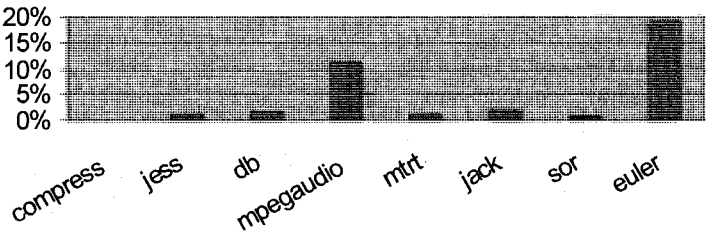


FIGURE 3-5. Performance gain due to avoiding null pointer checks. Simply removing the generation of null pointer checks at the beginning of the optimization flow when Java byte code is translated to Jikes HIR can cause substantial performance increases for some of the workloads we use.

optimization phases in the optimizing flow, e.g. by deleting optimizations in SSA (disabling optimizations in SSA could cause slowdowns of about 25%, 10%, 5%, and 5% for mpegaudio, compress, sor, and euler respectively but have almost negligible impact on other benchmarks), and found that the performance loss induced by null checks almost disappeared for mpegaudio, was slightly lowered for euler and sor, and was greatly lowered for other benchmarks. We conclude that certain optimization opportunities and the overall effectiveness of the optimizing compiler are hindered by the presence of null checks.

Figure 3-6 shows the compile time increase due to null checks and bounds checks; the IR size increase induced by the checks can substantially increase compilation overhead and slow down execution. In the baseline we simply delete both checks at the beginning of the optimization flow and thus there is no compilation overhead from these two checks.

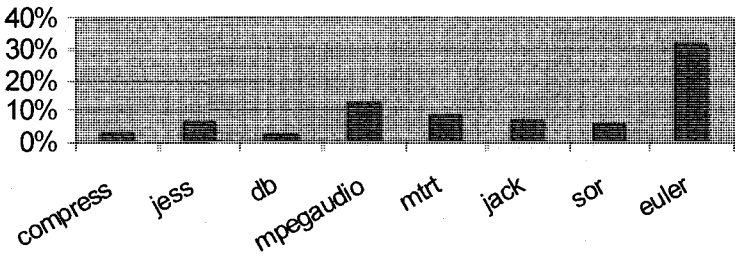


FIGURE 3-6. Compile time increase due to check instructions. The generation of null pointer checks and bound checks can cause substantial compile time inflation. Meanwhile, the compiled code size similarly bloats but in a less dramatic fashion than the compile time

3.6 Safe Programming Language Trend and its Impact

The precise exception model endorsed by the Java programming language simply reflects a general trend in programming languages -- safety. With more safety features introduced into programming languages, it is inevitable that some of these features might present a challenge to efficient code optimization. Namely, memory management (garbage collection), security, bug detection etc., could all affect the performance of programs written in future programming languages designed with safety in mind.

3.7 Summary

Java's precise exception model has caused noticeable overhead to its code optimizations and application performance degradation. With the increasing amount of on-chip transistors, we could afford to apply a co-design approach and use hardware to help the JVM to relax Java's precise exception constraints and improve Java program performance. In the next few chapters, I will present a speculative execution model and some lightweight speculative optimizations that can help improve Java performance. I will also dis-

cuss some techniques that can help minimize the extra hardware cost due to the maintenance of atomic hardware regions.

Chapter 4

A New Speculation Model for Java

This chapter proposes a speculative execution model to relax the optimization constraints imposed by Java's precise exceptions. The model relies on a hardware-software hybrid approach to achieve performance improvement, compile time reduction, code size reduction, and code optimization simplification. While the proposed model focuses on the relaxation of constraints due to precise exceptions, it could be used to relax any optimization constraints imposed by other safety features on a programming language. This chapter explains the hardware atomicity that the speculative execution model relies on and the needed support for hardware atomicity. It also discusses several approaches for atomic region formation. With the help of hardware atomic regions, speculative optimizations can be safely performed. Software does not need to maintain the speculative states within atomic regions since hardware can roll back atomic regions if necessary.

4.1 A New Speculative Execution Model

Our new speculative execution model, as shown in Figure 4-1, relies on the coordination of both hardware and software to enable aggressive, speculative yet simple code optimizations. In this model, software (the JVM) identifies atomic regions. An atomic region is executed in an all-or-nothing way in terms of the observable processor/system state. During the execution of an atomic region, hardware constantly monitors the region to ensure its validity. Therefore, we also call an atomic region an atomic guarded region, where "guarded" means that the region is monitored and guarded by hardware. Hardware

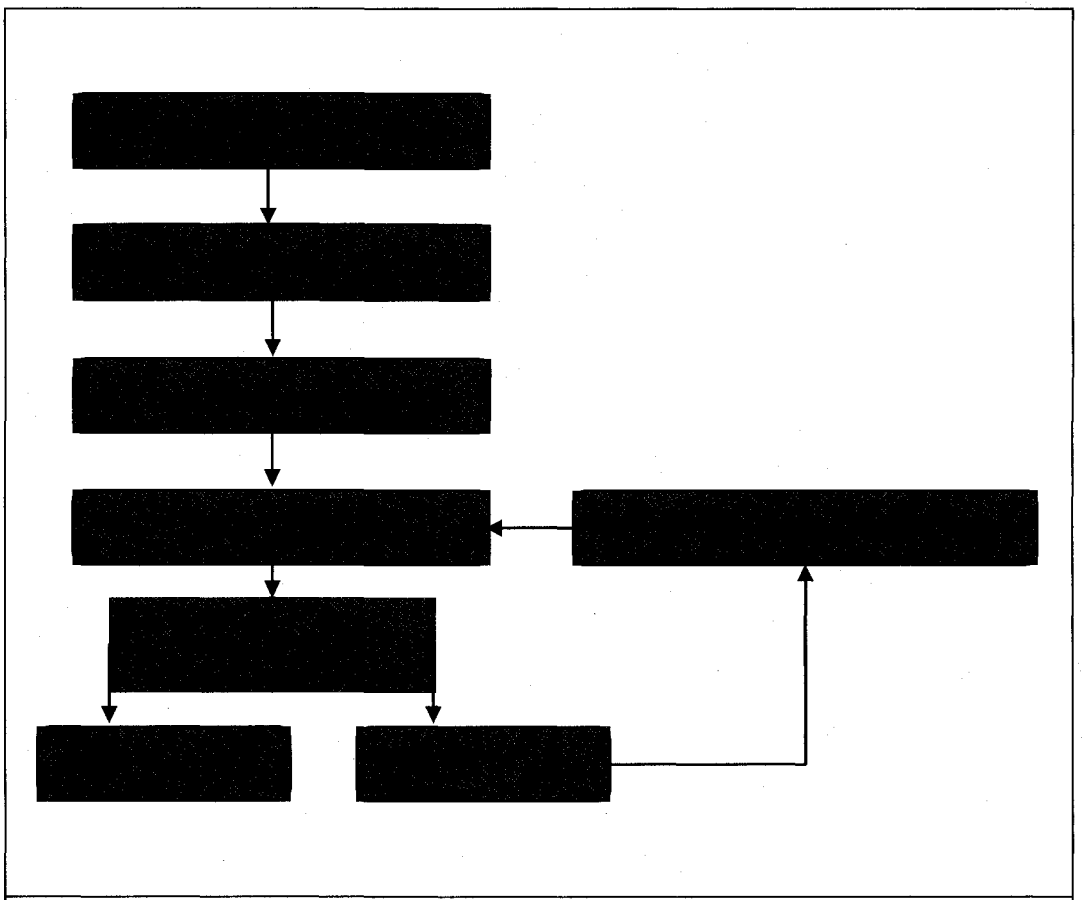


FIGURE 4-1. New speculative execution model. Here guarded region are atomic regions. We use atomic regions and guarded region interchangeably and they have the same meaning in this thesis

and software form a contract that needs to be honored in the whole execution of an atomic region. A contract could be that a certain condition needs to hold. After the region is identified and a contract is formed, the JVM can apply aggressive, speculative yet simple code optimizations that require less compile time, reduces the code size and speed up an application. During the execution of an atomic region, hardware constantly monitors the condition of the contract. If the contract is valid in the end of the region execution, the region can be simply committed. If the contract is violated during the region execution, hardware can roll back the executed part of the region and revert the machine state to what it was before the region starts execution. Software then applies conservative optimizations for

the same region and hardware can reexecute this conservatively compiled code.

The execution model utilizes the ample onchip hardware resource in future billion-transistor Chip Multiprocessors (CMPs) and harnesses it to provide hardware atomicity. Software does what is easy for itself: identify code segments that can be atomically executed, and apply flexible and effective code optimization. Hardware provides atomicity to software and can roll back a whole atomic region if needed. This division of work allows hardware and software do work that they are each good at yet at the same unites the two via a contract that both need to honor.

4.2 Hardware Atomicity

Hardware atomicity is a key concept in our speculative execution model and it provides the foundation for software speculation. Pure software speculation is possible but the supporting cost in software often outweighs the performance gain and thus is often a less desirable approach.

Hardware atomicity means that hardware can execute a sequence of code in an all-or-nothing fashion. The machine state change will not be visible until the whole sequence of code within an atomic region finishes execution and commits. The atomicity granularity can be easily adaptable and cover code sequences with hundreds, thousands, or even more instructions.

Hardware atomicity needs to be provided to software in a flexible way so that software can form atomic regions freely. Atomic region based code execution can be intermixed with regular code execution; the whole program can also be broken down into continuous atomic regions. When code is executed normally instead of in atomic regions,

hardware can simply remove all the extra operations performed to ensure atomicity correctness and thus remove the overhead due to hardware atomicity support.

4.3 Hardware Support for Atomicity

In order for hardware to support atomicity, the machine state needs to be able to recover to the point right before the atomic region starting point. The machine state consists of architecture registers and memory content. Architecture registers need to be restored to the original values and the memory content modifications need to be undone when an atomic region rollback occurs.

There are many ways of implementing register value restoration upon a rollback. The most straightforward and easy-to-implement way is register checkpointing, especially batch based checkpointing upon the entrance of an atomic region. In this kind of register checkpointing, hardware has two copies of register files -- the architected register file (ARF) and the checkpointed register file (CRF), as shown in Figure 4-2. Upon the entrance of an atomic region, hardware does a wholesale register file copy from the ARF to the CRF. After register checkpointing, the architected registers can be freely modified within an atomic region. In case of a recovery, the checkpointed register values can be copied from the CRF to the ARF.

In general, there are two ways of supporting memory content reversal -- write logging and write buffering. In write logging, every write checks if the old value at the to-be-modified address has already been logged. If no, the write needs to save a log to keep the old value somewhere. If yes, the write can simply modify the memory content. Write buffering simply buffers the new values for each modified address. A smart write buffering

scheme can remove redundant values at one memory address and always keep the most up-to-date value. Write logging and write buffering are quite similar in implementation complexity. Our work simply uses write logging to support memory content reversal. As

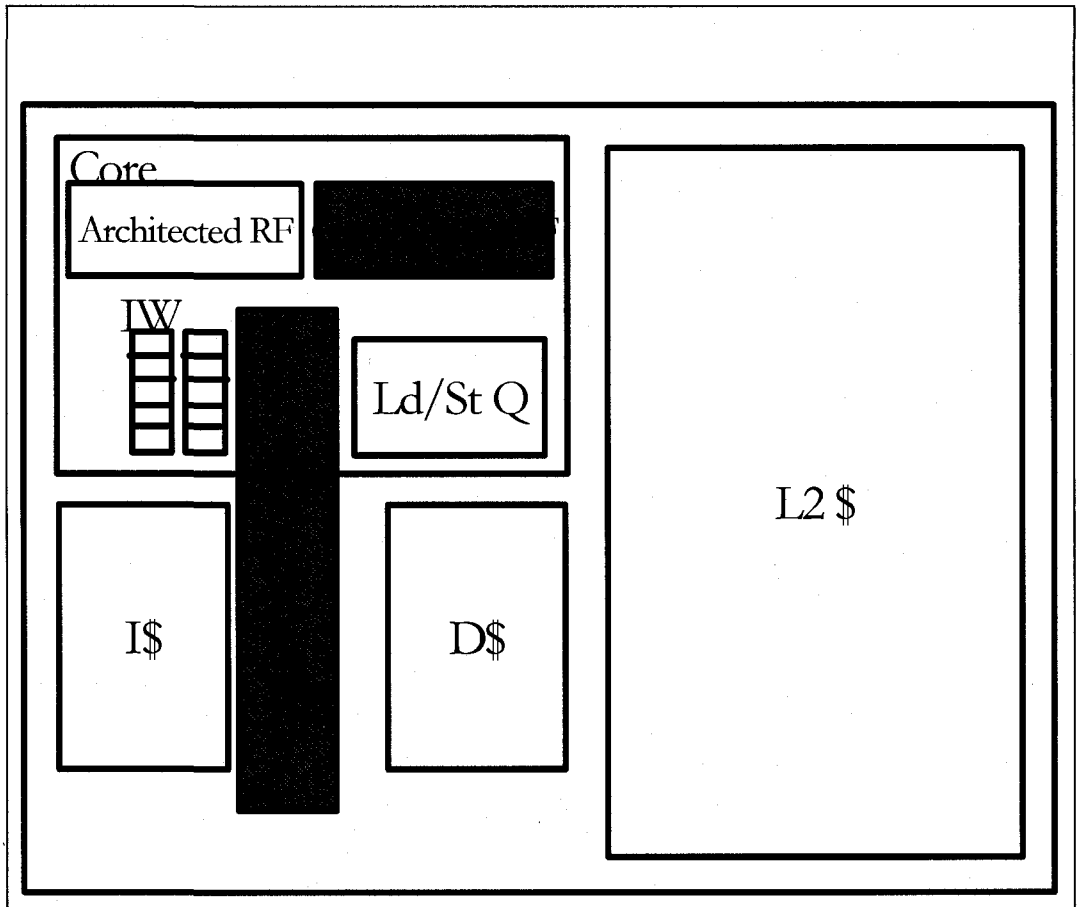


FIGURE 4-2. Hardware atomicity support. Added hardware structures are the checkpointed register file (CRF) for register checkpointing and the write logging buffer (WLB) for write logging

shown in Figure 4-2, there could be some exclusive hardware resource called the write logging buffer (WLB), which is used to keep logs. A write checks if an existing log already exists in the WLB first; if not, a new log needs to be written into the WLB before the write can proceed. When the WLB overflows, the overflowing content in the WLB needs to be saved to the DRAM (Dynamic Random Access Memory). An alternative solu-

tion is to use a portion of the data cache and the L2 cache to save logs. In this way, caches will have both data and logs and there might be conflict misses caused by that coexistence. In order to deal with unbounded atomic regions, a certain range of physical memory addresses and some disk space could be reserved to save logs. When overflow occurs, whether it is an overflowing of WLB or caches, the logs are saved to the reserved physical memory range and even down to the reserved disk space in case of a shortage in physical memory space.

Besides the register and memory hardware support, there needs to be some communication interface to let software signal the entrance and the exit of an atomic region. This is done via the instruction set architecture (ISA). The ISA needs to provide at least two instructions -- region start and region commit, which clearly define atomic region boundaries. Region start signals the entrance of an atomic region and region commit indicates the end of an atomic region. Depending on the hardware implementation and the ISA sophistication, there could be more instructions devoted to hardware atomicity. Two instructions called conditional region start/commit could also be included in the ISA. Conditional means a region is committed and a new region is started based on a certain condition. The condition could be the remaining onchip logging or buffering resources. If hardware decides that there are still plenty of resources left, it could opt for not committing the current region and splice the next region with the current region.

The ISA also needs to provide a mechanism for software to indicate what registers need to be checkpointed if register checkpointing is performed in a distributed way and certain register checkpointing can be avoided. This could be done with an extra bit in the instruction encoding to indicate if the instruction's destination register needs to be check-

pointed. It could also be done by the addition of another instruction called checkpointing which could follow the original instruction to indicate if its destination register needs checkpointing. The ISA might also need a register marking instruction to enable/disable the checkpointing of a set of architected registers.

Similarly, instruction encoding could use an extra bit to indicate if a write needs to have accompanying logging operations. An extra instruction, write logging, could also be appended to a write to perform logging for this write separately. The addition of write logging in the binary gives software freedom to optimize away unnecessary logging and reduce logging cost.

We will describe our techniques to remove unnecessary register checkpointing and write logging in Chapter 6.

4.4 Atomic Region Placement

In our speculative execution model, regions are formed by the JVM (software). Region placement consists of region insertion time, region insertion location, and region size. Ideally, regions need to be identified early in the optimization flow so that later optimizations can take advantage of the formation of the regions. Regions could also be identified in the basic optimization pass or the profiling stage; more advanced re-optimizations could optimize code with region knowledge later on.

Regions could be inserted continuously or intermixed with regular code execution. The intermixing of regions and regular code is sometimes unavoidable if irreversible operations such as I/O are performed since such actions cannot be placed in atomic regions. Regions could be formed within or across a function. Profile information can identify the

insertion points of regions and basic blocks that could fall into the same region. A region could be a superblock or several superblocks.

The size of a region determines the pressure on hardware atomicity support and the opportunity for speculative software optimizations. If a region is big, it may require a lot of hardware resources to support memory content reversal and register recovery. However, the speculative optimization opportunities might be abundant. If a region is small, there might not be many speculative optimization opportunities, but fewer hardware resource will be needed. For our speculative execution to achieve positive performance benefit, the optimization benefits need to outweigh the hardware cost; therefore, the right region size is an important design decision.

Region placement also needs to address garbage collection. Garbage collection is usually performed in two ways. First, it can be triggered when there is a shortage of free memory. Second, it can be directly inserted by either a JVM or a programmer. Memory-shortage triggered garbage collection can be viewed as an exception that will lead to a region rollback. It should rarely occur, especially when an explicit free memory check and possible garbage collection are inserted at the beginning or the end of region execution (such a check and possible garbage collection are performed upon the entrance and the exit of a function in the original Jikes implementation). The programmer induced garbage collection, which is a very rare event, can be excluded from atomic regions.

In this thesis, we discuss a few static heuristics -- leaf function based region placement (Leaf), caller/callee based continuous region placement (C&C), and caller/callee/innermost loop continuous region placement (CCIL):

- **Leaf function based region placement heuristic:** A leaf function is an applica-

```
leafFunction {
 region_start;
 ...
 region_commit;
}
```

(a) Leaf

```
main {
 region_start;
 ...
 callerFunction( );
 ...
 region_commit;
}
callerFunction {
 region_commit;
 region_start;
 ...
 calleeFunction( );
 ...
 region_commit;
 region_start;
}
calleeFunction {
 region_commit;
 region_start;
 ...
 region_commit;
 region_start;
}
```

(b) C&C

```
main {
 region_start;
 ...
 callerFunction( );
 ...
 region_commit;
}
callerFunction {
 region_commit;
 region_start;
 ...
 calleeFunction( );
 ...
 region_commit;
 region_start;
}
calleeFunction {
 region_commit;
 region_start;
 ...
 loop1 {
  region_commit;
  region_start;
  ...
  loop2 { ... }
  ...
  region_commit;
  regionstart;
 }
 ...
 region_commit;
 region_start;
}
```

(c) CCIL

FIGURE 4-3. Static region placement heuristics. (a) is the leaf function based discontinuous region placement; (b) is the caller/callee continuous region placement heuristic; (c) is the caller/callee/innermost loop continuous region placement

tion function that does not contain any function call to another application function after inlining has been executed. This heuristic simply treats leaf functions as regions. It assumes that an application spends most of its time in leaf functions. A leaf function can have Java library function calls since a high performance JVM

usually has its own proprietary library implementation and can limit external effects, e.g. the number of writes, of a library function call or at least calls to a majority of library functions. A region start is placed at the entrance of a leaf function and a region commit is placed at the exit as shown in Figure 4-3(a). This approach has two drawbacks. First, there is a fair amount of execution time in non-leaf functions. Second, some leaf functions can generate more write traffic than the hardware's limited logging resources. This could lead to unnecessary replays due to buffer overflow. However, this heuristic generates the smallest number of regions among the three heuristics.

- **Caller/callee continuous region placement heuristic:** This approach, illustrated in Figure 4-3(b), tries to extract speculative optimization opportunities in all functions. It ends the current region and starts a new one at the entrance and the exit of a function except for the main function where a region is started at its entrance and the current region is ended at its exit. Thus all the code in every function is enclosed within an atomic region. One drawback is that C&C can force retention of a bounds check in loops with a function call, since the call forces a region boundary and no bounds check can be moved across this boundary. The region commits/starts could be replaced with conditional commits/starts. When such instructions are executed, hardware commit occurs only if hardware logging resources almost exhausted. Otherwise these instructions are treated as NOPs in hardware and the current and next regions are spliced together. This could help reduce unnecessary commits and starts.
- **Caller/callee innermost-loop continuous region placement heuristic:** This

scheme is more aggressive than C&C and it can further break down multilevel loops with lots of write traffic, which occurs fairly frequently in scientific benchmarks. CCIL creates more small regions than C&C since there can be many multi-level loops in applications that do not generate many writes. An example of CCIL is shown in Figure 4-3(c). CCIL prevents bounds checks in outer loops from being moved out of outer loops. However, the innermost loops are the hottest and moving bounds checks out of such loops can still lead to a significant performance gain. Similar to C&C, the region commits/starts here could be replaced with conditional commits/starts to reduce unnecessary region commits and extend region sizes.

An alternative to static region placement is dynamic region placement based on online profile information. This could lead to a whole new research area which focuses on finding the core profile information and identifying atomic regions that could lead to optimal performance improvement. This falls out of the thesis scope and will be left as future research.

In Section 6.3, we will describe a software technique called region shrinking. It can speculatively hoist up potentially excepting instructions (PEIs), which can effectively reduce the size of an atomic region. The purpose of this technique is to reduce region size so that some gigantic multi-level loops, which require too many logging operations for hardware to efficiently handle, can be excluded from an atomic region.

4.5 Implementation Challenges

In order for the proposed speculative optimization model to work effectively, there

are two main challenges. The first one is to identify speculative optimizations that could save compile time, reduce compile code size, and improve performance. The second one is to reduce the hardware support cost to below the breakeven point where performance gain from better optimizations could offset the extra hardware cost. In order to achieve to overall performance improvement, the performance gain from speculative optimizations needs to outweigh the slowdowns due to the hardware atomicity support. The following two chapters will try to address speculative optimization opportunities and hardware atomicity cost reductions respectively.

Chapter 5

Speculative Optimizations

This chapter describes a few speculative algorithms we have designed and exploited to speed up Java applications under our speculative execution model. The speculative algorithms fall into two categories: one to deal with null pointer checks (null checks or NCs) and the other to deal with array bounds checks (bounds checks or BCs). We discuss the design guidelines for these algorithms and then describe each algorithm in detail.

5.1 Guidelines for Speculative Algorithm Design

In designing our speculative algorithms, we have followed a set of design principles in general.

Performance. The algorithm can be used in an optimizing compiler to reduce Java program execution time.

Complexity. The algorithm does not increase the complexity of other commonly seen algorithms in an optimizing compiler. The interactions between the designed algorithm and other algorithms do not create complex corner cases.

Simplicity. The implementation of the algorithm is manageable by a graduate student.

Compile Time. The algorithm does not require much compile time, ideally less than 1% of a typical optimizing compiler's overall compile time. Preferably, it can reduce the compile time of other optimizations and thus achieve a reduction of the total compile

time.

Code Quality. The algorithm can reduce the final optimized code size. It can also reduce unnecessary branches and minimize the disruptive effects, e.g. loss of spatial localities, due to branches.

5.2 Speculative Null Check Elimination

The non-speculative handling of null checks is to safely generate null pointer checks for each load and store via a pointer and then conservatively determine if a null check is redundant with a previous one. If it is a redundant null check, it can be safely removed. The drawback of this approach is that null checks are generated at the beginning of the optimization flow and that not all of them can be eliminated by redundancy removal algorithms. In the majority of the phases in the optimization flow, there are still many null checks in the code. In the end of the optimization flow, these null checks will be converted to branch instructions and affect the performance.

With the formation of atomic regions and the aid of the operating system to detect a virtual memory page zero access (a null pointer access is equivalent to an access to memory address zero, which is an access to virtual memory page zero), null checks can be speculatively avoided. The corresponding loads or stores will be able to raise an exception when they refer to null pointers. This requires that redundancy elimination optimizations do not remove the last store or load accessing a certain pointer. The requirement can be easily enforced in the optimizing compiler.

Static region formation techniques can form regions very early in the optimization flow, most likely right after the stage where Java byte code is converted to an intermediate

representation. After regions are inserted, analysis can be performed to detect if a basic block is included in an atomic region. If so, all null checks in the basic block can be removed or avoided (depending on whether the null check is actually generated in the byte-code-to-intermediate-representation stage). Similarly, dynamic region formation techniques rely on previous execution information of the same method to insert regions and region boundaries could be known even before re-optimization starts the optimization flow. Therefore, in both region formation techniques, regions are identified early and the speculative elimination of null checks can completely remove the null checks, keep the minimal code size and compile time, and avoid performance degradation in the final generated code due to null check induced branches.

A more general speculative algorithm is shown in Figure 5-1. It is not needed for the static region placement heuristics used in this thesis. It is only used for intraprocedural analysis. In general, dominator and post-dominator information are needed to determine if

```
// only for intraprocedural analysis
Compute dominator/post-dominator information;
foreach basic block (BB) {
  boolean isInRegion = false;
  if ((a region start is in this BB's dominators &&
       there is no region end in between) &&
      (a region commit is in this BB's post-dominators &&
       there is no region start in between))
    isInRegion = true;
  if (isInRegion)
    eliminate all null checks in this BB;
}
```

FIGURE 5-1. Speculative null check elimination algorithm.

a NC is within an atomic region (dominators and post-dominators are already computed for other optimization purposes, hence incurring no additional overhead). First, we examine a basic block's dominators to check that there is a region start and that there is no region commit between the region start and the basic block. Second, we examine a basic block's post-dominators to check that there is a region commit and there is no region start between the basic block and the region commit. The basic block is within a region if both conditions are satisfied; therefore, its NCs can be speculatively removed.

Since the early removal of NCs relies on the execution of the corresponding loads/stores to preserve the exception behavior, caution needs to be taken to prevent data-flow-dead code elimination algorithms from optimizing away dangling loads/stores. An alternative solution is to replace loads that can be optimized away with null checks. It is worth noting that the computations on which data-flow-dead loads/stores are data-dependent cannot be optimized away with or without the speculative NC elimination algorithm.

5.3 Speculative Array Bounds Check Elimination

Bounds checks (BCs) can be removed when they are proved to be subsumed by another BC. Our baseline JVM (Jikes RVM) incorporates ABCD [15], a state-of-the-art BC elimination algorithm. However, there are many remaining BCs that incur performance overhead and impede aggressive optimization. We develop a speculative local BC elimination algorithm and a speculative loop-based global BC elimination algorithm based on the loop monotonic statement detection algorithm proposed by Spezialetti and Gupta [92]. We describe more aggressive loop-based BC elimination algorithms based on application characteristics. We describe our algorithms in the context of upper bounds and

the algorithms' duals (complementary versions) can easily handle lower bounds.

5.3.1 SSA-Based Local Bounds Check Elimination

This clean, general, and lightweight algorithm is designed to speculatively eliminate redundant bounds checks within a basic block (BB). An example is shown in Figure 5-2. The three array accesses are distributed in a basic block. A non-speculative local BC elimination algorithm cannot safely remove the bounds checks for A[i-1] and A[i]. Pure software-based speculation such as check promotion cannot efficiently handle this either. Such speculation needs to promote the strictest check, in this case the check for A[i+1], above this BB, which might not be worthwhile if software needs to keep track of the original check order. Some software speculation requires a replication of the BB with one version containing all checks while the other dropping the checks. This leads to code bloat and can complicate JVM performance tuning. Some might propose to use a stub function that activates the JVM to regenerate this BB with checks when the promoted check fails at runtime execution. One stub function per BB and the BB regeneration information needed for this stub function can easily introduce enough overhead to offset the gain from speculatively removing some bounds checks.

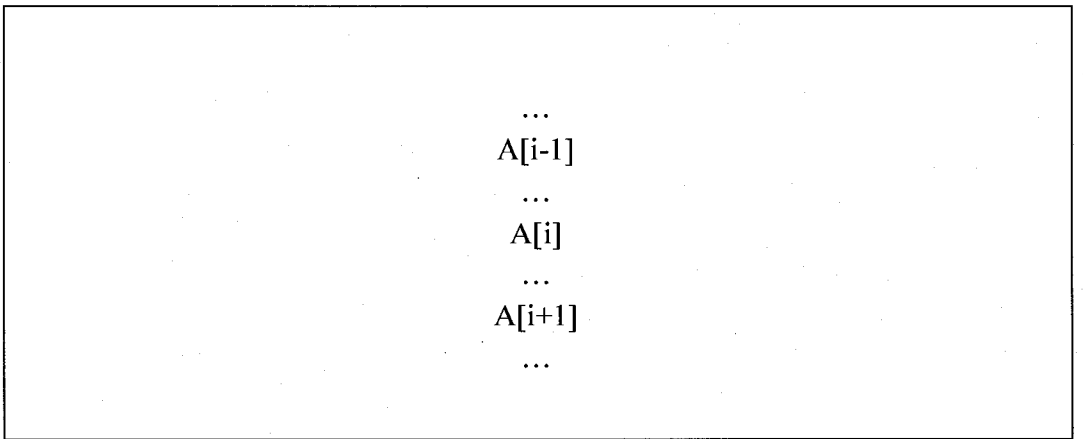


FIGURE 5-2. Example for local bounds check elimination.

With our approach, the two bounds checks can be speculatively removed without introducing any runtime overhead, since we no longer have to maintain their relative order. Our algorithm reduces the code size and adds zero runtime overhead in the commonly executed code.

The algorithm's prerequisites are SSA and def/use chains. The algorithm is more efficient if local common subexpression elimination (CSE) is performed in advance. A tuple <array ref register, index register, constraint> is used to represent a bounds check. Here, constraint represents the difference between the index register and bounds check index value. For example, A[i-1] is converted to <A, i, -1>, A[i] to <A, i, 0>, and A[i+1] to a tuple <A, i, 1>. Different bounds checks are compared against each other regardless their program order. Tuples belong to the same group if their array ref registers and index registers are the same. In the same group only the bounds check with the largest constraint is not redundant. Bounds checks with constant array indexes are converted to tuples belonging to a group with the array ref register and a special index register. The algorithm is shown in Figure 5-3. In tracing back the index register's def chain we only consider moves and additions/subtractions involving a register and a constant; other operations with more general forms can be included later.

The algorithm's efficiency depends on the efficiency of array SSA construction. Due to the difficulty of pointer analysis, it is sometimes impossible to construct comprehensive SSA form for certain array pointers. As shown in Figure 5-4, it is almost impossible to prove that x.a[5] and x.a[4] use the same array pointer so that x.a has to be assigned to two different variables in array SSA. In Jikes, array SSA form is constructed conservatively and then global value number analysis can help prove certain array pointers to refer

1. Convert an array bounds check (BC) A[index] to a tuple.
   - If index is a register (r_1) defined using move, trace back to the defining statement (r_n=...) that is not a move instruction and start from 1 to create a tuple for A[r_n].
   - If index is constant, convert BC to <A, special_reg, constraint>.
   - If index is defined by an addition/subtraction that involves a register (r_1) and a constant, create a tuple <A, r_1, constraint>.
   - If the index register is defined by a phi instruction or it is a parameter register, create a tuple <A, index, constraint>.
2. Check the tuple in its specific group.
   - If the group does not exist, create its group and update the group's current BC.
   - If the group exists and its current constraint is larger than or equal to this tuple's, mark the BC redundant.
   - If the group exists and its current constraint is smaller than this tuple's, mark the previous BC redundant and update this group's current BC to the new one.

FIGURE 5-3. SSA-based speculative local bounds check elimination algorithm.

to the same one. In the implementation of our algorithm, we do not use global value number analysis although it could possibly help improve our algorithm coverage.

```
y = x;
x.a[5] = 5;
y.a = z.a;
.. = x.a[4];
```

FIGURE 5-4. Array SSA construction limitation example.

5.3.2 Loop-Based Global Bounds Check Elimination

Our loop-based algorithm is superior to loop versioning [56][71], a software-based speculative technique to remove BCs in loops, since loop versioning adds runtime execution overhead, increases the code size, and only works for specific loops. It can limit the

effectiveness of other optimizations such as loop unrolling and dramatically increase the difficulty of the JVM performance tuning. Our algorithm adds zero runtime overhead, does not increase the code size, and is applicable to all loops.

The algorithm is developed based on a loop monotonic statement detection algorithm in [92]. According to [92], a loop monotonic statement is one that always increases or decreases a variable while a loop invariant statement assigns the same loop invariant to its modified variable during loop iterations. The algorithm characterizes statements in a loop as monotonic, invariant, and chaotic. Loop monotonic statements can be divided into three categories: basic, dependent and cyclically monotonic statements. A basic loop monotonic statement does not depend on any other loop monotonic statement. A dependent loop monotonic statement depends on at least one loop monotonic statement. A cyclically monotonic statement is one that depends on another monotonic statement that directly or indirectly depends on the cyclically monotonic statement itself. In another word, a cycle is formed in the dependency chain. In the example shown in Figure 5-5, statement (4) is a basic monotonic statement; statement (1) is a dependent monotonic statement; statement (2) and (3) are cyclically monotonic statements.

Our algorithm focuses on the monotonicity of variables instead. The value of a loop monotonic variable always gets increased or decreased while that of a loop invariant variable remains unchanged during loop iterations. The overview of the algorithm is given in Figure 5-6. The algorithm requires loops, dominators, and def chains to be computed first. Among loops, inner ones are processed before outer ones.

In step 1 variables used as array subscripts are identified. The monotonicity analysis targets such variables instead of every variable or statement in a loop in order to reduce

```
for (i = 1; i < 10; ) {
  ...
  j = i + 2;          (1)
  n = m + 1;          (2)
  m = n + 2;          (3)
  ...
  i++;                (4)
}
```

FIGURE 5-5. Three kinds of loop monotonic statements.

1. Find variables used as array subscripts.
2. Traverse the def chain of array subscript variables until variables with no in-loop definitions.
3. Construct data dependence graph (DDG) for the identified variables.
4. Characterize variables into potentially basic monotonic variables, potentially dependent monotonic variables, and potentially cyclically monotonic variables.
5. Prune potentially cyclically monotonic variables and mark them chaotic.
6. Identify the initial values of variables positive/negative/either/non-negative/non-positive constants.
7. Derive monotonicity of potentially basic monotonic variables.
8. Derive monotonicity of potentially dependent monotonic variables in a topological order on the DDG.
9. Move array bounds checks outside the loop if the subscript variables are invariant or monotonic and the array reference pointer can be moved outside the loop.

FIGURE 5-6. Loop-based speculative bounds check elimination.

the analysis cost.

Step 2 finds all the variables necessary for the monotonicity analysis of array subscript variables. The traversal of the def chain for an array subscript variable finds all the in-loop ancestors of array subscript variables. Additional early traversal termination con-

ditions can be introduced to reduce computation cost. First, a variable in the def chain has more than two in-loop defining statements with different operators. Second, one of the defining statements is a load instruction or a call instruction.

Step 3 is to construct a data dependence graph (DDG) for all the variables to be analyzed. A DDG is a directed graph representing all the analyzed variables towards each other.

Step 4 marks the tree root nodes in the DDG as potentially basic monotonic. It then identifies the Strongly Connected Components (SCCs) in the DDG and other variables that are data dependent on at least one variable in a SCC. These are potentially cyclically monotonic variables. In step 5, we mark such variables chaotic to avoid analyzing them in later stages.

Step 6 identifies the initial values of the variables if they have initial values upon the entrance of the loop. This is not a trivial task as it relies on dominators and post-dominators to sort out the relationships of different definitions of a variable outside the loop. We notice that three special cases can cover many cases in programs. Case 1 is that the variable is assigned to a constant in the immediate dominator of the loop. In Figure 5-7, variable j falls into this case for the inner loop. In case 2, a variable only has one definition outside the loop and it is in a dominating basic block other than the immediate dominator of this loop. Variable k for the inner loop in Figure 5-7 is an example. In case 3, the variable is initialized as a constant upon the entrance to the outer loop and gets increased/decreased by a constant in the outer loop iteration. Variable i for the inner loop is an example for the third case. The three special cases can significantly reduce the computation cost while capturing most opportunity.

```
for (int i = 1, k = 1; i < n; i++) {
  for (int j = 1; j < n; j++) {
    A[i]
    ...
    A[j]
    ...
    A[k++]
  }
}
```

FIGURE 5-7. Example for variable initial value identification.

Step 7 and 8 derive the monotonicity of potentially basic monotonic and potentially dependent monotonic variables. Here, our definition of monotonicity also includes invariance, which is different from S&G. We use their algorithms to characterize variable's monotonicity. We also add support for instructions such as move, neg, and shift.

Step 9 moves BCs outside the loop if possible. We avoid replicating the first iteration and the last iteration to prevent code bloat. The final BC is checked after the loop if the subscript is monotonically increasing; the initial BC is checked before the loop if the subscript is monotonically decreasing. Both BCs need to be checked if the subscript is monotonic. The BC can be moved to either place if its subscript is a loop invariant. Necessary compensation may be applied for the subscript variable of a BC that is moved out of the loop since the final value of the array subscript variable may be the value at the final loop iteration plus the variable's stepping value. We also rely on speculation to simplify BC motion as illustrated in Figure 5-8. Two pad basic blocks (BB2p and BB4p) and a replicated branch in BB2p need to be generated to guarantee the correctness of moving BCs outside the loop as shown in part (b) in Figure 5-8 if no speculation is used. BCs can not

be directly moved to BB2 and BB4 as they might not be executed in the original loop. 56
However, this worry is unnecessary since almost all busy loops execute at least one iteration. Moving BCs to BB2 or BB4 will rarely cause misspeculation. With support for guarded regions we can safely put BCs in BB2 and BB4 assuming that replay rarely happens. Therefore, we can keep the original loop structure. BCs can still be moved out of the loop even if they are executed conditionally. However, this is more likely to cause guarded regions to roll back.

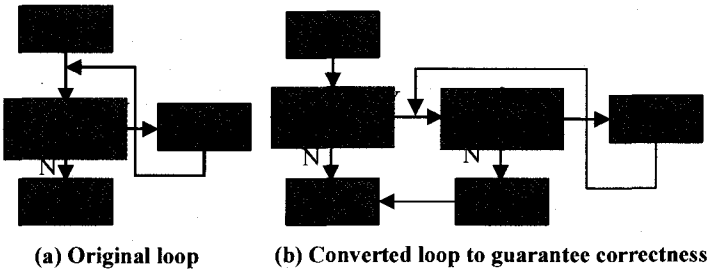


FIGURE 5-8. Simplified loop manipulation with atomic regions.

5.3.3 More Loop-Based Global Bounds Check Elimination

A typical array access pattern we have seen in real applications cannot be captured by the loop-based global algorithm. The programmer sets an upper bound for the value of an array-indexing variable. In the example in Figure 5-9 the programmer assumes that j can not be larger than 15. In this case the speculative optimizing compiler can safely assume that array A has a size most likely larger than 15. Therefore, a BC A[15] can be placed before the loop and the BC in the loop can be eliminated. This is an example of slightly riskier speculation. The dynamic compiler can not guarantee that array A has a size larger than 15 but it makes an educated guess that this should be most likely true.

Therefore, it decides to speculatively hoist the BC.

```
for (int i = 0; i < n; i++) {
  ...
  j = (j + 1) & 0xf;
  A[j];
  ...
}
```

FIGURE 5-9. Bounds check with non-monotonic loop variable.

Many applications access multidimensional arrays, as shown by the example in Figure 5-10. For such arrays the loop-based algorithm has limited effectiveness. The loop-based algorithm can move the BC for A[i] outside the outer loop, but the BC involving variable j can only be moved outside the inner loop. For applications with many multidimensional array accesses, many such opportunities remain unextracted. One possible solution is to provide hardware support for register min/max value monitoring, and replay a guarded region if a particular register reaches a value that exceeds the array bounds. This approach works well for the example in Figure 5-10 and for most other array access patterns. In the example, two values need to be watched in hardware - the min value v1 for the array length of A[i] and the max value v2 for variable j. The BC involving j can be completely eliminated in the loop. Then v1 is compared to v2 after the loop. An exception is thrown and the guarded region replays if v1 is less than v2. Register value monitoring can even be applied to array BCs involving non-monotonic variables. This solution requires the processor to have enough registers to hold each monitored variable. The current IA32 processor only has 8 integer registers and register spills can occur, complicating code generation and potentially causing performance hazards. However, 64-bit

AMD64/EMT64 extensions to IA32 have 16 registers, Itanium has 128 registers, and Power5 has 32 registers. Hence, future processors will have more registers and register spills will become less of an issue for register value monitoring.

```
for (int i = 0; i < n; i++) {
   ...
   for (int j = 0; j < m; j++) {
      ...
      A[i][j]
      ...
   }
   ...
}
```

FIGURE 5-10. Example code with continuous 2D array access.

A further complication arises in the presence of asymmetrical arrays. An example is shown in Figure 5-11. In the example A[0] and A[2] have a different array length from A[1]. If hardware monitors the value of the array length for A[i] and the value of j for A[i][j] to ensure upper bounds checking validity, hardware can only make sure that the current max value of j is less than the current max value of A[i] array length. This could still lead to the possibility of the violation of upper bounds checks. As the in the assymtric array in the example, if there were a two-level loop traversing A in the row and then the column order, the max value for A[i] array length seen by hardware after accessing the first row would be 4. If j had a value of 4 in the traversal of the row A[1], the upper bound would be violated while hardware could not detect this. This is why assymmetric arrays need to be addressed. A possible solution could be given from either the language level or the implementation level. The language can specify symmetrical arrays. The JVM can

also include a flag in its array implementation to indicate an array is symmetrical. In addition, asymmetrical multidimensional-arrays rarely occur in real applications, which should ease the possible implementation of the proposed solution.

```
int [][] A = new int[3][];
A[0] = new int[4];
A[1] = new int[3];
A[2] = new int[4];
```

FIGURE 5-11. Asymmetrical array.

5.4 Other Possible Speculative Algorithms

This section describes a few possible speculative algorithms that are very easy to implement. We have not seen noticeable performance improvement in our benchmark suite; however, for a particular benchmark they might give significant performance gain.

5.4.1 "Catch"-Based Speculative Dead Code Removal

Some Java applications may have a large amount of catch clauses to handle exceptions. The code in catch clauses is usually on the cold path and rarely gets executed. Therefore, catch clauses rarely affect code performance. However, the JVM still generates code for the catch clauses and the exception handling paths still exist in the control flow graph (CFG). Where there are many catch clauses, they can slow down compilation and increase the code size.

Figure 5-12 shows an example where a catch clause is enclosed by an atomic region. With rollback support the catch clause can be simply removed. In case of an exception occurring, the code can be re-compiled conservatively with the catch clause and

re-run to determine the right action for the thrown exception. 60

```
region_start
...
try {
...
}
catch (...) {
...
}
...
region_commit
```

FIGURE 5-12. Example for "catch" clause removal.

In our SPECjvm98 and Java Grande benchmarks, we do not see many catch clauses. However, the Java library functions have a fair amount of catch clauses. The extensive use of library functions could possibly cause longer compiler time and larger code size if catch clauses were compiled.

5.4.2 Speculative Loop Invariant Code Motion for PEIs

Our loop-based bounds check elimination algorithm already covers speculative loop invariant code motion for bounds checks. Our speculative null check elimination algorithm can completely remove the null checks if the code is in an atomic region due to the operating system support. However, if an operating system did not support exception-on-zero-page-access, loop invariant code motion could help move some null checks out of

the loops.

With our speculative algorithms to remove null checks and bounds checks (the most common PEIs in Java code), there are not many PEIs for consideration for loop invariant code motion. Loads and stores are notably the majority of PEIs excluding null checks and bounds checks. If a program has many loads and stores that are loop invariant, they can be speculatively moved out of the loop if the loop is guarded by an atomic region. Unfortunately, our benchmarks do not have many loop-invariant loads and stores that can have a noticeable performance impact. However, for the right benchmark, this speculative algorithm could still improve its performance by moving loads and stores out of the loops.

5.5 Other Possible Design Explorations

Our research has been focusing on the design of new efficient algorithms to achieve performance improvement. Another direction, as in [73], is to look into the improvement of existing algorithms on Java code that is in an atomic region. For such code, the cold path in the CFG can be converted to assertions and the CFG can be greatly simplified. Work in [73] shows that performance improvement could be improved by simply applying existing algorithms to simplified CFGs.

5.6 Summary

Among all the speculative optimizations discussed in this chapter, I evaluated the speculative null pointer check elimination algorithm, the SSA-based local bounds check elimination algorithm, and the loop-based global bounds check elimination algorithm and the results of evaluation are in Chapter 7. However, other speculative optimizations could

possibly be useful given the pertinent hardware design and Java applications. 62

Chapter 6

Reduce Hardware Atomicity Support Cost

Support for hardware atomicity comes at a price. There are two general requirements for hardware to support an atomic region rollback. First, hardware needs to restore register values upon re-entrance to the atomic region. This can be done by checkpointing register values and restoring those checkpointed values on recovery. Second, the memory state needs to be rolled back to the same point. This is usually done by buffering the speculative writes or logging the old values during atomic execution. Most prior work in speculative optimization, as well as in transactional memory, which has a similar atomicity requirement, assumes heavyweight hardware support for both of these operations. Given the current trend towards many relatively simple cores per die, we are skeptical that such heavyweight hardware support will materialize. Furthermore, whether or not such support is strictly necessary remains an open research question. We instead assume minimal hardware support, and examine various hardware and software alternatives for reducing the frequency of both register checkpointing and write logging.

Our minimal machine requires few changes to existing processor hardware. Both registers and memory values are logged at instruction commit during region execution to a hidden address range in pinned physical memory. This machine model requires very modest hardware changes: control logic for monitoring atomic region starts and commits and managing an in-memory log, an extra register file read port in the instruction commit stage, arbitration logic and datapath support to read old values from the cache at instruction commit, and support for performing additional cache writes at instruction commit.

Since recovery is infrequent, so it is performed entirely in software by the runtime system, by replaying the log entry and recovering both register and memory state. We also assume simple extensions to the ISA to allow software techniques to communicate to hardware which registers need to be checkpointed and which writes need to be logged. We assume that the region_start operation specifies a register mask that indicates which registers need to be checkpointed, and that each memory write is preceded by a LOG instruction that indicates that the previous value at that memory location needs to be logged. [1]

Given this straightforward hardware support, we propose a variety of software and hardware techniques to reduce register checkpointing and unnecessary logging in atomic hardware for Java programs. For register checkpointing reduction, we exploit register calling conventions (SW), register dirtiness analysis (SW), and the physical register file free-list buffer (HW). For logging reduction, we propose a stack write logging elimination algorithm (SW), a heap write logging elimination algorithm (SW), region shrinking (SW), and write buffering (HW). A brief overview of these techniques is provided here, with detailed discussions in later sections of the chapter.

Register calling conventions. Atomic region placement aligned with function calls provides a natural way for cost reduction of both register checkpointing and write logging. Calling convention utilization relies on this to significantly reduce the amount of register checkpointing needed. The continuous caller/callee placement scheme, one of the atomic region placement schemes discussed in Section 4.4, aligns atomic regions with function calls.

---

1. Alternatively, a reserved bit in the instruction word itself could indicate the need for register checkpointing or write logging, but it is usually easier to add new instructions than it is to modify existing instruction encoding.

The JVM's calling convention breaks down registers into volatiles and non-volatiles. It also specifies registers used for function parameters and return values. The values of volatiles do not need to be preserved across a call site and thus it is unnecessary to checkpoint them if atomic regions are aligned with calls.

Register dirtiness analysis. This determines if a non-volatile or a return register is modified in the atomic region after a function call return. If they are not modified, it is not necessary to checkpoint them.

Free-list buffer. The physical registers in an out-of-order processor buffer the previous values of modified registers until the instructions writing those registers retire from the instruction window. At that point, the physical register containing the previous value is based on the free list. By delaying this action using a free-list buffer, we can further preserve these values until the region commits, subject to availability of physical registers.

Stack writes. The first software technique is the stack write logging elimination algorithm. The executing thread's stack is extended upon the entrance of a callee (an atomic guarded region) and many writes within the guarded region store to this new stack frame. In case of a rollback, the new stack frame would be destroyed and rebuilt. Therefore, it is unnecessary to perform logging for such stack writes. We design an algorithm to identify such stack writes and evaluate the effectiveness in a particular guarded region placement scheme - continuous caller/callee placement. We find that the algorithm can remove on average 68% and as high as 99% of the stack write logging.

Heap writes. The second software technique is a heap write logging elimination algorithm. We notice that many Java heap writes are always executed before any reads to the same addresses in an atomic guarded region. Our algorithm, based on a unified heap

analysis framework in [31], can identify such heap writes and eliminate unnecessary logging for such writes. In the continuous caller/callee placement scheme, we find that our algorithm can remove on average 30% and as high as 51% of the heap writes.

Region shrinking. Region shrinking utilizes speculative PEI hoisting to reduce the effective region size. After the last speculative PEI executes, the region is no longer speculative and thus no checkpointing or logging is needed. This technique effectively reduces logging overhead due to large, busy loops with array access patterns that can be analyzed at compile time.

Write buffer. This technique relies on an on-chip write buffer's buffering capability to delay the logging operation for a write. When a write is retired from the write buffer and the corresponding guarded region commits, it is unnecessary to perform logging for this write. A 64-entry write buffer combined with the above software techniques can remove on average 94% of the write logging for the studied benchmarks.

6.1 Techniques to Reduce Register Checkpointing

Both software and hardware can help reduce register checkpointing. In this section, we present two software techniques (register calling convention and register dirtiness analysis), and one hardware technique (the free list buffer).

6.1.1 Software Technique I: Register Calling Convention

Register calling convention deals with function call parameter passthroughs, function call returns, volatile registers, and non-volatile registers. Volatile registers do not need to be saved across call sites while non-volatile registers need to keep their values across call sites and are typically saved by either a caller or a callee. Utilizing register calling

convention can be a powerful technique for eliminating register checkpointing when atomic regions are aligned with function calls.

In the continuous caller/callee placement scheme, regions are inserted at call boundaries and thus there is no need to save volatile registers upon the entrance and the exit of an atomic callee. In this scheme, there are two types of atomic regions as shown in Figure 6-1. The first type is a region (type I) that is before a function call and the second one is a region (type II) starting just after the return from a function call.

```
atomic region boundary I
callee_func(...) {
  prologue;  code;  epilogue; }
atomic region boundary II
```

FIGURE 6-1. Two types of atomic regions.

In reality, microprocessors, systems, compilers, and programming languages are often designed by different vendors. Therefore, microprocessors make no assumptions about the possible calling convention and they simply give software freedom to determine volatile registers and non-volatile ones. Software usually uses certain registers or the stack to pass in parameters; it also uses registers or the stack to return values from a function call. Some registers are marked as non-volatiles and they need to be saved and later restored if they are used in a callee. The calling convention used by Jikes RVM on x86 32 bit processors are show in Table 6-1. A type I region needs to save EAX and EDX if they are used for parameter passthrough and modified within the atomic region. Non-volatiles that are used in a callee are saved to the stack in the prologue and thus they do not need to be checkpointed even if they are modified. None of the non-volatiles need to be checkpointed. In a type II region, the return register EAX needs to be checkpointed if it is used

to return a value and modified in the following region. Volatiles do not need to be check- 68
pointed. Non-volatiles need to be checkpointed.

Table 6-1: Calling convention for Jikes RVM on 32bit x86 processors.

Function parameters	EAX, EDX
Returns	EAX
Non-volatiles	EBX, EBP, EDI
Volatiles	FP registers, EAX, ECX, EDX, ESI, ESP

Assume that an x86 32 bit processor has 8 general purpose registers and 8 floating point registers. Type I regions can avoid checkpointing 11 out of the 16 registers and Type II regions can avoid checkpointing 12 out of the 16 registers if we conservatively checkpoint registers used for the parameter passthrough and the function return. Actually, many functions have no parameters or returns. This could lead to further register checkpointing savings.

6.1.2 Software Technique II: Register Dirtiness Analysis

A backward dataflow-based register dirtiness analysis has been developed to determine if a non-volatile register or the return register (EAX) needs to be checkpointed in an atomic region after a return from a callee function. It can be performed in a function where more than one atomic region have been placed. It can give you a minimal set of registers to be checkpointed, which can be a much smaller set than simply counting modified registers. The analysis is performed after register allocation is done.

Type I regions are not considered in this algorithm since such regions are aligned with callees and the callee prologue can save non-volatiles and parameter registers to the stack. An exception handler simply needs to copy back the saved content back to the original registers when a rollback occurs.

Type II regions are what the analysis focuses on. It detects which of the non-vola-

tile registers and the return register are written in the following atomic region. If they are not modified, there is no need to checkpoint them.

The algorithm is shown in Figure 6-2. The algorithm identifies if a target register will possibly be modified after a call. If so, this register needs to be checkpointed for the next region. For the return register EAX, it is a little different. If the call doesn't have any return, there is no need to worry about the return register. It calculates the Kill and Gen sets for each basic block and then initializes a work list with all the basic blocks in a certain order. Then it starts iterating until the work list becomes empty. With the calculated out set for a basic block (if it has calls), it can trace backwards to find the checkpoint register set after each call in this basic block.

A simple example is shown in Figure 6-3. In step I, it calculates the Kill and Gen sets of BB1, BB2, and BB3. Step III calculates the In and Out sets of the three basic blocks. In Step IV, it finds the checkpoint set after call 1 and call 2. In call 2, it removes EAX from the set since the call does not have any return.

6.1.3 Hardware Technique I: Instruction-Window Buffering

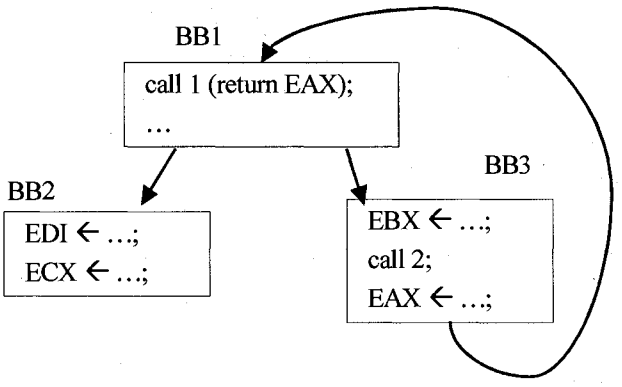
An instruction window is used by a microprocessor to execute instructions out of order but retire them in order [89][47]. In design, it can be a circular buffer. Instructions are inserted at the head pointer and retired at the tail pointer in the program order. Each entry contains information about opcodes, operands (both operands and their readiness), and other state information (such as exception, instruction commit, etc.). Ready instructions (whose operands are ready) in an instruction window are issued out-of-order based on a certain set of issuing and selection logic. After an instruction finishes execution, it sits in the instruction window, waiting for its turn to retire. If it writes to an architectural

```
Target registers: non-volatiles and EAX (return register)
Kill (call) = {all registers}, Gen (call) = {}
Kill (other) = {}, Gen (other) = {target register modified}
Algorithm:
Step I)  Initialize Kill (BB) and Gen (BB) to empty.
         foreach inst in BB in reverse order {
           Kill(BB) = (Kill(BB) + Kill(inst)) – Gen(inst);
           Gen(BB) = (Gen(BB) – Kill(inst)) + Gen(inst);
         }
Step II)  Initialize workList as reverse top order of basic blocks.
Step III) while (workList is not empty) {
            b = remove_top(workList);
            Out(b) = Union of In(s) if s is a successor of b;
            In(b) = (Out(b) – Kill(b)) U Gen(b);
            if (In(b) changes) add its sucessors to workList;
          }
Step IV)  Foreach BB that has call instructions {
            checkpoint_set = Out(BB);
            foreach inst in BB in reverse order {
              checkpoint_set = (checkpoint_set – Kill(inst) U Gen(inst);
             }
           }
          Remove EAX from a call's checkpoint_set if there is no return;
```

FIGURE 6-2. Dataflow algorithm to remove unnecessary register checkpointing.

register, the value is usually buffered in the instruction window until the entry retires.

The buffering capability by an instruction window provides a great opportunity for the reduction of register checkpointing. A register does not need to be checkpointed if the first instruction writing to it is still in the instruction window, or a logical extension of the window. When this instruction retires from the instructions window, there is a high proba-



Step I:

Kill(BB1) = {EBX, EDI, EBP, EAX}, Gen(BB1) = {}

Kill(BB2) = {}, Gen(BB2) = {EDI}

Kill(BB3) = {EDI, EBP, EAX}, Gen(BB3) = {EBX}

Step III:

In(BB1) = {}, Out(BB1) = {EDI, EBX}

In(BB2) = {EDI}, Out(BB2) = {}

In(BB3) = {EBX}, Out(BB3) = {}

Step IV:

checkpoint_set(call 2) = {}

checkpoint_set(call 1) = {EDI, EBX}

FIGURE 6-3. Dataflow algorithm example.

bility that the atomic region that this instruction belongs to is known to be exception free. In this case, no checkpointing needs to be performed for this register.

This hardware technique can help further save checkpointing for registers that are identified by software as possible checkpointing targets for a particular atomic region. The proposed software techniques can identify a large set of registers that do not need to be checkpointed. The register checkpointing information from software to hardware can be transferred using a register marking instruction at the beginning of a region. The instruction takes only one cycle and it marks away the registers that do not need checkpointing

for this region.

The instruction window retirement policy needs to be adjusted to achieve maximal register buffering. We extend the lifetime of physical registers by using a free-list buffer, which delays the placement of registers on the free list until the corresponding region commits. As long as the free list has enough available registers, we can entirely avoid register checkpointing, since the checkpointed values are maintained in the physical register file.

6.1.4 Discussion

Calling conventions are designed to facilitate the interprocedural register allocation. The breakdown of registers into volatiles and non-volatiles considers the trade-off of the extension of registers across call sites versus the explicit saving of non-volatiles during the call. In the new context of register checkpointing for hardware atomicity, additional checkpointing cost may make the breakdown worth revisiting. The right breakdown can certainly minimize the overall cost and thus increase performance.

Calling conventions can only be exploited for atomic regions aligned with function calls. For regions created on other boundaries, an atomic region convention similar to calling convention could be designed to reduce register checkpointing.

The effectiveness of the register dirtiness analysis certainly relies on the choice of the register allocator and the cost model adopted by the allocator. Jikes RVM uses linear scan [79] instead of graph-based register allocation [19]. Further, the register allocator might include the checkpointing cost into its cost model when it comes to register allocation decisions.

The amount of write logging can be greatly reduced using a combination of software and hardware techniques. In this section, we present two software techniques and one hardware technique (write buffering) to help us significantly reduce write logging.

6.2.1 Stack Write Logging Elimination Algorithm

This algorithm is a software technique that can be implemented in a JVM. It can help remove the unnecessary logging operations for stack writes. The algorithm relies on two observations. First, atomic regions are aligned with function entrances and exits in many region placement schemes such as the continuous caller/callee placement scheme. Second, many stack writes only modify the portion of the stack that would be destroyed and rebuilt if the region were rolled back and re-executed. Therefore, the compiler can perform analysis to identify eligible stack writes that do not need logging support. Furthermore, in a strongly typed language like Java, the compiler can easily check that a guarded region only includes regular stack reads and writes, i.e. an executing thread's stack is only accessed normally and there are no aliases to the stack in the guarded region, which should be the common case in almost all Java code.

The algorithm, as shown in Figure 6-4, can be applied early in the optimization flow, e.g. right after guarded region placement, to identify basic blocks (BBs) where stack writes do not need logging. The algorithm starts by finding basic blocks with call-introducing instructions (CIIs) and tries to identify the set of basic blocks where stack writes only write to the stack portion that could be rebuilt in case of a rollback. If a BB does not need any logging, stack writes inserted into this BB later will not need logging support. The algorithm gets activated if the scope of a guarded region aligns with the allocation and

deallocation of a stack portion, e.g. a function entrance and a function exit. Within a region, check instructions are converted to assertions so that they are not CIIs.

```
Prerequisite: guarded region placement
Algorithm:
  Step I ).  Find set Φ with basic blocks (BBs) containing CIIs;
  Step II).  Foreach loop L from the innermost to the outermost  {
               if L's loop body has a CII     {
                 Collapse the loop body to a single extended BB;
                 Mark this extended BB to have a CII;
                 Add L's loop head to set Φ;
               }
             }
  Step III). Find set Ψ with BBs dominating all the BBs in set Φ;
```

FIGURE 6-4. Algorithm to remove unnecessary write logging.

An example is shown in Figure 6-5 to show how the algorithm works. There are atomic region boundaries before the function entrance and after the function exit. There is a call instruction in basic block (BB3) and thus there is a region boundary before and after the call. BB3 and BB4 form a loop. In step I, BB3 is added to set Φ. In step II, an extended basic block that represents the loop including BB3 and BB4 is added to the set. In step III, BB1 and BB2 are found to dominate all the BBs in set Φ and thus stack writes in BB1 and BB2 do not need any logging.

In order for the algorithm to be valid, no new CIIs can be inserted after this algorithm is performed. The optimization flow of a JVM can easily be aware of this. In addition, stack write generating algorithms such as register spills in register allocation probably need to fine tune their heuristics to maximize the placement of stack writes in

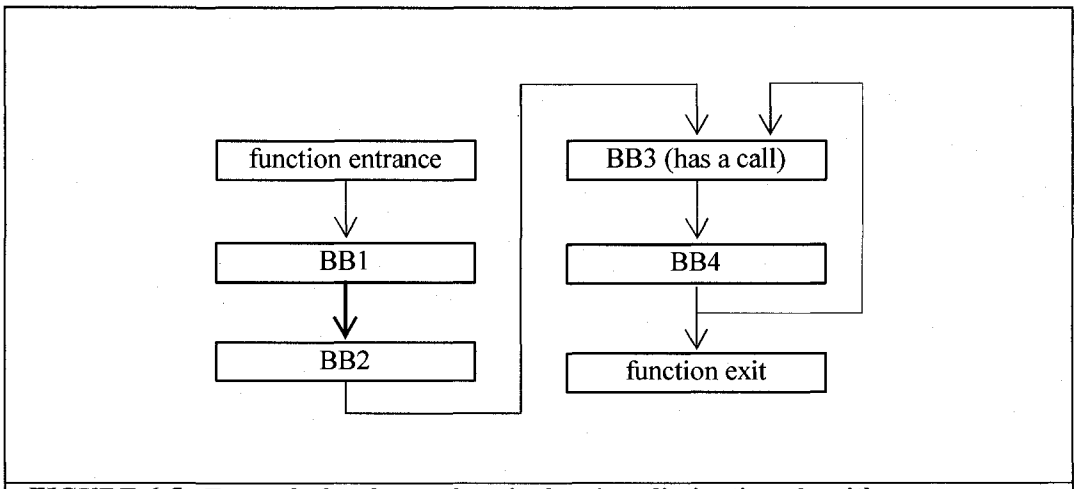


FIGURE 6-5. Example for the stack write logging elimination algorithm.

BBs without any need for stack write logging support.

6.2.2 Heap Write Logging Elimination Algorithm

This section presents an algorithm that helps eliminate the logging support for some heap writes. The algorithm is developed from an observation that a heap write always occurs before any heap read for many scalar variables and array accesses in a guarded region. This means that the old value for this particular scalar variable or array access is no longer needed and thus no logging is needed. In case of a rollback, a write always occurs before any other read and thus the old value will not be needed. The optimization flow can be easily designed so that any code re-optimization due to a rollback will not move a write before any read.

Our algorithm is constructed based on the heap variable analysis framework developed in [31]. A heap variable represents an object instance field access, a global static field access, or an array element access. Heap variables representing the former two are called scalar heap variables (accessed via putstatic/getstatic and putfield/getfield in Java byte code) and heap variables representing array element accesses are called array heap

variables (accessed via a load or store in Java byte code). According to the definition here, an array pointer access via either putstatic/getstatic or putfield/getfield is a scalar variable. The breakdown into scalar and heap variables here is a little different from the usual breakdown in compiler intermediate representation such as scalar and array SSA where an array pointer is regarded as a non-scalar variable.

In the algorithm shown in Figure 6-6, step I constructs the Static Single Assignment (SSA) form including both scalar and array SSA and step II performs Global Value Number (GVN) analysis on the constructed SSA form. The algorithm needs SSA for the following reasons. First, it allows Global Value Number (GVN) analysis to be done more efficiently. Second, it makes it easier to tell that writes are before reads when then access the same SSA names. Third, it can prevent some aliasing possibilities to make analysis easier. The GVN analysis is performed for all variables, not just the variables that have SSA names. Jikes routinely performs SSA construction and GVN analysis so we simply reuse the code there. Jikes also provides two functions, DefinitelySame (DS) and DefinitelyDifferent (DD), in its GVN analysis to help differentiate object/array instances or scalar values from other object/array instances or scalar values. The two functions use the GVN analysis results coupled with other information such as object/array allocation locations and function parameters. Parameters are very rarely aliases but it could be hard to prove statically. We can simply assert that parameters are not aliases at the very beginning of a region if necessary. If there were an alias, a rollback would happen and conservative recompilation would be used. We extended the DefinitelyDifferent function further with the object type information. Due to Java's strong typing, two objects of different types where neither is a subclass of the other are different from each other and thus occupy dif-

Step I: Construct scalar and array Static Single Assignment (SSA)

Step II: Perform Global Value Number (GVN) analysis

Step III: Distribute scalar/array heap variable accesses into (n+1) distinct groups with a search algorithm using DefinitelyDifferent and DefinitelySame on objects or arrays

Each of the n groups contains heap variable accesses referring to one distinct object or array instance

Last group contains other heap variable accesses

Step IV: for each of the (n+1) scalar heap variable groups

if (each read has a write that accesses the same heap location &&

the write dominates the read)

the writes in this group do not need logging

Step V: Break each of the first n array heap variable groups into (m+1) sets with a search algorithm using DefinitelyDifferent and DefinitelySame on array indexes

Each of the m sets contains array accesses to the same array indexes that are not accessed by array accesses in other sets

Last set contains other array accesses

Step VI: for each of the (m+1) sets of each of the n array heap variable groups

if (each read has a write that accesses the same heap location(s) &&

the write dominates the read)

the writes in this set do not need logging

Step VII: for the last of the (n+1) array heap variable groups

if (each read has a write that accesses the same heap location(s) &&

the write dominates the read)

the writes in this group do not need logging

FIGURE 6-6. Algorithm to remove unnecessary heap write logging.

```
bool DefinitelyDifferent(o1, o2) {
 // GVN assisted analysis
 if (there exists a constant o3 && GVN(o3) == GVN(o1) &&
     there exists a constant o4 && GVN(o4) == GVN(o2))
   return GVN(o1) != GVN (o2);
 if (there exists o3  && GVN(o3) == GVN(o1) && o3 is created w/ NEW &&
     there exists o4  && GVN(o4) == GVN(o2) && o4 is created w/ NEW)
   return GVN(o1) != GVN(o2);
 if (there exists o3  && GVN(o3) == GVN(o1)  && o3 is created w/ NEW &&
    o2 can be traced back to a function parameter  ||
    there exists o3  && GVN(o3) == GVN(o2)  && o3 is created w/ NEW &&
    o1 can be traced back to a function parameter)
   return true;
 if ( GVN_Analyzer.congruenceClass(o1) has a parameter &&
      GVN_Analyzer.congruenceClass(o2) has a parameter &&
      no alias in parameters)
   return GVN(o1) != GVN(o2);
 // type assisted analysis
 if (type(o1) is known && type(o2) is known)
   return (type(o1) != type(o2) &&
           neither is a subclass of the other);
 return false;
}
bool DefinitelySame(o1, o2) {
 // GVN assisted analysis
 if (GVN(o1) == GVN(o2))
   return true;
 else
   return false;
}
```

FIGURE 6-7. DefinitelyDifferent (DD) and DefinitelySame (DS).

ferent heap locations. We only made a limited attempt at clearly known types. If it were too hard to do type analysis, we would simply give up and return false. The two functions, DefinitelyDifferent and DefinitelySame, are shown in Figure 6-7.

Step III creates n+1 groups for both scalar and array heap variable accesses with a simple search algorithm that uses DefinitelySame and DefinitelyDifferent on object and array pointers (global static variables accessed via putstatic/getstatic can be viewed as a special instance of scalar heap variables that can be put into a group with which no object reference pointer is associated). The search algorithm is shown in Figure 6-8. It is used here to illustrate how groups can be created and it may not be the most efficient search algorithm. The function in Figure 6-8 is to break scalar heap variables into groups. The function with all the obj(input) subfunctions replaced with the array(input) functions can be used to break all array heap variables into groups (the definitions of obj(input) and array(input) are shown in the same figure). In each of the first n groups, the heap variable accesses only refer to an object or an array instance that is definitely different from other object or array instances. The last group contains the heap variable accesses that cannot be identified as pointing to a unique object or array instance. The last group can contain heap variable accesses pointing to more than on object or array instance. In the next step, we analyze scalar heap variable groups to find if write logging can be removed for the writes in these groups. In order to remove write logging safely, all reads need to have dominant writes that access the same memory locations. The write logging in the last group can be similarly removed even though it has array heap variable accesses pointing to more than one object or array instance. However, the existence of array heap variable access pointing to more than one object or array instance in the last group simply reduces the chances that

```
// This is for scalar heap variables and the input array should only have scalar heap variables
// A similar search function can be used to break  array heap variables to groups; there
// obj(input) needs to be replaced with array(input) and heap_vars[ ] can only have
// array heap variables.  The first  returns input's object pointer while the second
// returns input's array pointer.
Vector searchForGroups( heap_vars[ ] ) {
  Vector groups = new Vector( );
  Vector misc_grp = new Vector( );
FirstForeach:
  foreach heap_var in heap_vars[ ] {
    if (heap_var is marked not DD)
      continue;
    foreach grp in groups {
      // obj(input) returns input's object pointer;
      // it could be replaced with array(input), which could return input's array pointer
      if (DefinitelySame(obj(heap_var), obj(first_element(grp)))
        add heap_var to grp and continue from FirstForeach;
    }
    foreach heap_var1 in heap_vars[ ] other than heap_var and heap_vars[ ] elements that are DD
      // obj(input) could be replaced with array(input)
      if (!DefinitelyDifferent(obj(heap_var), obj(heap_var1)))
        mark both not DD, add both to misc_grp if not already there, and continue from FirstForeach;
    // this is a DD heap_var
    mark heap_var DD, create a new group for heap_var, and add this group to groups;
  }
  groups.add(misc_grp);
  return groups;
}
```

FIGURE 6-8. Search algorithm to generate groups.

the write logging there can be removed since a read from an object or array instance can affect a write to another object or array instance.

In step V, we further break array accesses to unique array instances into m+1 sets using a search algorithm using DefinitelyDifferent and DefinitelySame. The search algorithm is similar to what is used in step II except that it is used on array index variables. The search algorithm is shown in Figure 6-9. In each of the first m sets, array indexes have the same values that are definitely different from those of indexes in other sets. In the next step, we can perform analysis on each such set to decide if write logging can be removed for this set. In order to do so, reads need to have dominant writes that access the same locations. In the last step, we take care of the last array heap variable group that cannot be broken into sets. For this group, we can safely remove write logging if all reads have dominant writes that access the same memory locations.

Figure 6-10 shows two examples for our algorithm -- one for scalar heap variables and the other for array heap variables. In example_func, there are four scalar heap variable accesses -- foo1.a, foo2.a, globalFoo1.a, and globalFoo2.a. Our algorithm is able to create (2+1) groups where n equals to 2. The first group contains the scalar heap variable accesses for foo2.a since foo2 points to a unique foo instance; the second group contains the scalar heap variable accesses for foo1.a since foo1 points to a unique foo instance; the third group contains the scalar heap variable access for both globalFoo1.a and globalFoo2.a since globalFoo1 and globalFoo2 might point to the same foo instance. In the first group, there are no writes to foo2.a. In the second group, there is a read and a write for foo1.a and the write dominates the read. Therefore, logging is not needed for this write. In the last group, the read from globalFoo1.a does not have a dominant write and

```
// The input heap_vars[ ] contains the array heap variables that are in one single array
// heap variable group that refers to one unique array instance
Vector searchForSets( heap_vars[ ] ) {
  Vector sets = new Vector( );
  Vector misc_set = new Vector( );
FirstForeach:
  foreach heap_var in heap_vars[ ] {
    if (heap_var is marked not DD)
      continue;
    foreach set in sets {
      // index(input ) returns input's index variable for the array access
      if (DefinitelySame(index(heap_var), index(first_element(set)))
        add heap_var to set and continue from FirstForeach;
    }
    foreach heap_var1 in heap_vars[ ] other than heap_var and heap_vars[ ] elements that are DD
      // index(input ) returns input's index variable for the array access
      if (!DefinitelyDifferent(index(heap_var), index(heap_var1)))
        mark both not DD, add both to misc_set if not already there, and continue from FirstForeach;
    // this is a DD heap_var
    mark heap_var DD, create a new set for heap_var, and add this set to sets;
  }
  sets.add(misc_set);
  return sets;
}
```

FIGURE 6-9. Search algorithm to generate sets for array heap variables.

thus the logging for the write to globalFoo2.a cannot be removed. If there were a write to globalFoo1.a dominating the read from globalFoo1.a, the write loggings for both globalFoo1.a and globalFoo2.a could be removed since both all reads in the third group

would have dominant writes to the same memory locations.

In example_func2, there are four array heap variable accesses -- B[globalIdx1], B[globalIdx2], A[3], and A[2]. Our algorithm creates (2+0) groups since all the heap variables can be put in groups with definitely different array instances. There is no last group where it holds heap variables for arrays instances that might overlap with others. In the first group, the heap variables are put into (0+1) sets. There are no sets that hold heap variables for definitely different array indexes for array B. The last set holds B[globalIdx1] and B[globalIdx2] and the two indexes may or may not be the same value. The read from B[globalIdx2] does not have a dominant write and the logging for the write to B[globalIdx1] needs to be preserved. In the second group, the heap variables are put into (2+0) sets. Indexes of 2 and 3 are know values and can be definitely different from each other and thus there is no last set that hold indexes that cannot be distinguished from each other. In the first set, there is no write. In the second set, there is a write that dominates the only read and thus the write logging can be removed.

The GVN analysis can help improve the coverage of the write logging that can be removed. As shown in Figure 6-11, the heap variables for A[3] and A[j] would be put into the last set of the array heap variable group for A, where the read of A[j] of does not have a dominant write. Therefore, the write logging for both A[3] and A[j] could not be removed. With GVN, 3 and j can be proved to have the same value. A[3] and A[j] can be put into one of the first n sets of the array heap variable group for A. In this set, the read of A[j] has a dominant write in "A[3] = ...". Therefore, the write logging for both A[3] and A[j] can be removed.

The type analysis we added in the DefinitelyDifferent function can also improve

```
void example_func1( ) {
  foo1 = new foo( );
  foo2 = new foo( );
  int b = globalFoo1.a;
  int c = foo2.a
  foo1.a = 2;
  int c = foo1.a;
  globalFoo2.a = 4;
}
```

```
void example_func2( ) {
  int [ ] A = new int[4];
  int [ ] B = new int[4];
  B[globalIdx1] = 2;
  int b = B[globalIdx2];
  int c = A[3];
  A[2] = 5;
  int d = A[2];
}
```

FIGURE 6-10. Example for heap write logging removal.

the coverage of the write logging removed. As shown in Figure 6-12, the types of two arrays (intArray and doubleArray) are int[] and double[] respectively. The GVN analysis could not differentiate them since the arrays are not constructed within the function or passed through function parameters. The write logging for "doubleArray[3] = ..." could not be removed since there was a read of intArray[3] before the write. With type analysis, intArray and doubleArray are definitely different and thus put into two different groups. Further analysis in the algorithm can simply tell that the write dominates the read for intArray[3] and thus the write logging can be removed.

In order for this software technique to work completely correctly, the optimization flow must be aware of the existence of the application of this algorithm and the interactions of this algorithm with other algorithms that may affect this algorithm's correctness. For example, speculative load/store hoisting might affect the validity of this algorithm. Other global value based optimizations such as constant propagation (if they are across a

```
void example_func( ) {
 …
 int [ ] A = new int[4];
 i = 2;
 if (…)
  j = 3;
 else
  j = i + 1;
 A[3] = …;
 int a = A[j];
 A[j] = …;
 …
}
```

FIGURE 6-11. Example where GVN can enhance algorithm coverage.

```
void example_func( ) {
 …
 int b = intArray[3];
 doubleArray[3] = …;
 double c = doubleArray[3];
 …
}
```

FIGURE 6-12. Example where type analysis can enhance algorithm coverage.

single atomic region) might affect the validity.

6.2.3 Write Buffering

On the hardware side, the processor can use its on-chip buffering resources to remove unnecessary logging. Hardware techniques are more general than software techniques and usually address both heap and stack writes. A write buffer, which is implemented in many microprocessors, is one such on-chip buffering device that can help remove logging for both stack and heap writes. A write buffer saves stores and lets stores commit to the cache hierarchy later. It is usually designed as a Content Address Memory (CAM) and can provide simultaneous lookups to provide load forwarding. A write buffer can help relieve the increasing latency and the demanding memory bandwidth constraints for data caches.

The utilization of a write buffer to remove unnecessary logging simply takes advantage of the delayed store commits to the cache hierarchy. By the time a store is retired from the write buffer and saved to the data cache, it is possible that the owning region is already known to be exception free and safe to commit. Therefore, no logging operation is needed for this evicted store. If many evicted stores are known to be non-speculative, logging cost can be greatly reduced.

However, this does not come without any additional cost. The effectiveness of removing unnecessary logging might require bigger write buffers, which have larger lookup latencies and whose latencies are harder for the pipeline to accommodate. Furthermore, the effectiveness may also require the tuning of the write buffer draining heuristic to keep more stores in the write buffer as long as possible, which can potentially cause the write buffer to be frequently full and affect future store instructions.

In our work, we use a two-phase write buffer retirement policy. When the write

buffer is less than half full, we reduce the frequency of store retirement from the write buffer. When the write buffer is more than half full, we use a normal write buffer retirement policy used in a microprocessor. This proves to have almost no performance impact while greatly reducing the need to log writes.

The write buffer needs very minimal extension as shown in Figure 6-13. Each entry simply needs two new fields to remember the region id and a no-log bit. The region id shows the region that owns this store. The no-log bit indicates if the store requires logging when it retires from the write buffer. During the execution of a region, stores are saved into the write buffer. When the region is known to commit, its region ID is looked up in the store buffer and all the valid entries with the same region ID will have their no-log bit set so that no logging will be performed when this store is evicted. If an entry is selected to be replaced before the no-log bit is set, a logging operation is performed.

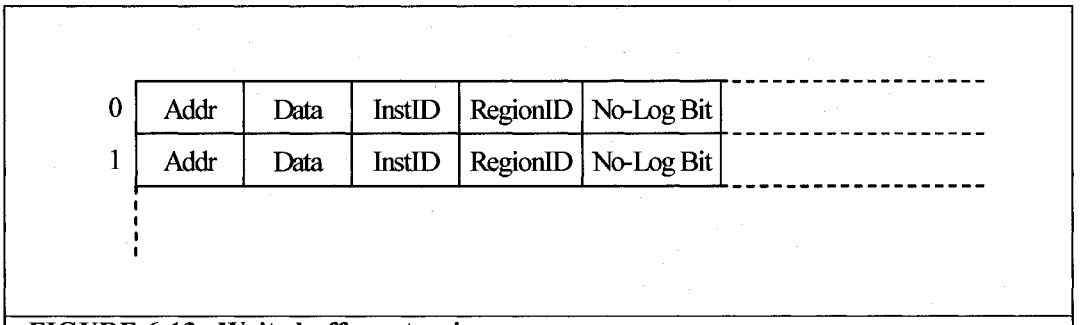


FIGURE 6-13. Write buffer extension.

6.2.4 Discussion

The stack write logging elimination algorithm relies on the alignment of atomic regions with function calls. Java programs have a very high stack write percentage. This makes stack write logging elimination very attractive.

The heap write logging elimination algorithm relies on Java's strong typing. It may not work well for weakly typed languages.

The instruction window buffering could possibly be used on top of a write buffer to further reduce write logging.

6.3 Region Shrinking

The section presents a software technique called region shrinking which speculatively hoists PEIs to effectively reduce the region size. After the last speculative PEI is executed, the original region is no longer speculative and thus the rest of the region no longer needs register checkpointing or write logging. The last speculative PEI is followed by the hoisted region_commit, which needs to remember its original location. When an exception is thrown in the speculative part, the region needs to be recompiled according to the original region scope. However, no action is needed if an exception is thrown in the non-speculative part since it reports an exception in the correct order and does not violate Java's precise exception model. This technique is extremely helpful for large multi-level loops that are seen in some benchmarks. For most benchmarks and most regions, region shrinking is set to be off by default. It is only turned on when a region in a benchmark is large and busy and an ideal candidate for region shrinking. In a dynamic environment like a JVM, profile information can be easily obtained to decide which regions need this transformation.

Figure 6-14 shows an example of how region shrinking can be done. The example is a simplified version of the busiest loop in the benchmark sor. The loop traverses a 2D array multiple times. The original code has null checks and bounds checks generated in the program order and none are speculatively removed. After the 3-level loop is placed in an atomic region, speculative null check elimination algorithms and speculative bounds

check elimination algorithms can remove null checks and move bounds checks to places shown in the middle column of Figure 6-14 early in the optimization flow. The drawback of this approach is that the 3-level loop generates many write instructions and the hardware logging cost can be big enough to offset any speculative optimization gain. With region shrinking, the speculative bounds check elimination algorithms can hoist bounds checks to before the 3-level loop since it can easily figure out the checking boundary values. The speculative null check elimination algorithm can have two options. It can either move the null check for G to before the 3-level loop while leave the null check for Gi in the original place. It can also simply leave both null checks in place. In either case, the remaining null checks will be removed since they will be combined with the load instruction at the end of the optimization flow (see Section 3.5 for how null check combining is done in Jikes). Figure 6-14 shows the second case where the two null checks are remaining in the original places but will be removed due to null check combining with loads. Since the bounds check elimination accounts for most of the speculative optimization gain, it does not cause much speculative optimization performance loss to leave null checks in place in the majority phases of the optimization flow. With the speculative PEI hoisting, all speculative checks can be moved before the loop and the loop can execute in a non-speculative state. This can completely free hardware from the burden of write logging in large loops. This technique works extremely well for sor and db where large loops are observed while it is unnecessary for the rest of the studied benchmarks.

6.4 Summary

Our register checkpointing reduction and write logging reduction techniques con-

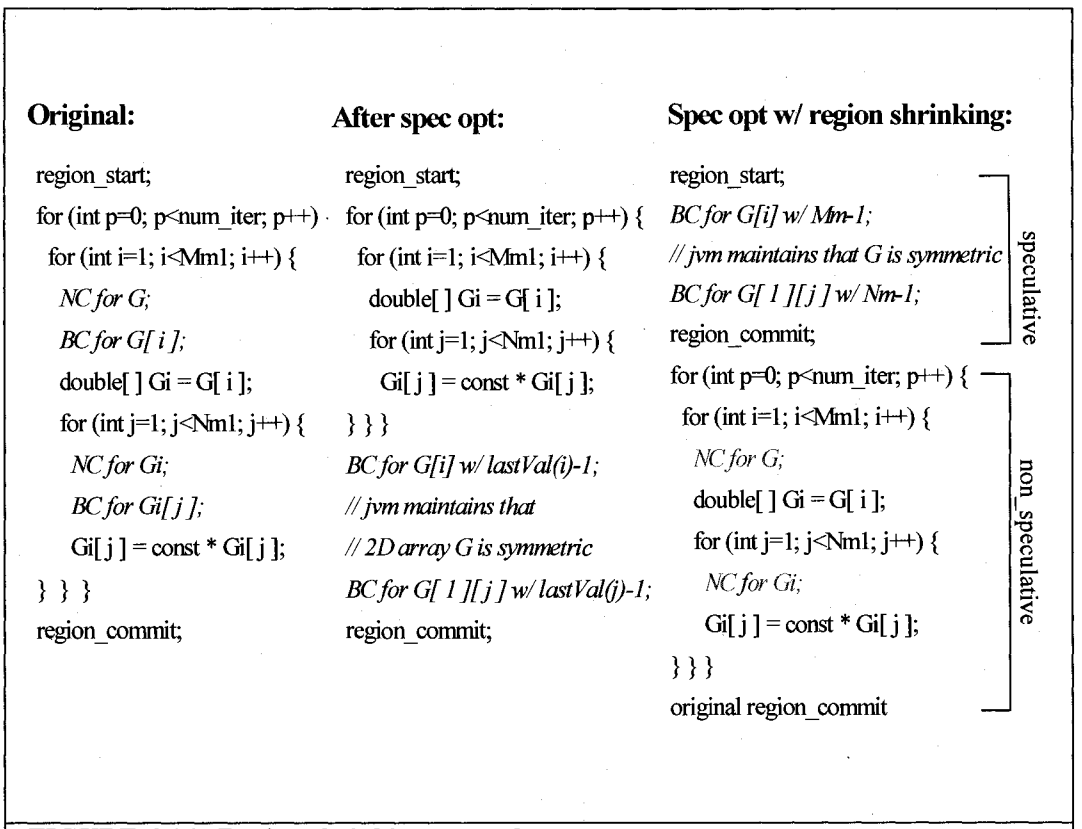


FIGURE 6-14. Region shrinking example.

sist of both software and hardware approaches. Software and hardware can both contribute and it is the combination of the two that has proven to be very powerful. In the following chapter, I will present evaluations of these techniques.

Chapter 7

Experimental Studies

This chapter presents our experimental methodology and detailed results. The experimental study includes two steps. The first step is an exploratory study that focuses on the extraction of speculative performance opportunities and the design of speculative optimizations. We use native machine execution as the experimental platform to quickly evaluate ideas and identify the performance potential. The second step concentrates on the evaluation of the atomicity support hardware cost and the techniques to reduce the hardware cost. In this step we extract detailed traces from native machine execution and then feed the traces to a detailed cycle-accurate simulator to fully evaluate our proposed execution model -- speculative execution with atomic guarded regions. The experimental results validate that our execution model is a very promising one for Java and the hardware cost is manageable in a co-designed environment that includes a JVM and a microprocessor.

7.1 Step I: Exploratory Studies on Native Machine Execution

7.1.1 Methodology

The main goal is to identify performance improvement opportunities from the application of lightweight speculative optimizations within an atomic region. In our studied benchmarks, no exception is ever thrown and no replay is ever needed. A thorough qualitative evaluation, using native machine execution, provides not only a quick turnaround time, but also reasonably accurate performance estimates that are more than adequate for achieving our goal.

Experiments are performed with Jikes RVM [5] v2.3.4 on a 2.4GHz Pentium4 based uniprocessor machine with 1GB memory and Redhat Linux 2.4.22.

Jikes RVM is built with production configuration. Methods are directly compiled at opt2 (the highest optimization level) by the optimizing (opt) compiler, which shows the impact of speculative optimizations and leads to a quick and easy comparison between the baseline and the optimized version.

We use the SPECjvm98 benchmarks [91] and two benchmarks in Java Grande [33]. The benchmark information is shown in Table 7-1. We follow the run rules and run benchmarks multiple times to report the best numbers.

Table 7-1: Benchmark Information.

Benchmarks	Descriptions
compress	LZW compression program
jess	NASA's CLIPS rule-based expert system
db	Data management benchmark
maudio (mpegaudio)	MPEG-3 audio stream core algorithm
mtrt	Program ray-tracing an image
jack	Real parser-generator
sor	Successive over-relaxation algorithm
euler	Program in computational fluid dynamics

In our study, we use Intel VTune performance analyzer [55] to identify method hotness information and use the Performance Counter Library (PCL) [12] to measure the region size for atomic regions.

7.1.2 Results

We show that the proposed speculative optimizations can improve performance with perfect region placement. We also evaluate the compile time overhead of the implemented speculative algorithms. Finally, we show that the proposed automatic placement algorithms can achieve a good percentage of the potential speedup from perfect region

placement. We also show that the proposed hardware support for conditional region commit/start can be a key to the success of speculative optimizations and different placement schemes.

7.1.2.1 Perfect region placement

By perfect region placement we mean that regions can be ideally placed so that all possible speculative optimizations can occur within a region. An example of perfect region placement is treating the whole application as a region, hence assuming the hardware to have an effectively unbounded logging capability. In this situation we can apply our speculative algorithms without worrying about region boundaries. In Figure 7-1, we show the speedups of the benchmark suite due to the incremental application of our speculative algorithms. The average performance increases from 5.7% to 10% to 15.9% with the addition of speculative NC elimination, local BC elimination, and global BC elimination. The baseline run times are shown in Table 7-2. Compress's performance is not affected by our algorithms. Compress has about 60 BCs in total and its performance critical BCs can not be eliminated by our algorithms. There could be a speedup of more than 12% if such BCs could be speculatively eliminated.

7.1.2.2 Speculative optimization compile time and coverage

Our speculative algorithms are lightweight. The NC elimination algorithm iterates through basic blocks and removes NCs after a method is identified to be within a region. With perfect region placement and the proposed static region placement heuristics, a method is either in a region or not. Therefore, the NC elimination algorithm introduces almost zero overhead. With other potential region placement heuristics that randomly place regions within a method, the compile time increase due to the dominator/post-domi-

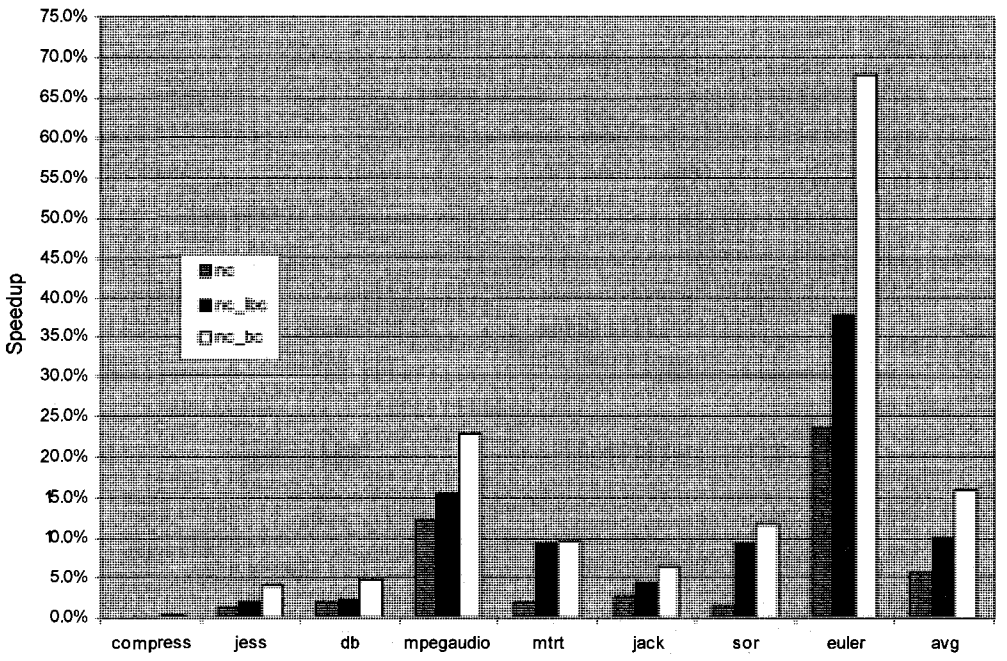


FIGURE 7-1. Speedups for perfect region placement. Here, nc is speculative null check elimination; nc_lbc is nc plus speculative local bounds check elimination; nc_bc is nc_lbc plus speculative loop-based bounds check elimination

Table 7-2: Baseline Run Time.

Benchmarks	Size (bytecode)	Time (seconds)
compress	19k	5.959
jess	35k	2.835
db	20k	15.740
maudio (mpegaudio)	51k	5.04
mtrt	24k	2.765
jack	36k	0.416
sor	10k	4.19
euler	22k	2.6

nator based speculative NC elimination should account for well less than 1% of the overall optimizing compiler's compile time according to our estimation.

The BC elimination algorithms are also very efficient. The local one and the global one account for no more than 0.51% and 0.37% of the overall compile time, as shown in Figure 7-2.

The percentages of BCs removed by our algorithms are shown in Figure 7-3. The

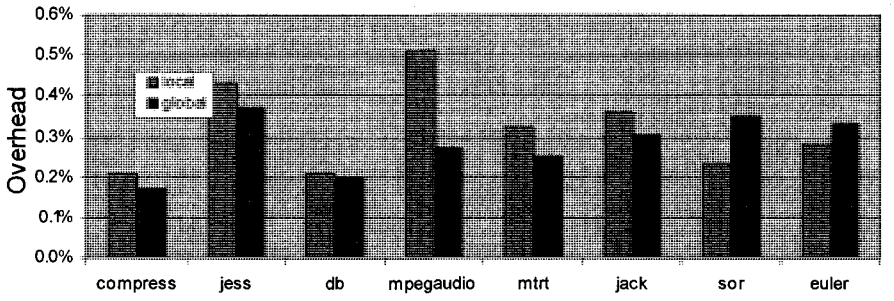


FIGURE 7-2. Overhead of bounds check elimination algorithms. The y axis represents the percentage of each algorithm's compile time among the overall compile time.

algorithm coverage is high. In perfect region placement, the coverage is more than 70% except for db and compress. In db the hot BCs are captured while in compress they are not captured. The three static region placement algorithms can capture many BCs captured by perfect placement.

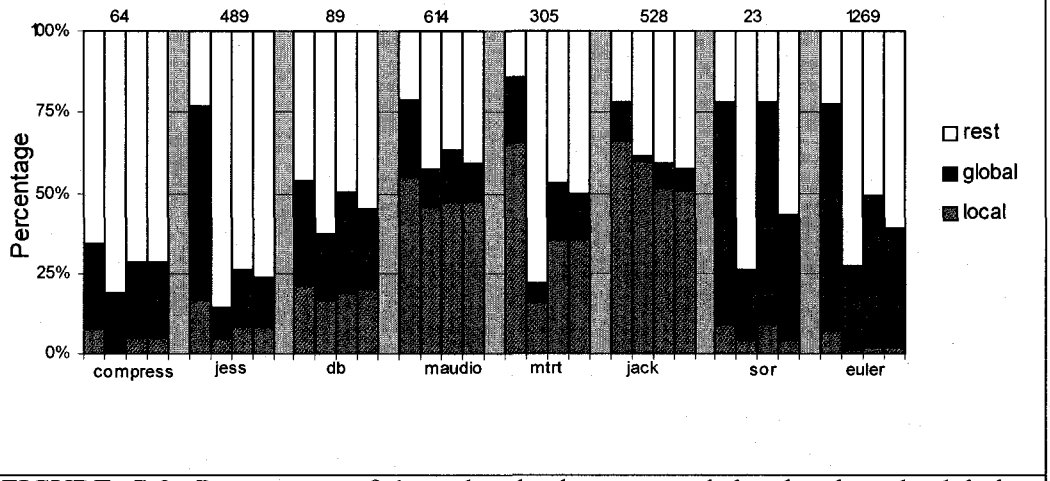


FIGURE 7-3. Percentage of bounds checks removed by local and global algorithms. The numbers at the top are the number of bounds checks for each benchmark. The four bars are perfect region, leaf, C&C, and CCIL. The three series, local, global, and rest, correspond to bounds checks removed by the local algorithm, removed by the global algorithm, and untouched

7.1.2.3 Automatic region placement

Automatic region placement should satisfy two conflicting goals -- the reduction of hardware resource requirements and the retention of speculative opportunities. It is a delicate art to find the right balance for the two conflicting goals.

Our automatic region placement algorithms can effectively extract the performance improvement achievable by perfect placement, as illustrated in Figure 7-4. The effectiveness of leaf depends on the fraction of program execution time in leaf functions. For the benchmarks with most execution time in leaf functions, leaf extracts almost all opportunities. C&C typically performs better than leaf since it factors in non-leaf functions. It does not perform as well as perfect region placement because some bounds checks can not be moved outside loops due to function calls in the loop body. CCIL performs almost as well as C&C. CCIL's performance is slightly worse since region boundaries are also formed right before and after the innermost loop and BCs cannot be moved across these boundaries. CCIL can help effectively break down large regions--for example in db and sor--to avoid unnecessary replays caused by insufficient hardware logging resources.

The number of regions and the average region size are shown in Figure 7-5 and Table 7-3 respectively. The region size is measured in terms of the number of dynamic writes in the region. For many applications such as compress, jess, maudio, and mtrt, the majority of regions are small ones with fewer than 100 writes in all three algorithms. If the hardware can buffer 4K writes, it can typically hold many regions before a commit. Conditional region end/starts are necessary and useful for such applications. Even for other applications such as db, jack, sor, and euler, a region commit occurs, on average, after tens of regions for most of their data points.

The leaf algorithm generates far fewer regions than the other two. However, it can-

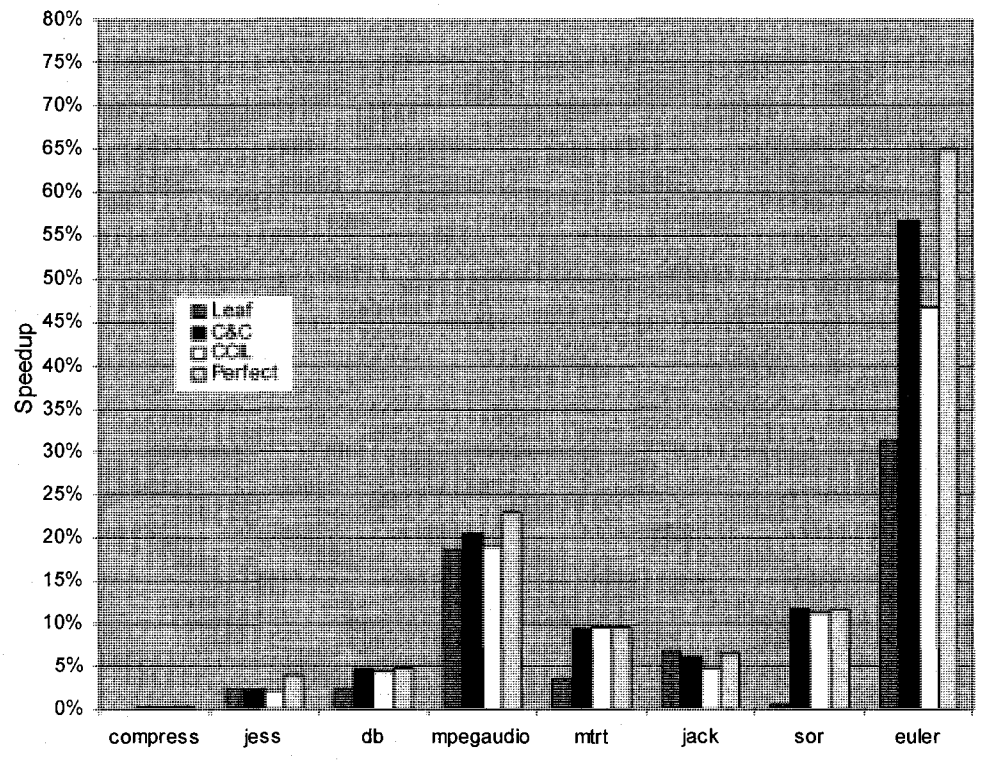


FIGURE 7-4. Speedups for leaf, C&C, and CCIL.

not capture some hot functions in quite a few applications such as compress, db, sor, and euler. C&C generates more regions than leaf. In db and sor, some hot functions have huge multilevel loops enclosed in regions, leading to the big average region size. In sor, there are only 27 regions and a few of them generate millions of writes, leading to the big average region size. These big regions cannot be captured by the leaf algorithm. CCIL breaks down some huge multilevel loops and helps bring down the average region size.

Region sizes are dependent on input sets. For jvm98 benchmarks we use the industry standard benchmarking inputs. For Grande benchmarks we use inputs with reasonable run time. When inputs become larger, better ways to break down large regions will be critical to fully explore the speculative algorithms' benefit.

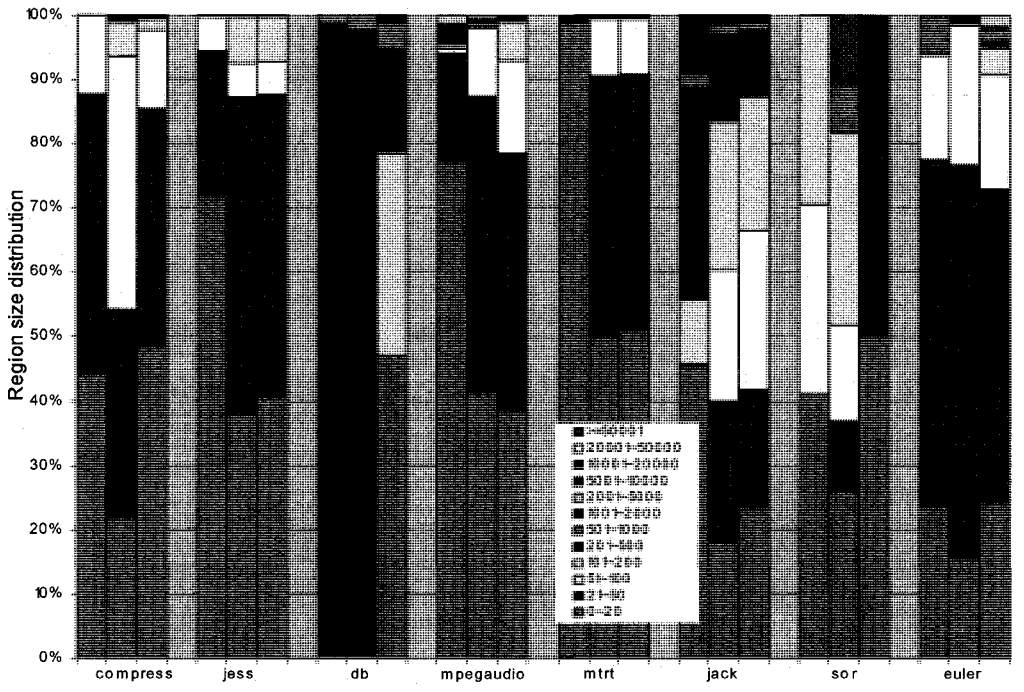


FIGURE 7-5. Region size distributions for leaf, C&C, and CCIL (left to right). The numbers here are the numbers of dynamic writes. Each color represents a range of the number of writes by a region

Table 7-3: Total number of regions executed and average region size.

App	compress	jess	db	maudio	mtrt	jack	sor	euler
Leaf	20m/28	13m/18	1.4m/697	29m/70	12m/11	160k/268	17/79	702k/300
C&C	39m/55	47m/37	1.6m/1304	61m/45	40m/26	1m/150	27/11984003	1m/231
CCIL	79m/31	49m/35	26m/184	73m/41	41m/25	1.2m/116	306k/780	1.4m/127

7.2 Detailed Evaluation on a Simulated Machine

7.2.1 Methodology

The goal of this study is to further evaluate the performance benefit on a more realistic simulated machine with the support of hardware atomicity. In this study, the hardware cost of atomicity support is fully considered and the proposed techniques for cost reduction are evaluated in detail.

We employ trace-driven simulation as our evaluation methodology. Trace-driven simulation is repeatable and can accurately model a simulated machine. It is also reasonably fast compared with an execution-driven simulation methodology. A big drawback of trace-driven simulation is the storage space required for the traces, especially when a significant portion of the workload runtime is simulated. To address this issue, we use shared memory to generate the trace on the fly during the simulation. The trace-collection process and the trace-driven simulator communicate with each other via a shared memory region. As shown in Figure 7-6, we run the trace-generation process and the trace-driven simulator process separately and the two processes synchronize with each other using a shared memory region.

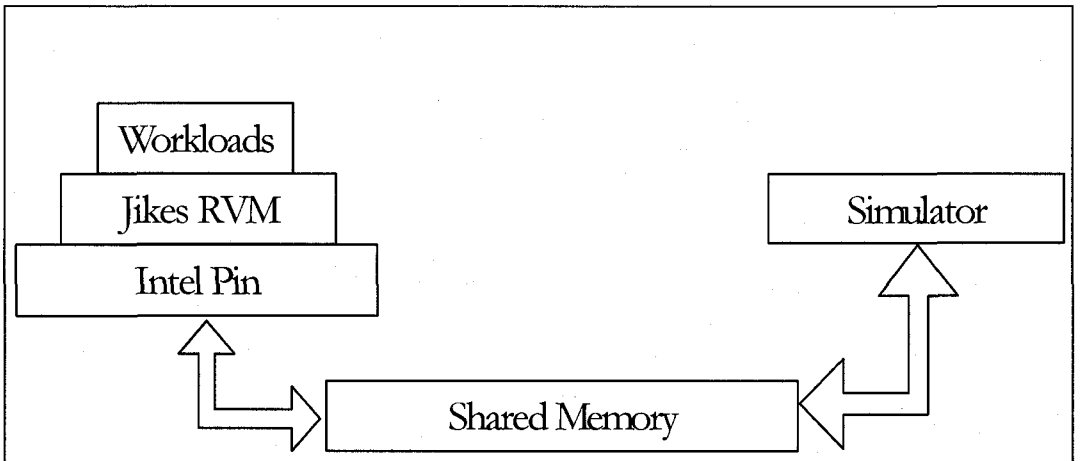


FIGURE 7-6. Our trace-driven simulation methodology.

In the experimental study, we implement our compiler algorithms in Jikes RVM 2.3.4 [5]. The compiler algorithms include both the speculative optimizations and the hardware cost reduction techniques. The Java applications are directly compiled at the highest opt level available and a continuous region placement scheme, the caller/callee placement, is used. We run Jikes RVM on top of Intel Pin [62] to extract the Java applica-

tion instruction trace. We use the same set of benchmarks [91][33] as described in Section 7.2.1.

7.2.2 Simulated Machine Model

Most prior work in speculative optimization, as well as in transactional memory, which has a similar atomicity requirement, assumes heavyweight hardware support for register checkpointing and memory write logging. In the heavyweight hardware support, all the registers are checkpointed when an atomic region is entered and all the memory writes are attempted to be logged within an atomic region. Given the current trend towards many relatively simple cores per die, we are skeptical that such heavyweight hardware support will materialize. Furthermore, whether or not such support is strictly necessary remains an open research question. In this experimental study, we instead assume minimal hardware support, and examine various hardware and software alternatives for reducing the frequency of both register checkpointing and write logging.

Since our speculative dynamic optimizations are profile-driven, we assume that recovery is infrequent, so it is performed entirely in software by the runtime system, by replaying the log entry and recovering both register and memory state. We also assume simple extensions to the ISA to allow software techniques to communicate to hardware which registers need to be checkpointed and which writes need to be logged. We assume that the region_start operation specifies a register mask that indicates which registers need to be checkpointed, and that each memory write is preceded by a LOG instruction that indicates that the previous value at that memory location needs to be logged.

First of all, as an upper bound on performance, we model an ideal machine, where both register checkpointing and write logging incur no performance cost. For a real

machine to approach this level of performance it would have to support instantaneous and boundless register file checkpointing, provide an additional cache read port to acquire the write values that need to be logged, and dedicate die area for an effectively unbounded write log. While it might be possible to build such a machine, the likely effects on cycle time, area, leakage power, and dynamic power would call into question the value of such radical hardware changes.

At the other end of the spectrum, we model a minimal machine (illustrated in Figure 7-7a), which requires few changes to existing processor hardware. Both registers and memory values are logged at instruction commit to a hidden address range in pinned physical memory. We assume a physical register file design where the previous value of each architected register survives in the register file until the physical register in question is returned to the free list. A physical register previously mapped to an architected register is returned to the free list when the instruction that defines the current mapping to the same architected register is committed. At this point, we read the value out of the register file and append it to the in-memory log entry for the current region (each log entry also contains a header that maps architected register names to the log contents). Similarly, as stores commit, the previous value is read from the cache at instruction commit and is written to the same in-memory log.

The minimal machine model requires very modest hardware changes: control logic for monitoring atomic region starts and commits and managing an in-memory log, an extra register file read port in the instruction commit stage, arbitration logic and datapath support to read old values from the cache at instruction commit, and support for performing additional cache writes at instruction commit.

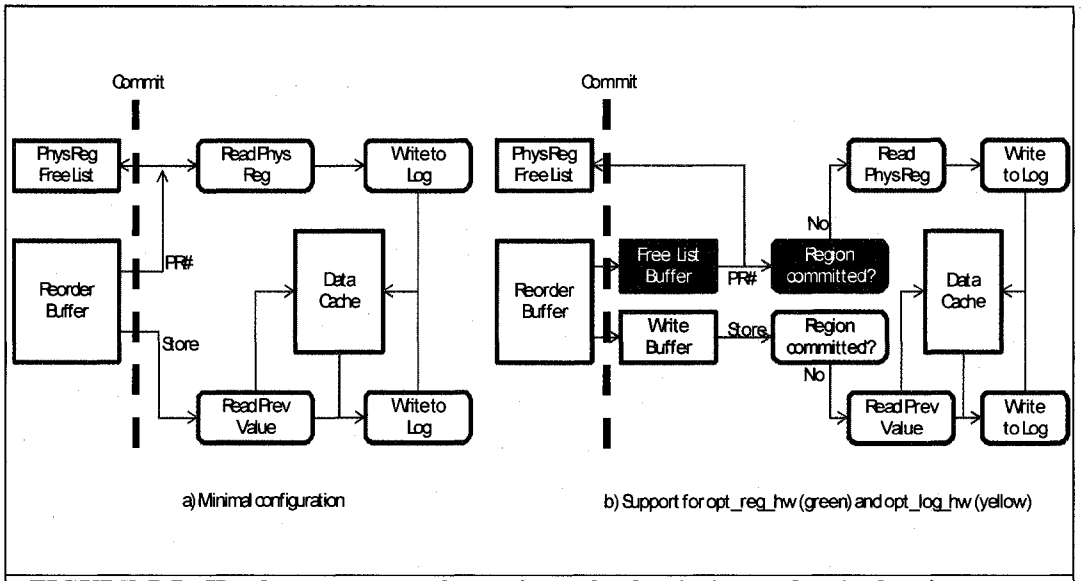


FIGURE 7-7. Hardware support for register checkpointing and write logging. The support is implemented in our simulated machine

As shown in Figure 7-7b, we also modeled the benefit of two simple optimizations to the hardware: one for register checkpointing and one for write logging. The opt_reg_hw configuration takes advantage of the fact that the physical register file often has sufficient buffering resources to avoid checkpointing registers to the in-memory log. In this configuration, the physical register free-list manager incorporates a free-list buffer that delays the freeing of registers that need to be checkpointed until the corresponding region commits. If the region commits, there is no need to log the old register values to the in-memory log, avoiding any overhead. However, if the free list becomes depleted, the processor can no longer make forward progress, so we force entries to retire from the free-list buffer. We also propose opt_log_hw, a similar optimization for write logging, which takes advantage of a pre-existing write buffer in our machine. Just as the free-list buffer attempts to delay checkpointing until the region commits, the write buffer delays the logging operation until the region commits. Of course, if the write buffer fills up, entries must be released as usual by committing them to the in-memory log.

The simulated machine parameters are listed in Table 7-4.

Table 7-4: Simulated machine parameters.

Out-of-order engine	4-wide fetch/issue/commit, 10-stage pipeline, physical register file with 64 entries, write buffer
Branch predictor	gshare, 4096-entry branch target table, 64K pattern history table entries, 32-entry RAS
Functional units	2 ALU(1 cycle), 1 multiplier (3 cycles) 1 L/S unit (1 cycle)
Memory system	L1 I-cache: 64KB, DM, 64B (1 cycle) L1 D-cache: 128KB, 2-way, 64B (2 cycle), 1 read port, 1 write port L2 Unified: 2MB, 8-way, 64B (10-cycle) Off-chip memory: 120-cycle latency

7.2.3 Results

7.2.3.1 Register checkpointing reduction

Calling conventions can significantly reduce register checkpointing while incurring very minimal cost. As shown in Figure 7-8, about 86% of the register checkpointing can be removed when both integer and floating point registers are considered. The effectiveness of calling convention depends on the number of pass-through parameters and whether the return register is used. In the best case where there are zero parameters and no return, almost 91% of register checkpointing can be removed. In the worst case where there are 2 parameters and 1 return allowed by the calling convention, about 81% of register checkpointing can be removed. Benchmarks typically fall into the range between the best and the worst cases.

Figure 7-9 shows the effectiveness of the physical register dirtiness analysis. The data points represent the percentage reductions among general purpose registers that calling convention deems to need checkpointing. On average about one third of such registers are not modified in type II regions and thus their checkpointing can be avoided. For db and jack, about half of such registers do not need checkpointing at all.

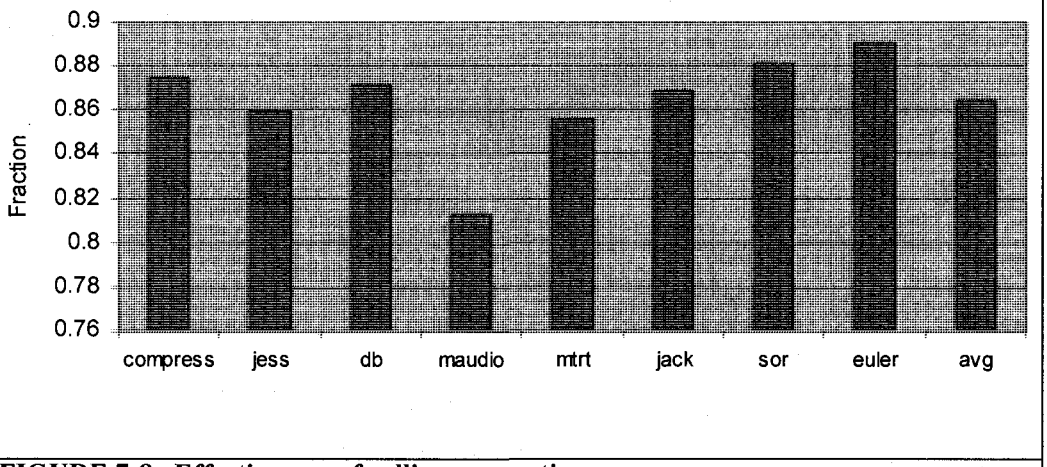


FIGURE 7-8. Effectiveness of calling convention.

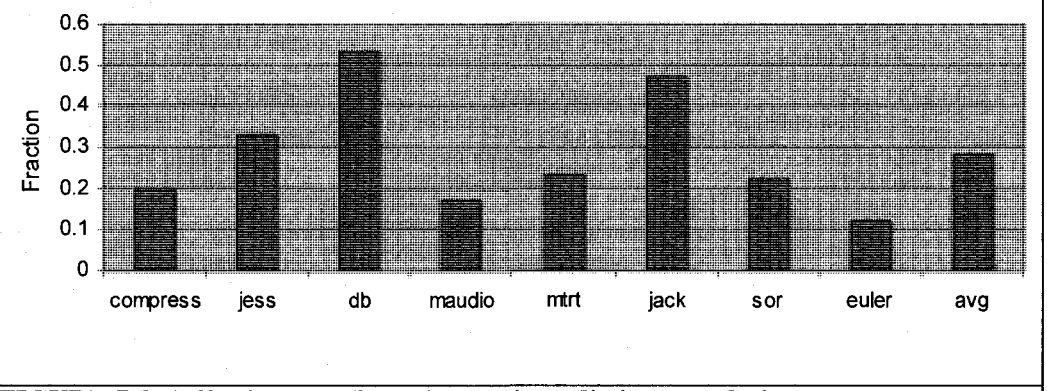


FIGURE 7-9. Effectiveness of physical register dirtiness analysis.

Figure 7-10 shows the percentages of general purpose registers that still need checkpointing after software analysis but can be avoided using a 16, 32, 64, 128, 256-entry instruction window with 24, 24, 32, 64, 128 physical registers respectively. In general, the percentages increase as the instruction window size increases except for sor. The benchmark sor only has a very large region and a few very small regions. This is why it is insensitive to the instruction window size. At a reasonable instruction window size of 128, we can remove about 69% of register checkpointing that can not be removed by software techniques.

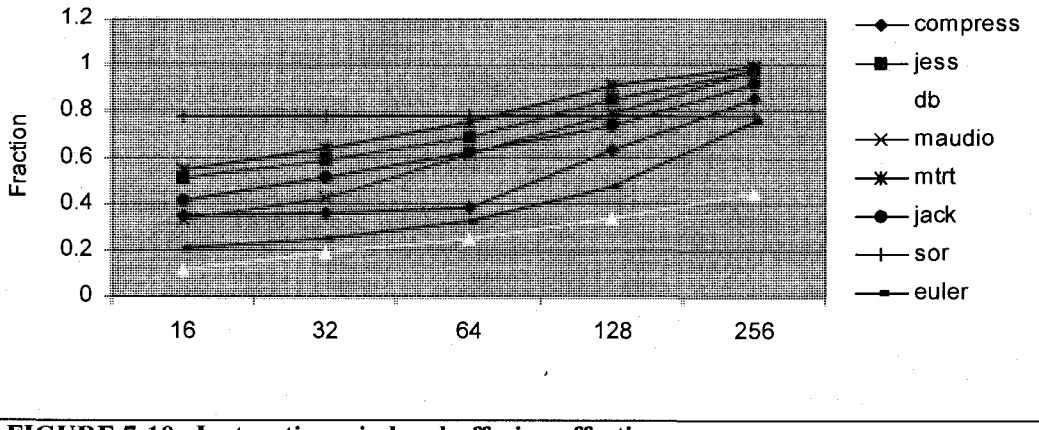


FIGURE 7-10. Instruction window buffering effectiveness.

Putting it all together, Figure 7-11 shows the percentages of register checkpointing that can be saved with software techniques and a 128-entry instruction window. The calling convention technique accounts for the majority of the savings while the other software techniques and hardware can further contribute to checkpointing savings. In the end, about 98% of total register checkpointing can be removed, which indicates that register checkpointing can incur almost no cost for our proposed hardware-software hybrid execution model.

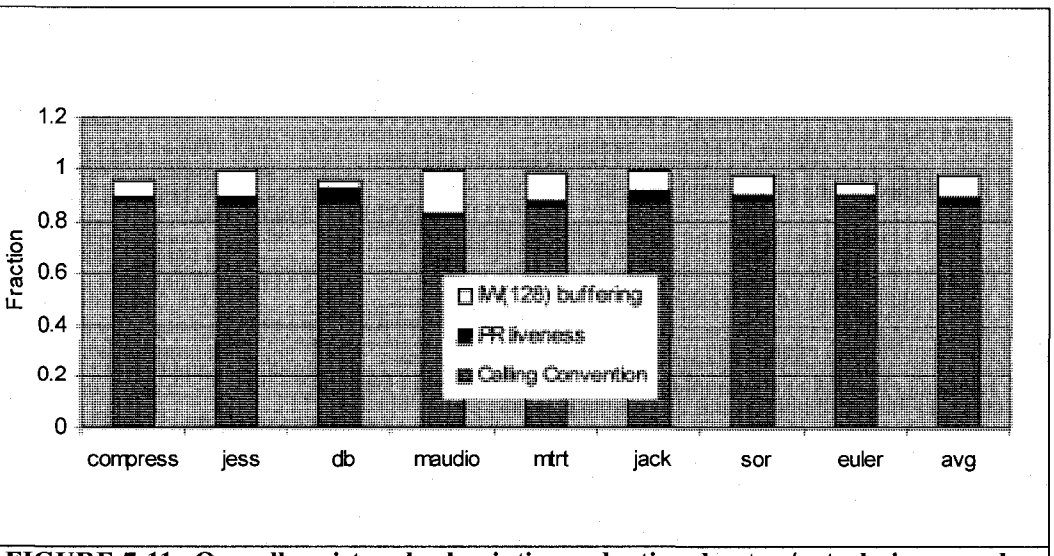


FIGURE 7-11. Overall register checkpointing reduction due to s/w techniques and a 128-entry IW.

Stack writes account for a large percentage of the total write traffic in Java applications, ranging from 65% to 87% as shown in our benchmarks in Figure 7-12. Therefore, techniques that can effectively reduce logging for stack writes are very important.

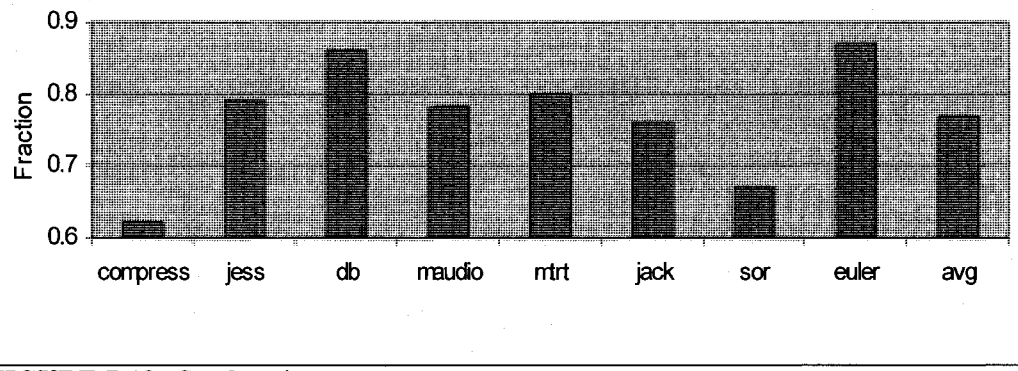


FIGURE 7-12. Stack write percentage.

Our stack write logging elimination algorithm can effectively remove unnecessary stack write logging as shown in Figure 7-13. Across the benchmark suite, a majority of stack write logging can be removed for 5 out of 8 benchmarks; about half of stack write logging can be removed for compress and mtrt. The only benchmark not performing well is db, which has many stack writes behind calls. In sor, there is a giant three-level nested loop, which accounts for most of the stack writes. In the code, a function call is inserted before the loop to start a timer and after the loop to stop the timer. If no such timer start/stop function existed, the loop would be in a leaf function, i.e. the stack write logging could be safely removed. This is why 98% of the stack write logging can be removed. Otherwise, if a compiler can recognize very large loops and allocate/deallocate stacks before/after them, stack write logging in such large loops can be safely removed.

The heap write logging elimination algorithm can remove a fair portion of the heap

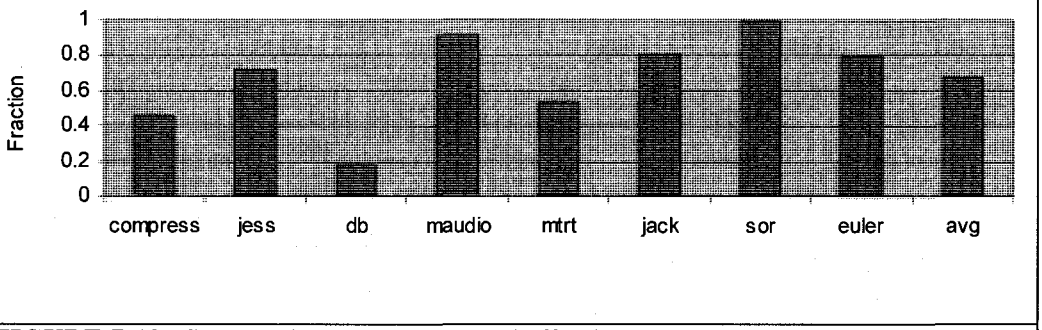


FIGURE 7-13. Stack write logging removal effectiveness.

write logging operations. As shown in Figure 7-14, the algorithm can remove more than 50% for one benchmark (mtrt) and more than 40% for three benchmarks (jess, maudio, and jack). The algorithm is moderately successful in removing logging for compress, db, and euler. In the benchmark Sor, there is a very large three-level loop that performs many iterations on a large 2D matrix. In the loop, reads could precede writes for matrix elements operations. This is why almost no heap write logging is removed for Sor.

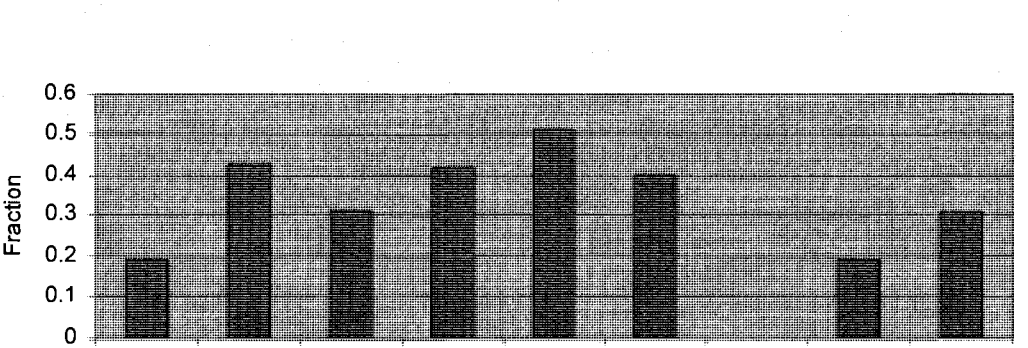


FIGURE 7-14. Heap write logging removal effectiveness.

Figure 7-15 shows the percentages of write logging that can be removed by a write buffer with 16, 32, 64, and 128 entries. A write buffer is effective for small regions and can achieve great savings for compress, jess, mtrt, and jack. However, at the size of 32 and

64 a write buffer can only save on average 49% and 56%. With the addition of software techniques, a write buffer can become much more efficient. As shown in Figure 7-16, its effectiveness is greatly improved. At the size of 32 and 64, the write buffer, combined with the software techniques, can remove on average 80% and 84% of the total write logging. For a few benchmarks such as jess, maudio, mtrt, and jack, the savings are well above 90% with a 64-entry write buffer. This buffer size could achieve above 80% savings for compress and euler. Benchmarks db and sor do not perform as well as the rest. Benchmark db is write intensive. Benchmark sor has a giant three-level loop in a single region that generates a lot of heap write traffic. Neither the software heap write logging removal nor the write buffer works effectively. However, the stack write logging removal algorithm works very well here.

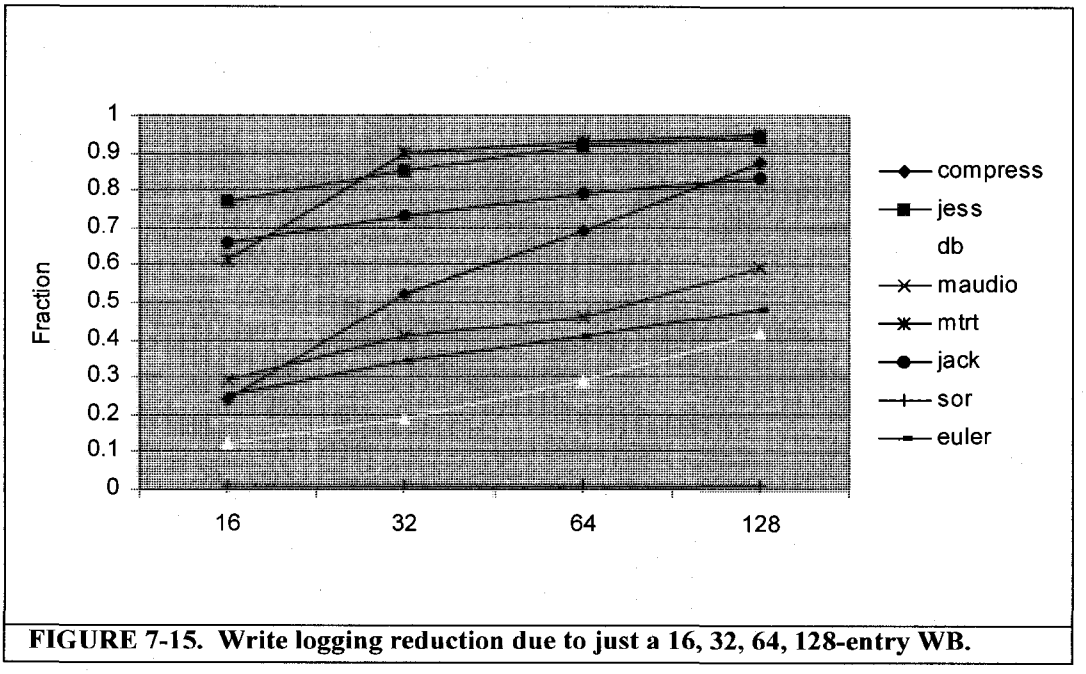


FIGURE 7-15. Write logging reduction due to just a 16, 32, 64, 128-entry WB.

7.2.3.3 Region shrinking

We only examine region shrinking's benefits on write logging reduction since our

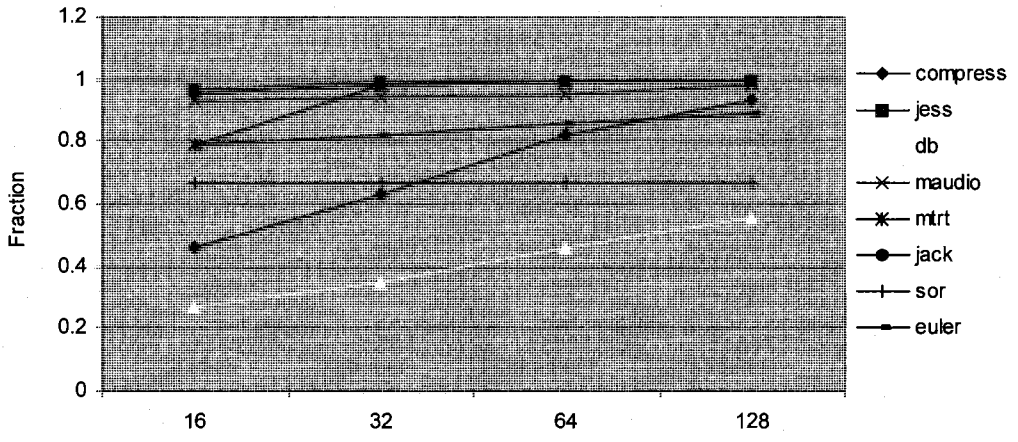


FIGURE 7-16. Write logging reduction due to software and a 16, 32, 64, 128-entry WB.

set of software and hardware techniques can already reduce register checkpointing very effectively. We set our criterion for turning on region shrinking as very large (account for more than 50% of total program writes), busy (account for more than 50% of program execution time) loops. Profile information tells us that only one loop in sor and one loop in db fall into this category.

Figure 7-17 breaks down write logging removed into contributions from the stack write logging removal, the heap write logging removal, a 64-entry write buffer, and region shrinking. Region shrinking is only activated for db and sor; accordingly, their write logging savings improve greatly to 95% and 99% respectively. Across the benchmark suite, the average saving has improved from 84% to 94%.

7.2.3.4 Performance

Our techniques of reducing hardware cost in support of atomic regions can minimize the detrimental effect on the program execution time and preserve the performance

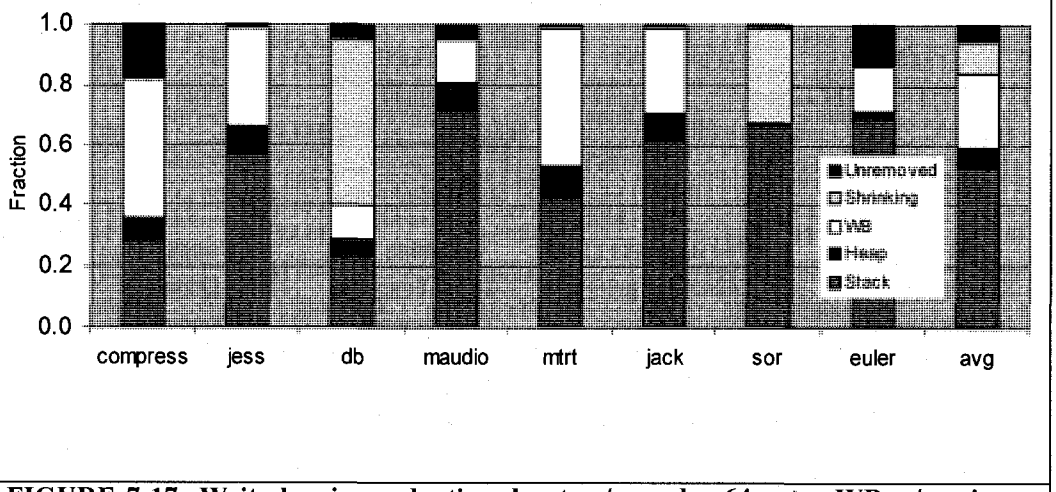


FIGURE 7-17. Write logging reduction due to s/w and a 64-entry WB w/ region shrinking.

gain due to the application of speculative optimizations. As shown in Figure 7-18, speculative optimizations can achieve on average a speedup of 14.2% without the consideration of any hardware cost. The hardware cost without any cost reduction optimization will slow down the benchmarks by an average of 6.3%. The adoption of register checkpointing techniques can bring the program performance back to the baseline while the further reduction of write logging can achieve an average speedup of 13.5%, which is very close to the speedup in the perfect case. The write logging removal can recover a larger performance loss than the register checkpointing removal (13.5% vs. 6.3%). Therefore, it is more important to study write logging removal techniques to reduce the overall logging operations.

We also compare software with hardware techniques with respect to their impact on program execution time. Figure 7-19 shows the program execution time by the application of software, hardware, and both register checkpointing reduction techniques to the minimal hardware. We can see clearly that software techniques have an advantage against

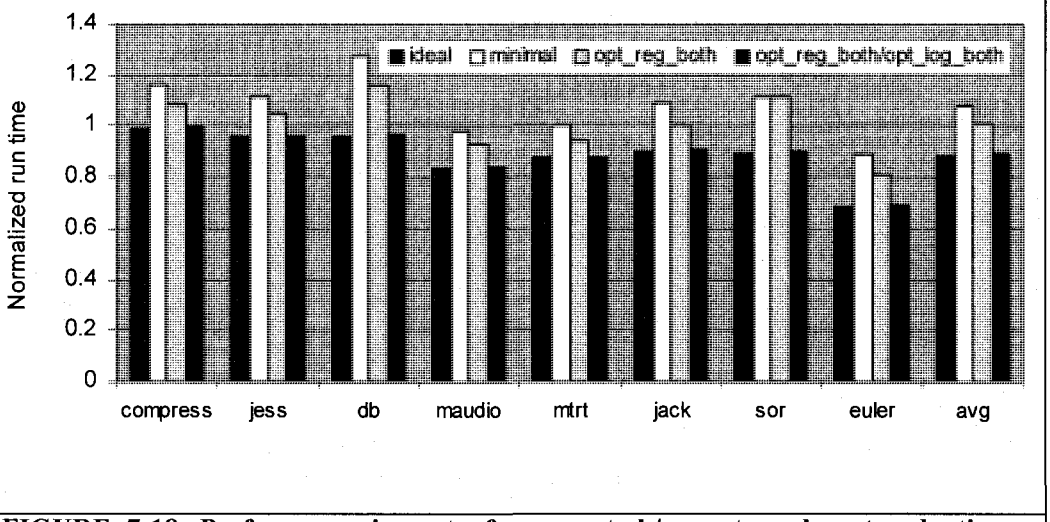


FIGURE 7-18. Performance impact of spec opt, h/w cost, and cost reduction techniques. Our baseline is no spec opt and minimal hardware (we use this as the baseline throughout this section). Ideal is spec opt + perfect hardware; minimal is spec opt + minimal hardware; opt_reg_both is spec opt + s/w reg opt + h/w reg opt (128-entry IW); opt_reg_both/opt_log_both is opt_reg_both + s/w log opt + h/w reg opt (64-entry write buffer). In opt_reg_both/opt_log_both, region shrinking is applied.

hardware techniques as opt_reg_sw's average execution time is very close to opt_reg_both's with a slowdown of only 0.5%. The hardware-only approach has a slowdown of more than 2% compared with the opt_reg_both case. Furthermore, we apply software, hardware, and both write logging techniques to opt_reg_both to study the effectiveness of these three methods. The results are shown in Figure 7-20. For write logging elimination, there is no clear winner. Both software and hardware can contribute. The software-only and hardware-only approaches cause slowdowns of more than 2% compared with the hybrid approach.

7.3 Summary

The early native machine experiments lead to a quick verification that our speculative execution model is useful and can lead to the design of lightweight, performance-

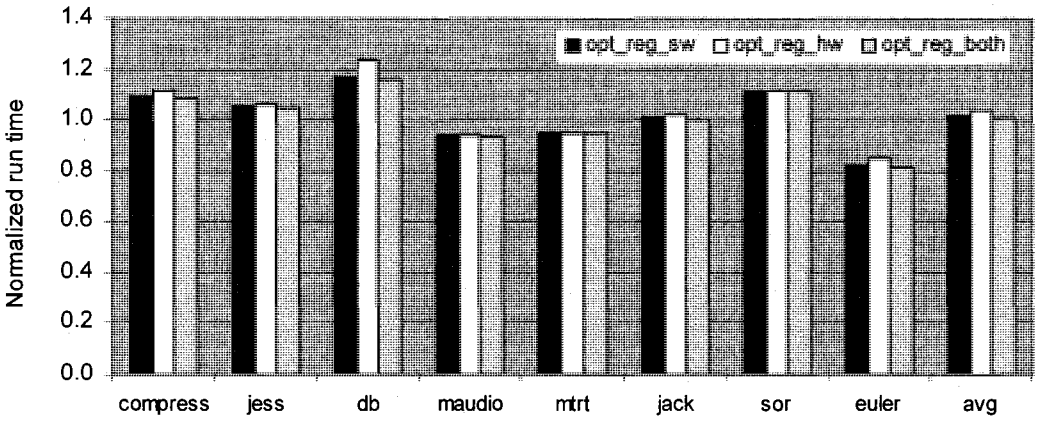


FIGURE 7-19. Performance impact on minimal hardware from register checkpointing reduction. Here, opt_reg_sw means only software techniques are applied to the minimal hardware to reduce register checkpointing; opt_reg_hw only uses hardware techniques; opt_reg_both uses both techniques.

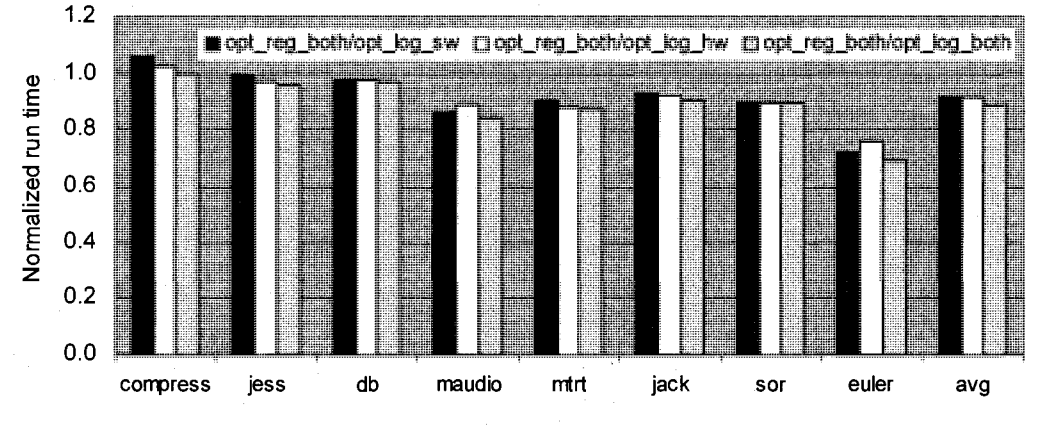


FIGURE 7-20. Impact on program execution time due to write logging elimination. Starting from opt_reg_both, we look at the contributions to program execution time from software, hardware, and both techniques for write logging removal. Opt_reg_both/opt_log_sw is opt_reg_both + s/w log opt; opt_reg_both/opt_log_hw is opt_reg_both + h/w log opt; opt_reg_both/opt_log_both is opt_reg_both + both.

enhancing compiler optimizations. Next, we addressed the detailed hardware design and the hardware atomicity support problem with the trace-driven simulation methodology.

We found that the impact of hardware atomicity support cost can be minimized by the set of hybrid cost reduction techniques and we can preserve 95% of the performance gain due to speculative optimizations. 113

Chapter 8

Conclusion

The fast increasing transistor budget for microprocessors and the complexity of large software systems are pushing the computer system design into a pivotal turning point. Microprocessor designers are facing a huge challenge -- how to efficiently use the enormous hardware resources available. At the same time, software designers are striving to make software fast yet easy to build, maintain, and use. The software engineering principals applied to large software system design unavoidably lead to some slowdowns in such systems.

A natural solution from a microprocessor designer's perspective is to co-design the microprocessor with software planning early in the overall computer system design. Following this design principal, this thesis looks at the popular Java programming language, the design of Java Virtual Machines (JVMs), and the interactions between them and a microprocessor.

More specifically, the Java programming language introduces a precise exception model to make it a safe, easy to maintain, and easy to use programming language. Unfortunately, the precise exception model could cause optimization constraints for an optimizing compiler in a JVM and thus hurt Java application performance. We first propose a speculative execution model that exposes atomic guarded regions in hardware to the JVM and allows the JVM to optimize the code speculatively within these atomic regions.

Under the speculative execution model, we investigated the effectiveness of several static atomic region formation heuristics: the leaf function based discrete region

placement heuristic, the caller/callee continuous region placement heuristic, and the caller/callee/innermost-loop continuous region placement heuristic. We found that the static heuristics could achieve the performance very close to that of perfect region placement (where all the possible speculation benefits could be reaped).

In addition, we successfully designed a few simple, efficient, yet powerful optimizations that could help remove the optimization constraints imposed by the Java precise exception model. Namely, we designed a speculative null pointer check elimination algorithm, a speculative SSA-based local array bounds check elimination algorithm, and a speculative loop-based global array bounds check elimination algorithm. Our algorithms could speculatively eliminate null pointer checks and array bounds checks early in the optimization flow of an optimizing compiler. Experiments show that we could improve the studied benchmark suite performance by an average of 15.9% at the expense of the increase of less than 1% overall compile time on a native machine when atomicity supporting hardware cost is not considered.

We also looked into a variety of ways of reducing additional hardware cost in support of atomic region based execution. Atomic execution requires register checkpointing and write logging to maintain the execution state upon the entrance of an atomic region. In case of a rollback, the original register values and memory state must be restored. For register checkpointing, we exploit register calling convention (SW), register liveness analysis (SW), and the physical register file free-list buffer (HW). For write logging reduction, we use a stack write logging elimination algorithm (SW), a heap write logging elimination algorithm (SW), region shrinking (SW), and write buffering (HW). With these techniques, we could remove on average 98% of register checkpointing and 94% of write logging for

the benchmark suite we studied. The reduction of hardware cost helps preserve 95% of the performance gain due to speculative optimizations. With the speculative optimizations and the atomicity hardware cost reductions, we show an average of 14% speedup compared with a baseline where no speculative optimization and no atomicity hardware cost are involved.

8.1 Future Work

Our future work mainly involves two fronts of efforts. The first one is within our speculative execution model. We look for more speculative optimization opportunities, i.e. designing other lightweight yet efficient speculative algorithms. We would like to go beyond just the relaxation of the Java precise exception constraints. A possible target may be the internet security issues and their impact on performance optimization. Another possible target may be bugs and their impact on performance optimization (especially when runtime monitoring is applied to bug detection). We would also like to investigate some dynamic region formation heuristics based on profiling information. More feasible techniques to reduce hardware atomicity cost is also something on our list.

The second front of efforts will be the overall picture of hardware-software co-design, i.e. how we can apply our current idea to more design cases between microprocessors and compilers (dynamic and static), even other software systems such as operating systems. We will investigate how to better partition the work between hardware and software and how to better utilize the ample onchip resources to further enable software system performance improvement.

References

[1] A.-R. Adl-Tabatabai, J. Bharadwaj, D.-Y. Chen, A. Ghuloum, V.S. Menon, B.R. Murphy, M. Serrano, and T. Shpeisman. The starjit compiler: a dynamic compiler for managed runtime environments. *Intel Technology Journal*, 7(1), Feb 2003.

[2] A.-R. Adl-Tabatabai, B.T. Lewis, V. Menon, B.R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2006.

[3] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.

[4] H. Akkary, R. Rajwar, and S.T. Srinivasan. Towards scalable large instruction window processors. In *Proceedings of the 36th International Symposium on Microarchitecture*, Dec 2003.

[5] B. Alpern, C.R. Attanasio, J.J. Barton, M.G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S.J. Fink, D. Grove, M. Hind, S.F. Hummel, D. Liever, V. Litvinov, M.F. Mergen, T. Ngo, J.R. Russell, V. Sarkar, M.J. Serrano, J.C. Shepherd, S.E. Smith, V.C. Sreedhar, H. Srinivasan, and J. Whaly. The jalapeno virtual machine. *IBM Systems Journal*, 39(1), 2000.

[6] C.S. Ananian, K. Asanoic, B.C. Kuszmaul, C.E. Leiserson, and S. Lie. Unbounded transactional memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, Feb 2005.

[7] M. Arnold, S. Fink, D. Grove, M. Hind, and P.F. Sweeney. Adaptive optimization in the jalapeno jvm. In *Proceedings of ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications*, Oct 2000.

[8] J.M. Asuru. Optimization of array subscript range checks. *ACM Letters on Programming Languages and Systems*, 1(2), June 1992.

[9] J. Auslander, M. Philiipose, C. Chambers, S.J. Eggers, and B.N. Bershad. Fast, effective dynamic compilation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun 1996.

[10] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2000.

[11] BEA. Bea jrockit 6. http://www.bea.com/jrockit6.

[12] R. Berrendorf, H. Ziegler, and B. Mohr. The performance counter library. ht-

tp://www.fz-juelich.de/jsc/PCL/.

[13] S.M. Blackburn, R. Garner, C. Hoffmann, A.M. Khang, K.S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S.Z. Guyer, M. Hirzel, A.L. Hosking, M. Jump, H.B. Lee, J.E.B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D.V. Dincklage, and B. Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Oct 2006.

[14] J. Bobba, K.E. Moore, H. Volos, L. Yen, M.D. Hill, M.M. Swift, and D.A. Wood. Performance pathologies in hardware transactional memory. In *Proceedings of International Symposium on Computer Architecture*, Jun 2007.

[15] R. Bodik, R. Gupta, and V. Sarkar. Abcd: Eliminating array bounds checks on demand. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2000.

[16] M. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Shreedhar, H. Srinivasan, and J. Whaley. The jalapeno dynamic optimizing compiler for Java. In *ACM Java Grande Conference*, June 1999.

[17] B.D Carlstrom, J. Chung, H. Chafi, A. McDonald, C.C. Minh, L. Hammond, C. Kozyrakis, and K. Olukotun. Transactional execution of Java programs. In *Proceedings of Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, Oct 2005.

[18] B.D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. Minh, C. Kozyrakis, and K. Olukotun. The atomos transactional programming language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2006.

[19] G.J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1982.

[20] C. Chambers. The design and implementation of self compiler and optimizing compiler for object-oriented programming languages. PhD Thesis, Stanford University, Mar 1992.

[21] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for self, a dynamically-typed object-oriented programming languages. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, Jun 1989.

[22] A. Chang and M. Mergen. 801 storage: Architecture and programming. *ACM Transactions on Computer Systems*, 6(1), Feb 1988.

[23] J. Chung, W. Baek, N.G. Bronson, J. Seo, C. Kozyrakis, and K. Olukotun. Ased: Availability, security, and debugging support using transactinal memory. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Jun 2008.

[24] J. Chung, H. Chafi, C.C. Minh, A. McDonald, B.D. Carlstrom, C. Kozyrakis, and K. Olukotun. The common case transactional behavior of multithreaded programs. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, Feb 2006.

[25] C. Click and M. Paleczny. A simple graph-based intermediate representation. In *Proceedings of ACM SIGPLAN Workshop on Intermediate Representations*, Jan 1995.

[26] K. Cooper and L. Torczon. *Engineering a Compiler (1st Edition)*. Morgan Kaufmann, 2003.

[27] A. Cristal, D. Ortega, J. Llosa, and M. Valero. Out-of-order commit processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, Feb 2003.

[28] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct 2006.

[29] J.C. Dehnert, B.K. Grant, J.P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the First Annual IEEE/ACM International Symposium on Code Generation and Optimization*, Mar 2003.

[30] D.R. Engler and T.A. Proebsting. Dcg: An efficient, retargetable dynamic code generation system. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Oct 1994.

[31] S. Fink, K. Knobe, and V. Sarkar. Unified analysis of array and object reference in strongly typed languages. In *Proceedings of Annual Symposium on Static Analysis*, June 2000.

[32] J. Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java(TM) Language Specification (3rd Edition)*. Prentice Hall, 2005.

[33] Java Grande. The Java grande forum benchmark suite. http://www.epcc.edu.ac.uk/javagrande/javag.html.

[34] B. Grant, M. Philipose, M. Mock, C. Chambers, and S.J. Eggers. An evaluation of staged run-time optimizations in dyc. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun 1999.

[35] J. Gray. The transaction concept: Virtues and limitations. In *Proceedings of Very Large Databases*, Sep 1981.

[36] M. Gupta, J.-D. Choi, and M. Hind. Optimizing Java programs in the presence of exceptions. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, Jun 2000.

[37] R. Gupta. A fresh look at optimizing array bound checking. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun 1990.

[38] R. Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 1(4), 1994.

[39] L. Hammond, V. Wong, M. Chen, B.D. Carlstrom, J.D. Davis, B. Hertzberg, M.K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, Jun 2004.

[40] Harmony. Harmony dynamic runtime layer virtual machine. http://harmony.apache.org/subcomponents/drlvm.

[41] T. Harris, A. Cristal, O.S. Unsal, E. Ayguade, F. Gagliardi, B. Smith, and M. Valero. Transactional memory: An overview. *IEEE Micro Special Issue: Hot Tutorials*, May 2007.

[42] T. Harris and K. Fraser. Language support for light-weight transactions. In *Proceedings of the 18th SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, Oct 2003.

[43] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transaction. In *Proceedings of International Symposium on Principles and Practice of Parallel Programming*, Mar 2005.

[44] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design Implementation*, Jun 2006.

[45] W.H. Harrison. Compiler analysis for the value ranges of variables. *IEEE Transactions on Software Engineering*, May 1977.

[46] K. Hazelwood and D. Grove. Adaptive online context-sensitive inlining. In *Proceedings of International Symposium on Code Generation and Optimization*, Mar 2003.

[47] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach: Third Edition*. Morgan Kaufmann, 2006.

[48] M. Herlihy and J.E.B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993.

[49] S. Hu. Efficient binary translation in co-designed virtual machines. PhD Thesis, University of Wisconsin - Madison, 2006.

[50] S. Hu and J.E. Smith. Using dynamic binary translation to fuse dependent instructions. In *Proceedings of International Symposium of Code Generation and Optimization*, Mar 2004.

[51] S. Hu and J.E. Smith. An approach for implementing efficient superscalar cisc processors. In *Proceedings of International Symposium on High-Performance Computer Architecture*, Feb 2006.

[52] S. Hu and J.E. Smith. Reducing startup time in co-designed virtual machines. In *Proceedings of International Symposium on Computer Architecture*, Jun 2006.

[53] J. Hunter and W. Crawford. *Java Servlet Programming.* O'Reilly Media, Inc., 2001.

[54] Intel. Open runtime platform. http://orp.sourceforge.net.

[55] Intel. The vtune performance analyzer. http://www.intel.com/cd/software/products/asmo-na/eng/vtune.

[56] Kazuaki Ishizaki, Mikio Takeuchi, Kiyokuni Kawachiya, Toshio Suganuma, Osamu Gohda, Tatsushi Inagaki, Akira Koseki, Kazunori Ogata, Motohiro Kawahito, Toshiaki Yasue, Takeshi Ogasawara, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Effectiveness of cross-platform optimizations for a Java just-in-time compiler. In *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Languages, Systems and Applications*, pages 187–204, 2003.

[57] B.W. Kernighan and D.M. Ritchie. *The C Programming Language (2nd Edition).* Prentice Hall, 1988.

[58] T. Knight. An architecture for mostly functional languages. In *Proceedings of the ACM Conference on LISP and Functional Programming*, Jun 1986.

[59] P. Kolte and M. Wolfe. Elimination of redundant array subscript range checks. *ACM SIGPLAN Notices*, 30(6), Jun 1995.

[60] S. Kumar, M. Chu, C.J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *Proceedings of the 11th Symposium on Principles and Practice of Parallel Programming*, Mar 2006.

[61] T. Lindholm and F. Yellin. *The Java(TM) Virtual Machine Specification (2nd Edition).* Addison-Wesley Professional, 1999.

[62] C. Luk, R. Cohn an dR. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of Conference on Programming Language and Implementation*, Jun 2005.

[63] V. Markstein, J. Cocke, and P. Markstein. Optimization of range checking. In *Proceedings of Symposium on Compiler Optimization*, Jun 1982.

[64] R. Marlet, C. Consel, and P. Boinot. Efficient incremental run-time specialization for free. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun 1999.

[65] J.F. Martinez, J. Renau, M.C. Huang, M. Prvulovic, and J. Torrelas. Cherry: Checkpointed early resource recycle in out-of-order microprocessors. In *Proceedings of the 35th International Symposium on Microarchitecture*, Nov 2002.

[66] A. McDonald, J. Chung, B.D. Carlstrom, C.C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural semantics for practical transactional memory. In *Proceedings of the 33rd International Symposium on Computer Architecture*, Jun 2006.

[67] Microsoft. The common language runtime. http://msdn.microsoft.com/netframework/programming/clr.

[68] C.C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th International Symposium on Computer Architecture*, Jun 2007.

[69] K.E. Moore, J. Bobba, M.J. Moravan, M.D. Hill, and D.A. Wood. Logtm: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High Performance Computer Architecture*, Feb 2006.

[70] M.J. Moravan, J. Bobba, K.E. Moore, L. Yen, M.D. Hill, B. Liblit, M.M. Swift, and D.A. Wood. Supporting nested transactional memory in logtm. In *Proceedings of the 10th Conference on Architectural Support for Programming Languages and Operating Systems*, Oct 2006.

[71] E. Moreira, S. P. Midkiff, and M. Gupta. From flop to megaflops: Java for technical computing. *ACM Trans. Program. Lang. Syst.*, 22(2):265–295, 2000.

[72] G.C. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun 1998.

[73] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles. Hardware atomicity for reliable software speculation. In *Proceedings of the 34th Annual International*

Sumposium of Computer Architecture, Jun 2007.

[74] N. Njoroge, J. Casper, S. Wee, Y. Teslyar, D. Ge, C. Kozyrakis, and K. Olukotun. Atlas: A chip-multiprocessor with transactional memory support. In *Proceedings of the Conference on Design Automation and Test in Europe (DATE)*, Apr 2007.

[75] M. Paleczny, C. Vick, and C. Click. The Java hotspot server compiler. In *Proceedings of the USENIX Symposium on Java Virtual Machine Research and Technology*, Apr 2001.

[76] S.J. Patel and S.S. Lumetta. A hardware framework for dynamic optimization. In *Proceedings of International Symposium on Microarchitecture*, Dec 2000.

[77] J.R.C. Patterson. Accurate static branch prediction by value range propagation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun 1998.

[78] M. Poletto, D. Engler, and M.F. Kaashoek. tcc: A system for fast, flexible, and high-level dynamic code generation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun 1999.

[79] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5), 1999.

[80] R.S. Pressman and R. Pressman. *Software Engineering: A Practioner's Approach (6th Edition)*. McGraw-Hill Science/Engineering/Math, 2004.

[81] R. Rajwar and J.R. Goodman. Speculative lock elision: Enabling highly cncurrent multithreaded execution. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, Dec 2001.

[82] R. Rajwar and J.R. Goodman. Transactional lock-free execution of lock-based programs. In *In Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Dec 2002.

[83] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, Jun 2005.

[84] R. Rugina and M. Rinard. Automatic parallelization of divide and conquer algorithms. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1999.

[85] B. Saha, A.-R. Adl-Tabatabai, R.L. Hudson, C.C. Minh, and B. Hertzberg. Mcrt-stm: A high performance software transactional memory system for a multi-core runtime. In *Proceedings of International Symposium on Principles and Practice of Parallel Programming*, Mar 2006.

[86] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural support for software transactional memory. In *Proceedings of International Symposium on Microarchitecture*, Dec 2006.

[87] D. Sanchez, L. Yen, M.D. Hill, and K. Sankaralingam. Implementing signatures for transactional memory. In *Proceedings of International Symposium on Microarchitecture*, Dec 2007.

[88] S. Sastry. Techniques for transparent program specialization in dynamic optimizers. PhD thesis, University of Wisconsin, 2003.

[89] J.P. Shen and M.H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill, 2005.

[90] J.E. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, 2005.

[91] SPEC. Specjvm98. http://www.spec.org/jvm98, 1998.

[92] M. Spezialetti and R. Gupta. Loop monotonic statements. *IEEE Transactions on Software Engineering*, 21(6), Jun 1995.

[93] L. Su and M.H. Lipasti. Speculative optimizations using hardware-monitored guarded regions for Java virtual machines. In *Proceedings of the 3rd SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, Jun 2007.

[94] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A dynamic optimization framework for a Java just-in-time compiler. In *Proceedings of the 16th SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, Oct 2001.

[95] N. Suzuki and K. Ishihata. Implementation of an array bound checker. In *Proceedings of 4th ACM Symposium on Principles of Programming Languages*, Jan 1977.

[96] Z. Xu, B. Miller, and T. Reps. Safety checking of machine code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun 2000.

[97] L. Yen, J. Bobba, M.R. Marty, K.E. Moore, H. Volos, M.D. Hill, M.M. Swift, and D.A. Wood. Logtm-se: Decoupling hardware transactional memory from caches. In *Proceedings of International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2007.