© 2012 Daniel Ahn

SOFTWARE AND ARCHITECTURE SUPPORT FOR THE BULK MULTICORE

BY

DANIEL AHN

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Doctoral Committee:

Professor Josep Torrellas, Chair
Calin Cascaval, Qualcomm Research
Professor Samuel Midkiff, Purdue University
Professor Wen-mei Hwu
Professor Vikram Adve
Assistant Professor Sam King

Abstract

Research on transactional memory began as a tool to improve the experience of programmers working on parallel code. Just as transactions in databases, it was the job of the runtime to detect any conflicts between parallel transactions and rollback the ones that needed to be re-executed, leaving the programmers blissfully unaware of the communication and synchronization that needs to happen. The programmer only needed to cover the sections of code that might generate conflicts in transactions, or atomic regions.

More recently, new uses for transactional execution were proposed where, not only were user specified sections of code executed transactionally but the entire program was executed using transactions. In this environment, the hardware is in charge of generating the transactions, also called chunks, unbeknownst to the user. This simple idea led to many improvements in programmability such as providing a sequentially consistent(SC) memory model, aiding atomicity violation, enabling deterministic reply, and even enable deterministic execution. However, the implications of this chunking hardware to the compiler layer has not been studied before, which is the subject of this Thesis.

The Thesis makes three contributions. First, it describes the modifications to the compiler necessary to enable the benefits in programmability, specifically SC memory model, to percolate up to the programmer language level from the hardware level. The already present hardware support for chunked execution is exposed to the compiler and used extensively for this purpose. Surprisingly, the ability to speculate using chunks leads to speedups over traditional compilers in many cases. Second, it describes how to expose hardware signatures, present in chunking hardware for the purposes of conflict detection and memory disambiguation, to the compiler to enable further

novel optimizations. An example is given where hardware signatures are used to summarize the side-effects of functions to perform function memoization at a large scale. Third, it describes how to use atomic regions and conflict detection hardware to improve alias analysis for general compiler optimizations using speculation. Loop Invariant Code Motion, a widely used traditional compiler pass, is run as an example client pass to test the potential of the new alias analysis.

Table of Contents

List of Tables . . . . **vii**

List of Figures . . . . **viii**

Chapter 1 Introduction **1**
1.1 Background on BulkSC 4
1.1.1 Overview 4
1.1.2 Implementation 5
1.2 Summary 7

Chapter 2 High-Performance Sequential Consistency through Cooperative Compiler and Hardware Support **8**
2.1 Introduction 8
2.1.1 Benefits of Supporting SC 9
2.1.2 Goal and Contributions 9
2.2 Background 11
2.2.1 Algorithm for Generating Chunks 11
2.2.2 Compiler-Driven Enforcement of SC 11
2.3 Compiler for SC 13
2.3.1 Main Ideas 13
2.3.2 New Instructions Added 16
2.3.3 Difference to Transactional Memory 18
2.3.4 Safe Version of the Atomic Region Code 19
2.4 Algorithm Design 21
2.4.1 Inserting Atomic Regions 21
2.4.2 Lock Compression and Region Nesting 24
2.4.3 Visibility with Synchronizations 24
2.4.4 Visibility with Data Races 25
2.5 Experimental Setup 26
2.5.1 Compiler and Simulator Infrastructure 26
2.5.2 Experiments and Applications 27
2.6 Evaluation 28
2.6.1 Understanding the Optimizations Enabled 28
2.6.2 Simulated Speedups 30
2.6.3 Characterizing the Transformations 33

2.7 Discussion 34
2.8 Related Work 34
2.8.1 Software-Only Sequential Consistency 34
2.8.2 Exploiting Support for Atomicity 35
2.9 Conclusions 36

Chapter 3 Software-Exposed Signatures for Code Analysis and Optimization 37
3.1 Introduction 37
3.2 Background on Memoization 39
3.3 Idea: Exposing Signatures to Software 39
3.3.1 Basic Idea 39
3.3.2 Examples 40
3.3.3 Design Overview and Guidelines 42
3.4 SoftSig Software Interface 45
3.4.1 The Signature Register File (SRF) 46
3.4.2 Collection 46
3.4.3 Disambiguation 46
3.4.4 Persistence, Signature Status, and Exceptions 47
3.4.5 Signature Manipulation 49
3.4.6 Interaction with Checkpointing 49
3.4.7 Managing Signature Registers 49
3.5 SoftSig Architecture 50
3.5.1 SoftSig Instruction Execution 50
3.5.2 Signature Register File 51
3.5.3 Allocation and Deallocation 52
3.5.4 Collection and Local Disambiguation 53
3.5.5 Remote Disambiguation 53
3.5.6 Exceptions 56
3.6 MemoiSE: Signature-Enhanced Memoization 56
3.6.1 A General Memoization Framework 57
3.6.2 The MemoiSE Algorithm 59
3.6.3 Optimizations for Lower Overhead 61
3.7 Evaluation 62
3.7.1 Experimental Setup 62
3.7.2 Impact of MemoiSE 63
3.7.3 Function Characterization 65
3.8 Related Work 68
3.8.1 Signatures & Bloom Filters 68
3.8.2 Memoization 69
3.9 Conclusion 70

Chapter 4 Alias Speculation Using Atomic Region Support 72
4.1 Introduction 72
4.2 Alias Speculation 75
4.2.1 Basic Idea 75
4.2.2 Compiler Transformations 76
4.2.3 ISA Extensions 78
4.2.4 Example of LICM using SAA 79
4.3 Implementation 80
4.3.1 The Cost-Benefit Model 80
4.3.2 Compiler Support 82
4.4 Evaluation 83
4.4.1 Experimental Setup 83
4.4.2 Optimization Examples 85
4.4.3 Experimental Results 89
4.5 Discussion 96
4.6 Related Work 100
4.6.1 Optimizations using Atomic Regions 100
4.6.2 Alias Analysis 100
4.7 Conclusions 101

Chapter 5 Conclusion 103

References 104

List of Tables

2.1 Events that affect chunk generation. 12
2.2 Instructions added so that the compiler manages the chunking. 14
2.3 Characterizing the dynamic behavior of the code transformed by BulkCompiler. AR stands for Atomic Region. 32

3.1 Design guidelines in SoftSig. 42
3.2 SoftSig software interface. 45
3.3 Parameters of the architecture simulated. 62
3.4 Applications studied. 63
3.5 Environments analyzed. 63
3.6 Five functions that are frequently memoized from the different applications. 67

4.1 Extensions to the ISA to enable SAA. 79
4.2 Parameters of the architecture simulated. 84
4.3 Analysis of examples shown in Figure 4.4 and Figure 4.5. 85
4.4 Average instruction statistics per atomic region for each benchmark. 97
4.5 Hardware resources required per atomic region for each benchmark. 98

List of Figures

1.1 The BulkSC architecture. 5
1.2 Bloom filter signatures. 6

2.1 Compiler-driven chunking for high performance SC. In the figure, each i_j represents a set of instructions. 12
2.2 Transforming a large code section. In the figure, each i_j represents a set of instructions. 17
2.3 Algorithm that inserts atomic regions in a method. 22
2.4 Examples of chunks with synchronization operations. 23
2.5 Speedups of *BulkCompiler_1* over *Baseline_1* (a), and of *BulkCompiler_20* over *Baseline_20* (b). 31

3.1 Three examples of how to use hardware signatures that are manipulatable in software. 39
3.2 Employing SR1 and SR2 in uses whose lifetime (length of the segment) is unpredictable statically. 43
3.3 Status Vector associated with a signature. 48
3.4 SoftSig architecture. 50
3.5 The signature register file. 52
3.6 The ICD prevents missing a remote conflict. 54
3.7 Applying the MemoiSE algorithm to function `foo`: function code layout (a), lookup table (b), Prologue (c), Setup (d), Epilogue (e), and stack layout (f). 57
3.8 Dynamic instruction count relative to *Baseline*. 64
3.9 Execution time relative to *Baseline*. 65
3.10 Mean number of SRF accesses per cycle of execution. 66

4.1 Using atomic region support for Loop Invariant Code Motion (a) without loop blocking and (b) with loop blocking. 79
4.2 Loop (a) before and (b) after performing LICM using SAA. 80
4.3 Cost-benefit model for an atomic region(AR). 81
4.4 Example loop in function mult_su3_na(...) of 433.milc (a) before and (b) after applying *SpecAA*. 86
4.5 Example loop in subroutine EXTRAPI() of 437.leslie3d (a) before and (b) after applying *SpecAA*. 87

4.6 Fraction of alias queries performed by LICM that returned a *may alias*, *no alias*, or *must alias* response for (a) SPEC INT2006 and (b) SPEC FP2006. B, D, and S stand for *BaselineAA*, *DSAA*, and *SpecAA*. 91
4.7 Speedups normalized to baseline for (a) SPEC INT2006 and (b) SPEC FP2006. 92
4.8 Dynamic instructions normalized to baseline broken down by category for (a) SPEC INT2006 and (b) SPEC FP2006. B, D, and S stand for *BaselineAA*, *DSAA*, and *SpecAA*. 94
4.9 Speedups when varying the speculation threshold for (a) SPEC INT2006 and (b) SPEC FP2006. 95

Chapter 1

Introduction

Research on transactional memory began as a tool to improve the experience of programmers working on parallel code. Just as transactions in databases, it was the job of the runtime to detect any conflicts between parallel transactions and rollback the ones that needed to be re-executed, leaving the programmers blissfully unaware of the communication and synchronization that needs to happen. The programmer only needed to cover the sections of code that might generate conflicts in transactions, or atomic regions.

More recently, new uses for transactional execution were proposed where, not only were user specified sections of code executed transactionally but the entire program was executed using transactions [9, 15, 27, 35, 88, 92, 59, 54]. In this environment, the hardware is in charge of generating the transactions, also called chunks, unbeknownst to the user. Executing instructions in chunks helps the parallel programmer further in unexpected ways such as providing a sequentially consistent(SC) memory model with low overhead [9, 15, 35, 88], aiding atomicity violation detection [54], enabling deterministic replay [59], and even deterministic execution [27]. All of this could be achieved without the help of chunked execution, but only through constricting or monitoring the interleaving of memory accesses between parallel threads, which severly slows down execution. With some hardware support for transactional execution, chunked execution can provide all these benefits with negligible overheads by using speculation. Furthermore, these benefits are applicable to both transaction-based or lock-based code since the chunking is done invisible to the user.

Advances in transactional memory research have recently begun to have a direct impact on industry as more and more silicon companies are announcing extensions to their instruction sets

to support transactional execution. The list now encompasses: Intel's Transactional Synchronization Extensions [1], AMD's Advanced Synchronization Facility [18], Azul's Vega [19], IBM's Bluegene/Q, and Sun's ROCK [17], and the Transmeta processors [26].

However, despite the many potential benefits to programmability, the implications of this chunking hardware on the compiler layer have not yet been explored. Since the compiler sits in between the chunking hardware and the programmer, for the benefits in programmability to percolate upwards, additional support in the software layer is necessary. Specifically, to expose an SC memory model to the programmer, the compiler also needs to guarantee SC execution while doing optimizations. Thankfully, the speculation hardware used for chunked execution can help the compiler achieve this without constricting optimization and, surprisingly, even outperform current compilers. Moreover, it is shown that exposing chunking hardware to the compiler can unlock further novel optimizations that was impossible in traditional systems. These optimizations make use of both atomic chunks and hardware signatures already present in chunking hardware. Hardware signatures, or hardware Bloom filters [8], are key components in a chunked architecture as shown in Section 1.1 and later in Appendix **??**.

First, the Thesis shows that the same idea of using speculation to relax memory ordering restrictions in hardware can be applied to compilers, to enable more optimizations than was traditionally possible under the language memory model. Specifically, it is shown that a compiler that adheres to the strict SC memory model can even outperform a compiler that adheres to the much more relaxed Java Memory Model, leveraging chunked execution. This is done by having the compiler demarcate atomic regions in the code, which are guaranteed to executed within a single chunk, thereby guaranteeing atomicity and isolation. Thus, the compiler is free to do any optimization and memory reordering it wishes within an atomic region without caring about how other threads will interleave with it. Using these atomic regions the compiler can even optimize across synchronization points, which was impossible even under the relaxed Java Memory Model, giving it an average speedup of 37% over the commercial grade Hotspot Java server compiler for selected Java benchmarks.

Second, the Thesis goes on to propose a sophisticated ISA for the compiler, or any other software tool, to control and manipulate hardware signatures for advanced code analysis and optimization. For this purpose, the hardware exposes a signature register file to software, each signature consisting of a Bloom filter [8] that encodes memory addresses into a register. The software has control over what set of addresses are collected in these signatures and can perform operations among them to check certain properties about the sets of addresses. One novel optimization named MemoiSE is studied on this sytem, which involves performing memoization on functions with complex side-effects by summarizing the accessed memory locations in signatures. It is shown that MemoiSE reduces dynamic instruction count by 9.3% thereby improving performance by 9%.

Third, the Thesis shows how hardware support for transactional execution can be exposed to the compiler to aid in alias analysis, which is crucial to many traditional compiler optimizations that require code movement. This is done by demarcating the program code with atomic regions as before, after which the compiler can speculate on alias relationships within an atomic region. Runtime checks are inserted to guarantee that the speculation succeeded after which the atomic region is committed. Otherwise the atomic region is rolled back and a version of code without speculation is executed. It is shown that the same hardware support to enable data conflict detection between transactions can be leveraged to enable these runtime checks with only slight modifications. This novel type of alias analysis is implemented on LLVM, a commercial grade compiler, and Loop Invariant Code Motion (LICM) is run as a client pass to evaluate improvements to code quality. It is shown that the new analysis when used with LICM reduces dynamic instruction count by 14% for SPEC FP2006 programs and 2% for SPEC INT2006 programs on average.

The remainder of this chapter presents background on a hardware implementation of chunked execution named BulkSC, which uses hardware signatures to perform disambiguation and conflict detection among the chunks. This implementation will serve as a basis for all the improvements and novel optimizations presented in this Proposal. Finally, this section concludes with an

outline for the rest of the Thesis Proposal.

1.1 Background on BulkSC

1.1.1 Overview

The BulkSC multiprocessor [15], executes by committing *Chunks* of instructions at a time. A chunk is a group of contiguous instructions *dynamically* formed by hardware. Typically, the hardware forms a chunk every 2,000 instructions.

Each chunk executes on the processor *atomically* and *in isolation*. Atomic execution means that none of the actions of the chunk are made visible to the rest of the system (processors or main memory) until when the chunk commits. Execution in isolation means that the chunk appears to have executed at the point of commit instantaneously without any interference from the rest of the system. In reality, BulkSC speculates that there will be no interference and executes the chunk in parallel with the rest of the system and if there is an interference, rolls back and re-executes the chunk.

In addition, BulkSC guarantees that chunks commit in program order in each processor, and there is an arbiter that globally orders their commits. Coupled with the atomicity and isolation properties of chunks, it is trivial to prove that BulkSC supports SC at the chunk level — and, as a consequence, SC at the instruction level. The instructions just have to obey program order inside the chunks without any memory consistency model considerations.

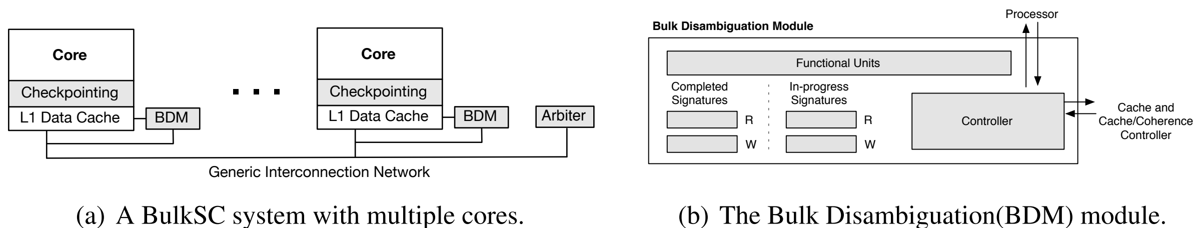
This last property is key in enabling a high-performance SC implementation. In BulkSC, the hardware can reorder and overlap all memory accesses within a chunk — except, of course, those that participate in single-thread dependences. In particular, synchronization instructions induce no reordering constraint. Indeed, *fences* inside a chunk are *transformed into no-ops*. Their functionality — to delay execution until certain references are performed — is useless since, by construction, no other processor will observe the actual order of instruction execution within a

chunk. Moreover, a processor can also overlap the execution of consecutive chunks [15].

1.1.2 Implementation

Figure 1.1(a) shows an implementation of the BulkSC architecture. The additions to a conventional system to enable BulkSC are shaded in gray. Most of the modifications to the processor, besides the register checkpointing support to enable rollback, resides below the cache hierarchy and is implemented in a module named the Bulk Disambiguation Module(BDM) shown in Figure 1.1(b). The BDM includes *R* and *W* signatures which are essentially Bloom filters [8] that automatically encode the addresses read and written by a chunk as it executes. There are two sets of signatures, *completed* and *in-progress*, to support the overlapped execution of consecutive chunks.

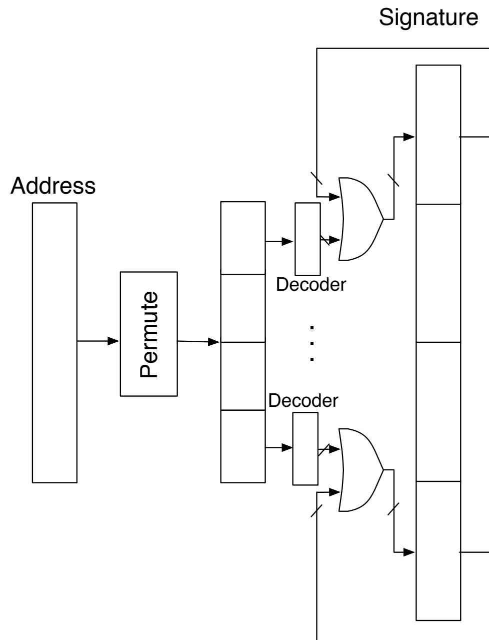
Figure 1.2(a) describe how addresses are encoded into signatures. The address bits are initially permuted. Then, in the resulting address, we select a few bit-fields. Each of these bit-fields is then decoded and bit-wise OR'ed to the current value of the corresponding bit-field in the signature. This operation is done in hardware. Some basic primitive operations that can be performed on signatures are shown in Figure 1.2(b). Signature intersection and union are bit-wise AND and OR operations, respectively, on two signatures. Membership testing can be implemented by inserting the member to a signature first and then performing an intersection with that signature. Being bit-wise operations, these operations on sets of addresses can be performed very quickly and efficiently, which is key to software optimizations that are described in this Thesis.



(a) A BulkSC system with multiple cores.

(b) The Bulk Disambiguation(BDM) module.

Figure 1.1: The BulkSC architecture.



(a) Address insertion into a signature.

Op.	Description
$\cap$	Signature intersection
$\cup$	Signature union
$= \emptyset$	Is signature empty?
$\in$	Membership of an address in a signature

(b) Some primitive operations on signatures.

Figure 1.2: Bloom filter signatures.

Atomic chunk execution is supported by buffering the state generated by the chunk in the L1 cache, which is controlled by the BDM. As the chunk executes, the signatures encodes the memory addresses accessed by the chunk. After the chunk completes, the hardware sends *W* to an arbiter, which forwards it to other processors. In the other processors, *W* is intersected with the local *R* and *W* signatures in the BDM. A non-null result indicates an overlap of addresses, which causes the local chunk to get squashed and restarted. In addition, all local cache lines whose addresses are in *W* are invalidated to avoid stale data. The signatures are key in enabling atomicity and isolation in chunk execution, as well as maintaining cache coherence. The chunked execution is invisible to the software layers; all that is visible is a high performance SC machine.

1.2 Summary

Chunk-based execution is a powerful architectural technique. Chapter 2 proposes the use of atomic regions to perform memory ordering speculation in the compiler by leveraging the chunk-based hardware. Chapter 3 proposes the use of hardware signatures to enable various optimizations and analyses. Chapter 4 proposes using both atomic regions and hardware signatures to perform alias analysis, and its succeeding chapter concludes.

Chapter 2

High-Performance Sequential Consistency through Cooperative Compiler and Hardware Support

2.1 Introduction

The arrival of multicore chips as the commodity architecture for many platforms has highlighted the need to make parallel programming easier. While this endeavor necessitates advances in all layers of the computing stack, at the hardware architecture layer it requires that multicores be designed to support programmer-friendly models of concurrency and memory consistency efficiently.

The memory consistency model specifies what values a load can return in a shared-memory multithreaded program [2]. One such model is Sequential Consistency (SC). SC mandates that the result of any execution of the program be the same as if the memory operations of all the processors were executed in some total sequential order, and those of each individual processor appear in this sequence in the order specified by its thread [44]. There is consensus that software writers prefer that the platform support SC because it offers the same simple memory interface as a multitasking uniprocessor.

For software that is well synchronized (i.e., one that does not contain data races), most systems used today support SC with high performance. This is because synchronization operations totally order those accesses from different threads that, if overlapped, could result in non-intuitive return values for loads. Unfortunately, much current software, ranging from user applications to libraries, virtual machine monitors, and OS, has data races — either by accident or by design. For these codes, SC is not provided. Moreover, as more beginner programmers attempt parallel programming on multicores in the next few years, the number of codes with data races may well

increase.

2.1.1 Benefits of Supporting SC

Devising a platform that supports SC with high performance for *all* codes — including those with data races — would have four key benefits. The first one is that debugging concurrent programs would be easier. This is because the possible outcomes of the memory accesses involved in the bug would be easier to reason about, and the debugger could in fact *reproduce* the buggy interleaving.

A second benefit stems from the fact that existing software correctness tools almost always assume SC — for example, Microsoft's CHESS [61]. Verifying software correctness under SC is already hard, and the state space balloons if non-SC interleavings need to be inspected as well. In the next few years, software correctness verification tools are expected to play a larger role. Using them in combination with an SC machine would make them most effective.

A third benefit of SC is that it would make the memory model of safe languages such as Java easier to understand and verify. The need to provide safety guarantees and enable performance at the same time has resulted in an increasingly complex and unintuitive memory model over the years. A high-performance SC memory model would trivially ensure Java's safety properties related to memory ordering, and improve its security and usability.

Finally, some programmers want to program with data races to obtain high performance. This includes, for instance, writers of OS and virtual machine monitors. If the machine provided SC, the risk of introducing bugs would be reduced and the code portability enhanced.

2.1.2 Goal and Contributions

From this discussion, we argue that supporting SC is a worthy goal. Recently, there have been several proposals for hardware architectures that support high-performance SC [9, 12, 15, 34, 35, 88, 92]. Some of these architectures support SC all the time by repeatedly committing groups of

instructions atomically — called chunks in BulkSC [15], transactions in TCC [35], or implicit transactions in checkpointed multiprocessors [88]. Each instruction group executes atomically and in isolation, generating a total commit order of chunks and, therefore, instructions, in the machine. Such properties guarantee SC. Moreover, thanks to operating in large instruction groups, the overheads of supporting SC are small. Conceivably, a similar environment can be attained with a primitive for atomic region execution such as that of Sun's Rock [17], if it is invoked continuously.

Unfortunately, for a platform to support SC, it is *not enough* that the hardware support SC; the software — in particular, the compiler for programs written in high-level languages — *has to support SC as well*. For this reason, there have been several research efforts on compilation for SC [41, 85, 90]. Such efforts have sought to transform the code to satisfy SC on conventional multiprocessor hardware. The results have been slowdowns — often significant — relative to the relaxed memory models of current machines.

Remarkably, with the group-commit architectures, we have an opportunity to develop a high-performance SC compiler layer. Since the hardware already supports high-performance SC, all we need is for the compiler to drive the group-formation operation, and adapt code transformations to it. With the combination of hardware and compiler, the result is a *whole-system high-performance SC platform*. Furthermore, since the hardware guarantees atomic group execution, the compiler can attempt more aggressive optimizations than in conventional, relaxed-consistent platforms. The result is even *higher performance* than current aggressive platforms.

This work presents the hardware-compiler interface and the main ideas for a compiler layer that works in the BulkSC architecture (as a representative of the group-commit architectures) to provide whole-system high-performance SC. We call our compiler algorithm *BulkCompiler*. Our specific contributions include: (i) ISA primitives for BulkCompiler to interface to the chunking hardware, (ii) compiler algorithms to drive chunking and code transformations to exploit chunks, and (iii) initial results of our algorithms with Java programs on a simulated BulkSC architecture.

Our results use Java applications modified with our compiler algorithms and compiled with

Sun's Hotspot server compiler [66]. A whole-system SC environment with BulkCompiler and simulated BulkSC architecture outperforms a simulated conventional hardware platform that uses the more relaxed Java Memory Model by an average of 37%. The speedups come from code optimization inside software-assembled instruction chunks.

This chapter is organized as follows: Section 2.2 gives a background; Sections 2.3 and 2.4 describe BulkCompiler and how it manages the chunks; Sections 2.5 and 2.6 evaluate the system; Section 2.7 assesses the results, and Section 2.8 discusses related work.

2.2 Background

We briefly describe the algorithm for generating chunks in the BulkSC architecture and the current approaches for compiler-driven enforcement of SC.

2.2.1 Algorithm for Generating Chunks

In BulkSC, the hardware finishes the current chunk and starts a new one when the number of dynamic instructions executed exceeds a certain threshold that we call *maxChunkSize* (e.g., 2,000 instructions). There are, however, some events that affect the regular generation of chunks. Table 2.1 lists these events and, under *Actions in BulkSC*, the actions taken [15]. For example, when the write set of the chunk is about to overflow the cache, the hardware commits the current chunk at this point and starts a new chunk. The last column of the table will be discussed later.

2.2.2 Compiler-Driven Enforcement of SC

A compiler can take programs with potential data races and transform them to enforce SC even on a machine that implements a relaxed memory consistency model [41, 85, 90]. The general idea is to identify the minimal set of ordered pairs of memory accesses that should not be reordered, and then (1) insert a fence along every path between the first and second access in each

Event	Actions in BulkSC	Actions with BulkCompiler Inside Atomic Region
maxChunkSize instructions executed	The hardware commits the current chunk and starts a new chunk	No action
Cache overflow	The hardware commits the current chunk at this point, and starts a new chunk	The hardware squashes the current chunk and restarts it at the Safe Version point
Data collision with remote chunk	The hardware squashes the chunk and re-executes it. If the chunk is squashed *M* times, then the chunk also reduces its size to minimize collisions	Same as in under BulkSC. However, if the chunk size has to be reduced, restart the chunk at the Safe Version point
Exceptions (including system calls)	When the code wants to perform an uncacheable access, the hardware commits the current chunk at this point, performs the uncached operation, and starts a new chunk	When the code wants to perform an uncacheable access, squash the chunk and restart it at the Safe Version point. Do not set up an atomic region to include uncacheable accesses
Interrupts	The hardware completes the current chunk and then processes the interrupt in a new chunk(s)	The hardware squashes the current chunk, processes the interrupt, and then restarts the initial chunk under an atomic region again

Table 2.1: Events that affect chunk generation.

pair, and (2) prohibit the compiler from performing any transformation that reorders any such pair.

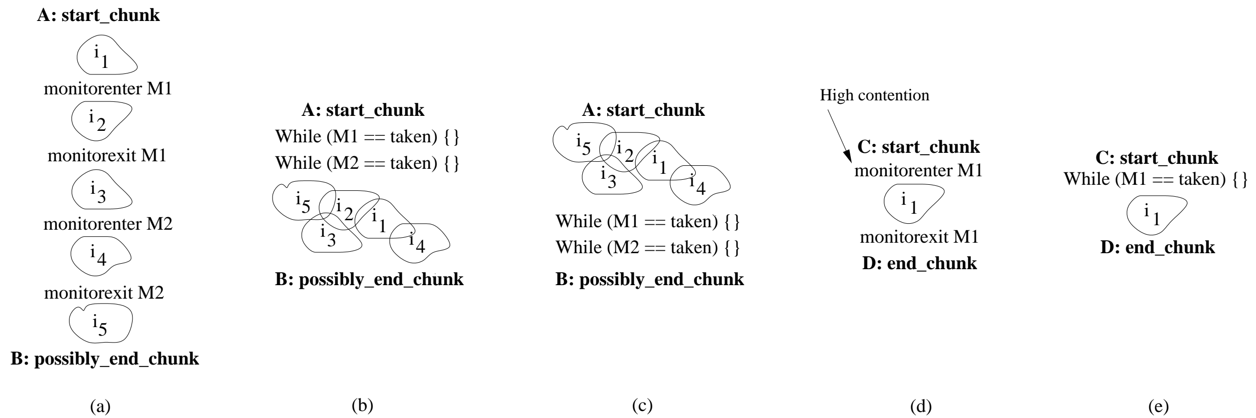


Figure 2.1: Compiler-driven chunking for high performance SC. In the figure, each i_j represents a set of instructions.

The compiler analysis needed involves first performing Escape analysis [85], which determines which loads and stores may refer to memory locations accessed by multiple threads. Then, May-happen-parallel (or Thread-structure) analysis [62, 85] determines which memory accesses can happen in parallel. Based on these, Delay Set analysis [78] determines which of the shared accesses should not be reordered within a thread.

Unfortunately, the compiler analysis required is very costly both in runtime and in implemen-

tation effort — in part because every step needs interprocedural analysis. Moreover, all three existing implementations [41, 85, 90] report noticeable slowdowns relative to execution of the application under the relaxed model — in some cases, applications become several times slower. The goal of this work is to deliver SC with even higher performance than current relaxed models.

2.3 Compiler for SC

We submit that an architecture with continuous group commit such as BulkSC [15], TCC [35], or Checkpointed Multiprocessors [88] can potentially deliver whole-system (hardware plus software) SC at a higher performance than conventional machines deliver a relaxed memory consistency model. This is because, if the compiler drives chunk formation appropriately, the atomicity guarantee of chunks can enable many compiler optimizations inside the chunk.

In particular, we focus on multiprocessor-related issues. We observe that synchronization and fences can substantially hurt the performance of conventional relaxed-consistency machines. At the same time, synchronization-aware chunk formation can eliminate some of these problems, and further enable conventional compiler optimizations that improve performance.

In this section, we discuss the main ideas, the new instructions added, and the basics of the algorithms in *BulkCompiler* — our compilation layer for group-commit architectures. In a later section (Section 2.7), we briefly discuss how we can also improve the performance of relaxed memory consistency models in these architectures and enable new compiler optimizations.

2.3.1 Main Ideas

A compiler for a group-commit architecture should select the chunk boundaries so that they (1) maximize the potential for compiler optimization and (2) minimize the chance of chunk squash. Since the design space is large, this work focuses on the multiprocessor related issues of synchronization and fences. In this area, BulkCompiler relies on one idea to maximize compiler optimization and one to minimize squashes.

Instruction	Functionality
beginAtomic PC	Finishes the current chunk, triggers a register checkpoint in hardware, and starts a new chunk. It takes as argument the program counter (PC) of the entry point to the *Safe Version* of the code, which will be executed if the chunk needs to be chopped into smaller chunks.
endAtomic&Cut	Finishes the current chunk and changes the mode of chunking from software-driven to hardware-driven. The hardware will start a new chunk next.
endAtomic	Changes the mode of chunking from software-driven to hardware-driven, enabling the hardware to finish the current chunk when it wants to (e.g., when the chunk size reaches *maxChunkSize*).
squashChunk	Squashes the current chunk and restarts it at the Safe Version. It involves clearing the BulkSC signatures, invalidating the cache lines written by the chunk, and restoring the checkpointed register file.
cutChunk	Finishes the current hardware-driven chunk, inducing the hardware to start a new one. It has no effect if found inside a *beginAtomic* to *endAtomic&Cut* (or *endAtomic*) region.

Table 2.2: Instructions added so that the compiler manages the chunking.

Maximizing Compiler Optimization

To maximize compiler optimization, BulkCompiler identifies *low contention* critical sections (which are mostly in the form of synchronized blocks in Java). Then, it includes one or several of them and their surrounding code in the same chunk (Figure 2.1(a)). After this, each acquire operation (*monitorenter* instruction in Java bytecode) is replaced with a spinning loop, which checks if the synchronization variable is taken using *plain loads*. Moreover, all the release operations (*monitorexit* in Java bytecode) are removed. Next, we move the spinning on the locks with plain loads to the top of the chunk — subject to data and control dependences — to prepare the code for compiler optimization better. Finally, with the synchronizations removed, we let the compiler aggressively reorder and optimize the code inside the chunk. The resulting code is shown in Figure 2.1(b), where the overlapping sets of instruction denote the effect of compiler optimization. Note that checking all the locks at the beginning of the chunk may slightly reduce concurrency. However, since we apply this transformation to low-contention critical sections, such effect is insignificant.

Since the chunk will be executed atomically, there is no need to acquire and release a lock. However, the chunk still needs to read the locks with plain loads, to check if any lock is taken. A lock can be taken if another thread, after failed attempt(s) to execute its own chunk atomically, reverted to a (non-speculative) *Safe Version* of the code, where it grabbed the lock. We will see

in Section 2.3.4 that every atomic region has a corresponding Safe Version, where any locks are acquired and released explicitly. This is the same approach followed by the Speculative Lock Elision (SLE) algorithm [70] and its implementation in the Sun Rock [28].

If any of the locks is taken, the code spins. When the owner of the lock commits the lock release, the spinning chunk will observe a data collision on the spinning variable. At that point, it will be squashed and re-started.

By eliminating the synchronization operations, this transformation improves performance in two ways. First, the processor avoids performing the costly synchronization operations, replacing acquires with the much cheaper loads. More importantly, however, is that this transformation eliminates the constraints on instruction reordering imposed by synchronization instructions. Indeed, even under current relaxed memory models, compilers neither move instructions across synchronization operations nor allocate shared data in registers across them. This disables many instances of conventional optimizations such as register allocation, common subexpression elimination, loop invariant code motion, or redundant code motion, to name a few. After we remove the synchronization operations, a conventional compiler can reorder instructions and perform all of these optimizations.

We can place the spinning on the locks with plain loads at the end of the chunk, after all the work is done (Figure 2.1(c)). This approach makes a difference when one or more locks are taken by other processors and, therefore, the chunk will eventually be squashed. In this case, having the spinning at the end of the chunk can enable prefetching of read-only data for the chunk re-execution. However, it may also cause exceptions resulting from accessing data of a critical section while another processor is also accessing it. Overall, since we apply this transformation to low-contention critical sections, these effects are not very significant.

Finally, this transformation is especially attractive in Java programs, which is the environment examined in this work. This is because Java programs have many low-contention critical sections in the form of synchronized methods — often in thread-safe Java libraries. The synchronized blocks in these methods are compiled into Java bytecode using the *monitorenter* and *monitorexit*

bytecode instructions surrounding the code in the block.

Minimizing Squashes

The second idea in BulkCompiler is to minimize squashes by identifying *high-contention* critical sections and tight-fitting a chunk around it (Figure 2.1(d)). As in the previous transformation, *monitorenter* is replaced with a loop that checks if the lock is taken using plain loads. *Monitorexit* is removed (Figure 2.1(e)). Tight-fitting the chunk reduces the chances that different processors collide on this critical section, and also reduces the number of wasted instructions per squash. It also enables processors to hand over access to popular critical sections to other processors sooner, since chunks commit sooner.

Even after all these transformations, chunks created by the compiler can collide at runtime — either on the synchronization variable or on another variable. In this case, they retry as per the default BulkSC execution. However, there are events that require reducing the size of the chunk, such as a cache overflow or performing an uncached memory access. Reducing the chunk size could lead to non-SC executions if the broken chunk exposes reordered references to shared data. To prevent this, BulkCompiler also creates the Safe Version of the code mentioned before. The Safe Version does not reorder references to shared variables and includes the *monitorenter* and *monitorexit* instructions.

Overall, with these changes on top of high-performance SC hardware, we target a performance higher than that attained with the relaxed Java Memory Model on conventional hardware, while providing whole-system SC.

2.3.2 New Instructions Added

Table 2.2 shows the instructions added to enable the compiler to manage the chunking. The principal ones are *beginAtomic*, which marks the beginning of an atomic region, and *endAtomic&Cut* or *endAtomic*, which mark the end. *BeginAtomic* causes the BulkSC hardware to finish the current chunk and start a new one. It also creates a register checkpoint to revert to if the chunk is

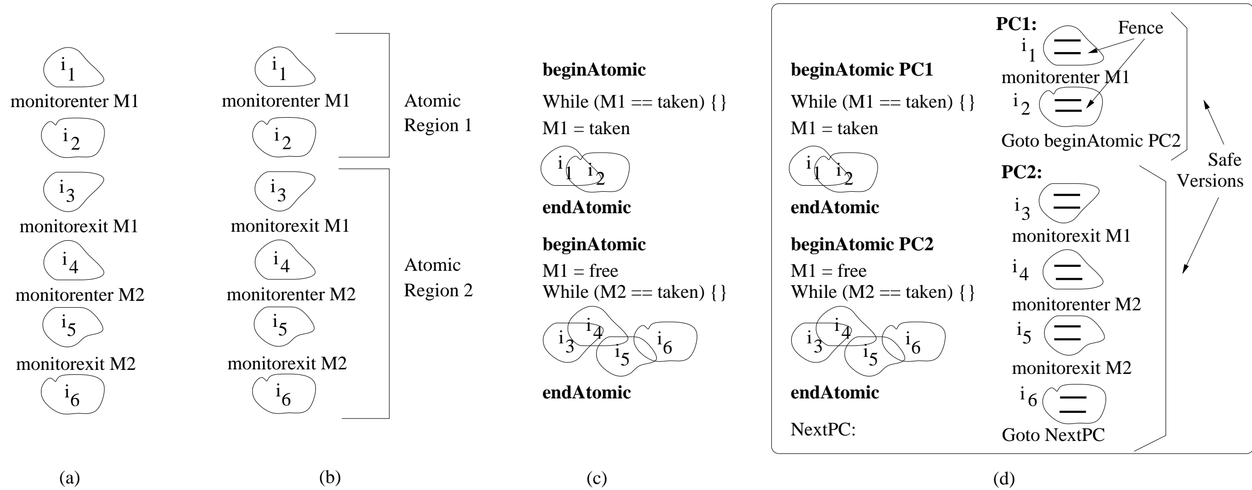


Figure 2.2: Transforming a large code section. In the figure, each i_j represents a set of instructions.

squashed. The instruction takes the program counter (PC) of the entry point to the Safe Version of the code for the chunk. When the atomic region is squashed, depending on the reason for the squash, the hardware returns execution to either the *beginAtomic* instruction or the entry point to the Safe Version.

EndAtomic&Cut terminates the current chunk and then lets the BulkSC hardware take over the chunking — the hardware will start a new chunk next. *EndAtomic* simply lets the BulkSC hardware take over the chunking. This means that the current chunk may continue executing until a total of *maxChunkSize* instructions since *beginAtomic* have been executed. When a chunk is executing within the *beginAtomic* to *endAtomic&Cut* (or *endAtomic*) instruction pairs, reaching the *maxChunkSize* instruction count does not cause chunk termination. Overall, with these primitives, we surround the groups of low-contention synchronized blocks as in Figure 2.1(b) with *beginAtomic* at point *A* and *endAtomic* at point *B*; we surround the high-contention synchronized blocks as in Figure 2.1(e) with *beginAtomic* at point *C* and *endAtomic&Cut* at point *D* in the figure.

To the compiler, *beginAtomic* has acquire semantics, which means that it cannot move any escaping reference (i.e., reference to shared data) that follows *beginAtomic* to before it. *EndAtomic&Cut* and *endAtomic* have release semantics, and the compiler cannot move any escaping reference that

precedes them to after them.

The table shows two more instructions, called *squashChunk* and *cutChunk*. The former squashes the current chunk and restarts at the Safe Version. It can be used for speculative compiler optimizations, which sometimes require a rollback after discovering that they have performed an illegal transformation (e.g., [64]). The *cutChunk* instruction simply finishes the current hardware-driven chunk, inducing the hardware to start a new chunk. It has no effect if found inside a *beginAtomic* to *endAtomic&Cut* (or *endAtomic*) region. Note that if, dynamically, *endAtomic&Cut*, *cutChunk*, or potentially *endAtomic* are immediately followed by *beginAtomic*, the latter does not start a second chunk beyond the one that the hardware is starting.

2.3.3 Difference to Transactional Memory

To understand our transformations, it is useful to compare them to Transactional Memory (TM). The main goal of TM is enhancing concurrency; the main goal of our transformations is enhancing the performance of each thread through compiler optimization while preserving SC. However, since we focus on optimization opportunities afforded by synchronizations, our use of an SLE-like algorithm also enhances concurrency, especially in high-contention critical sections.

To see the difference between the two goals, consider a synchronized block that is too large for the hardware to provide atomicity. Unlike TM, BulkCompiler still benefits from splitting the code into two atomic regions. This is seen in Figure 2.2(a), which shows code with two synchronized blocks protected by locks M1 and M2. Assume that BulkCompiler estimates that the code in the M1 block has a footprint that amply overflows the cache. Further, assume that it estimates that the code before the M1 block (i_1) could be optimized together with the code inside the block. In this case, it partitions the code into two atomic regions that it estimates fit in the cache (Figure 2.2(b)): one that executes i_1 and the beginning of the first block, and another that executes the rest of the code.

BulkCompiler relies on the hardware guarantee that each region executes atomically. It transforms the code as shown in Figure 2.2(c): synchronization operations become plain accesses and

the code is aggressively reordered and optimized. In particular, in the first atomic region, *monitorenter* is replaced with a spinning loop, which checks if the lock is taken using plain loads. If the lock is free, the code sets it to taken. If the chunk eventually finishes and commits, this lock update will be made visible; however, the chunk may be squashed before committing by the commit of another chunk that also set the lock. On the other hand, if the lock was not free, the code spins and will not commit. The chunk will eventually get squashed, either when the thread is preempted from the processor or when the chunk that releases the lock commits.

In the second atomic region, *monitorexit M1* simply becomes a plain write to the lock variable to release it. If the chunk commits, the write will be visible to the rest of the processors. Note that lock variable M1 has to be explicitly written as taken or freed, although the writes can be plain stores. This is because, since the synchronized block is now split into two regions, atomicity is no longer guaranteed and we have to rely on the value of the variable to prevent illegal interleavings. Finally, the accesses to *M2* are replaced with a spinning loop on *M2* with plain loads as described before. Overall, in all cases, the rest of the code is heavily optimized and the system satisfies SC.

2.3.4 Safe Version of the Atomic Region Code

It is possible that an atomic region gets squashed. Recall that Column 2 of Table 2.1 showed the events that affect chunks in the original BulkSC architecture. The last column of the table shows how we slightly change the BulkSC hardware so that it guarantees the atomicity of atomic regions.

First, inside an atomic region, the chunk is prevented from finishing when the number of instructions reaches past *maxChunkSize*, to guarantee that the entire atomic region does in fact commit atomically. Second, since this requirement can result in long atomic regions, we want to process interrupts as soon as they are received — rather than waiting until the current chunk completes. Consequently, on reception of an interrupt, the current chunk is squashed, the interrupt is processed, and then the initial chunk is restarted from the beginning — using the check-

point from *beginAtomic*.

Finally, to guarantee the atomicity of atomic regions, events that previously triggered a chunk squash may need to be handled differently. These events include (i) cache overflows, (ii) uncacheable accesses in exceptions (which include system calls), and (iii) data collisions with a remote chunk. How we handle these events largely depends on whether the event will (likely) repeat after the chunk is squashed and restarted.

The events that are unlikely to repeat are most data collisions. In this case, the atomic region is squashed and then re-executed from the beginning. The events that repeat are cache overflow, uncacheable accesses in exceptions, and repeated data collisions on the same chunk in pathological cases. Some cases of uncacheable accesses can be avoided by not including problematic system calls inside atomic regions. However, the rest of the events are largely unpredictable and hard to avoid. The atomic region cannot be simply squashed and re-executed since it will be squashed again.

To make progress in these cases, we would have to commit a downsized chunk —- i.e., the code up until we cause the cache overflow, or reach the uncacheable access or the access that causes the collision. However, this would break the atomicity of the chunk and, potentially, expose inconsistent or non-SC state. Consequently, to address these cases, a Safe Version of the code is generated for each atomic region. This safe code does not rely on atomic execution to preserve SC. If the atomic region needs to be truncated for any of the "repeatable" reasons, the chunk is squashed and execution is transferred to the PC of the Safe Version entry point — as given in the *beginAtomic* instruction.

The Safe Version of the code acquires and releases locks explicitly. Moreover, it also has to satisfy SC. Therefore, BulkCompiler conservatively identifies all the escaping references in the code using the algorithm in [48]. Then, it adds a fence at the beginning of the Safe Version code, and after every escaping reference. The fences prevent the compiler from reordering the escaping accesses — and hence ensure SC at a performance cost. The analysis of Section 2.2.2 could keep the overheads to a minimum. Figure 2.2(d) shows the final code for the example.

Fortunately, part of this performance loss is transparently recovered by the chunking hardware. Specifically, as the BulkSC hardware executes the Safe Version code with hardware-driven chunks, fences are *no-ops* (Section 1.1). The accesses that fall in the same chunk will be overlapped and reordered by the hardware, irrespective of the presence of the fences. Note also that, since Safe Versions are rarely executed, they will not hurt the instruction cache through code bloat noticeably.

2.4 Algorithm Design

In this section, we describe the algorithms that we use and some of the corner cases encountered.

2.4.1 Inserting Atomic Regions

At the highest level, our algorithm desires to have all escaping references contained in atomic regions, and for each region to be as large as possible to expose the maximum number of optimization opportunities. Doing this naively, however, will lead to excessive squashing of atomic regions due to conflicts or cache overflow, and difficulty in generating code for the Safe Versions of the regions.

The algorithm that we use is shown in Figure 2.3. This algorithm is applied to each method in turn. Prior to actually selecting atomic regions, the algorithm performs aggressive inlining, escape analysis [48], and loop blocking. Inlining reduces the impact of using an intraprocedural algorithm for selecting atomic regions. Escape analysis identifies the escaping references in the method, namely the references to objects that may be accessed by two or more threads. These references should be enclosed in atomic regions. Finally, loop blocking transforms inner-most loops into a loop nest, with a constant bound on the iteration count of the innermost loop. This allows the innermost loop to be enclosed in an atomic region that fits in the cache. Loops not containing any escaping references need not be blocked.

The algorithm then begins a traversal of the code, and each escaping reference is placed into

1. Perform aggressive inlining.
2. Perform escape analysis and mark escaping references.
3. Block inner-most loops that have escaping references.
4. Traverse code while enclosing each escaping reference in an atomic region.
5. Expand each atomic region r that is immediately control dependent on statement c. We enclose adjacent statements s while all the following hold:
   a. s is control equivalent to r. If s is not control equivalent to r, then:
      i. if s is inside the c control structure, expand r to contain the code from s to P_s (the post-dominator of s). The same applies if P_s is encountered first.
      ii. if $s = c$, first expand r downwards until P_c (the post-dominator of c), and then also add c to r. The same applies if P_c is encountered first.
   b. the estimated footprint of r fits in the cache.
   c. s is not in a highly-contended synchronized block that does not contain r.
6. Generate the Safe Version for all the atomic regions.

Figure 2.3: Algorithm that inserts atomic regions in a method.

an atomic region. After all escaping references are enclosed in atomic regions, a second pass is made to expand atomic regions and merge them where necessary. In the second pass, each atomic region is visited in turn. When an atomic region is visited, it is expanded to enclose code before and after the atomic region, with limits on this expansion as described shortly. If the expansion of an atomic region r_i encounters another atomic region r_j, r_j is merged into r_i, forming a single atomic region.

Three conditions need to hold during this expansion process. The first one is that the atomic region must begin and end at control equivalent points. Let c be the statement on which region r containing escaping reference e is immediately control dependent. This condition is easily satisfied when the statement s encountered while expanding region r is control equivalent to r. However, if it is not control equivalent, care must be taken. Specifically, (1) if s is inside the c control structure and the code from s to P_s (the post-dominator of s) is small enough so that s to P_s fits in the region, then s to P_s is added to r. The same applies if P_s is encountered instead of s. Moreover, (2) if $s = c$, then r is first expanded to cover all statements between c to P_c (the post-dominator of c) such that all statements control-dependent on c are inside r, and then c is also added to r. The same applies if P_c is encountered instead of c.

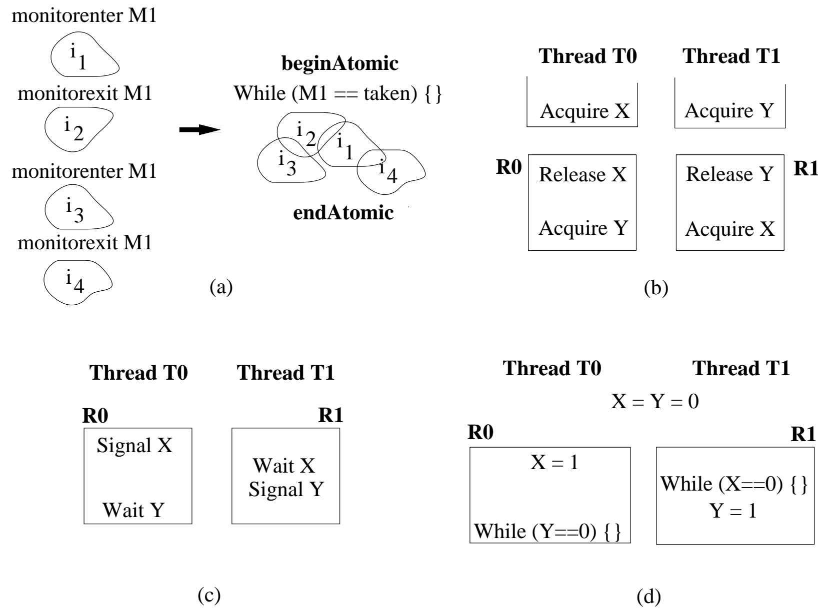


Figure 2.4: Examples of chunks with synchronization operations.

The second condition is that the estimated footprint of the atomic region fits in the cache. A model is used to estimate the contribution of each statement to the footprint. However, the available footprint is assumed exceeded if the algorithm attempts to (1) expand the atomic region into a loop other than the innermost loop around the escaping reference e, or (2) include in the atomic region a non-inlined method call.

The third condition is that atomic regions will not expand to contain statements within a highly-contended synchronized block unless the escaping reference e is in that block. If e is in a highly-contended block, the region will at most be expanded to cover the highly-contended synchronized block.

The resulting atomic regions start and end at control equivalent parts of the program. This ensures that all atomic region starts have a corresponding atomic region end, regardless of the path taken by the program when executing. It also simplifies the generation of the code for Safe Versions.

Finally, Safe Versions of the regions are formed by duplicating the block of code in the atomic regions. A fence is placed at the beginning of the Safe Version code and after every escaping ref-

erence, to ensure that the compiler does not reorder escaping references.

2.4.2 Lock Compression and Region Nesting

When an atomic region contains multiple synchronized blocks protected by the same lock, the algorithm introduces a single check for the lock variable (Figure 2.4(a)). We call this scheme *Lock Compression*.

It is possible that the code contains nested atomic regions. At runtime, our chunking hardware flattens them out, and considers them just one large atomic region. To do this, the hardware keeps a nesting-level counter, and the chunk ends only at the outermost *endAtomic&Cut* or *endAtomic*. Moreover, when a squash is triggered, the outermost atomic region is squashed and, if appropriate, its Safe Version is invoked.

2.4.3 Visibility with Synchronizations

When our algorithm produces an atomic region with accesses to multiple synchronization variables, there may be interactions between threads that cause problems of *Visibility*. As an example, the problem occurs when an atomic region in Thread *T0* releases variable *X* and acquires variable *Y*, while an atomic region in Thread *T1* releases variable *Y* and acquires variable *X*. This is shown for regions *R0* and *R1* in Figure 2.4(b). For simplicity, the figure shows acquire and release operations — in practice, our algorithm will have replaced them with plain memory accesses to the variables. In the figure, Region *R0* cannot complete and make its release of *X* visible to *R1* because it is spinning on *Y*, which *T1* holds. *R1* is in a symmetrical situation. The result is deadlock, as both threads are spinning on the acquires.

The problem is not limited to a pattern where both threads first release a variable and then acquire a second one. It also occurs when there is a *handshake* pattern between two threads. This pattern is shown in Figure 2.4(c), using Signal and Wait synchronization operations. Again, we show these operations for simplicity, although our algorithm uses plain accesses. In the figure,

Region *R0* signals synchronization variable *X* (effectively a release) and then waits on *Y* (effectively an acquire), while *R1* waits on *X* and then signals *Y*. Both threads end up spinning on the waits, unable to complete the regions.

These visibility problems do not occur with the hardware-driven chunks of BulkSC [15]. This is because such chunks complete as soon as *maxChunkSize* instructions are executed, rather than when a certain static instruction is reached. In both examples, the threads would spin in the acquires (or in the waits) until they reach *maxChunkSize* instructions. At that point, they would finish the chunks, making the two releases (in Figure 2.4(b)) or the signal to *X* (in Figure 2.4(c)) visible.

Similar visibility problems have been observed by proposals that integrate locks and transactions [74, 96] and by discussions of transactional memory atomicity semantics [56]. Ziarek *et al* [96] propose to solve the deadlock problem by detecting that two transactions are not completing, squashing them, and executing lock-based versions of the code. The authors state that these cases happen rarely.

BulkCompiler uses a similar approach, which is detailed in Section 2.4.4 and is simpler to implement. However, the problem with "unpaired" synchronization shown in Figure 2.4(b) cannot occur for high-contention critical sections because BulkCompiler tight-fits the atomic region around the section. For low-contention critical sections, an unpaired synchronization may be lumped with other access(es) to synchronization variable(s) within a single atomic region. However, because of the low contention for the synchronization variables, the probability of an interleaving that causes deadlock is very low. An alternative design is to have the compiler disable the creation of such atomic regions.

2.4.4 Visibility with Data Races

If the code is not properly synchronized, data races may produce the deadlock-prone access patterns discussed above. For example, Figure 2.4(d) shows data races that create the handshake pattern. In this case, the compiler may be unable to detect the possibility of deadlock — except,

perhaps, at the cost of expensive and conservative *Must-alias* analysis.

To handle this case and other deadlocks at runtime, BulkCompiler relies on detecting that two chunks are not completing, squashing them, and then triggering the execution of their Safe Versions. Note that, in our environment, detecting that chunks are not completing is easy. Rather than measuring wall-clock time, we count the number of completed instructions — which is needed by the BulkSC hardware anyway. If this number is very high, the processor is likely spinning on a tight loop. At that point, the spinning chunks are squashed and the Safe Versions executed. One option for chunks that suffer frequent timeouts is recompilation.

2.5 Experimental Setup

2.5.1 Compiler and Simulator Infrastructure

Our evaluation infrastructure uses two main components: the Hotspot Java Virtual Machine (JVM) for servers [66] from Sun Microsystems and a Simics-based [89] simulator of the BulkSC architecture [15]. Hotspot is an aggressive commercial-grade compiler with extensive support for just-in-time compilation and adaptive optimization. It is included in OpenJDK7 [84]. We use Hotspot to compile both the unmodified applications for a conventional architecture, and the applications modified with the BulkCompiler algorithms for a BulkSC architecture. We report the difference in performance.

We apply the algorithm described in Section 4.2.2 to Java source code using a profile-driven infrastructure that currently requires substantial hand-holding. We are in the process of automating the infrastructure. Since we are instrumenting at the Java source code level, we cannot directly insert our assembly instructions of Section 2.3.2. Instead, we use the JNI (Java Native Interface) to wrap the instructions in Java methods — at the cost of some overhead.

The resulting modified source is compiled to bytecode, and then run on the Hotspot JVM. The Hotspot JVM executes on top of a full-system execution-driven simulator built using Sim-

ics [89]. The simulator uses the x86 ISA extended with the BulkCompiler instructions. The simulator models a BulkSC multiprocessor [15], including the chunk-based speculative execution, checkpointing, chunk squash and rollback, signature operation, and the extensions needed for BulkCompiler. For comparison, we also model a plain, non-chunk-based multiprocessor.

We model a multicore with 4 single-issue processors running at 4 GHz. Each processor has a 4-way, 64-Kbyte L1 data cache with 64-byte lines. If the cache overflows while executing an atomic region, the chunk gets squashed. Given that the processor model is simple, we report performance in number of cycles taken by the program assuming a constant CPI of 1, irrespective of the instruction type, or whether an access hits or misses in the cache. In some of the experiments, we will assign a fixed cost in cycles to each CAS (Compare-And-Swap) operation. CAS is used to implement synchronization in the Hotspot JVM. In all cases, the results are measured after the application has run a sufficient number of instructions to warm up the code cache.

2.5.2 Experiments and Applications

We start by identifying which synchronization variables in the application have high contention and which have low contention. For this, we use Hotspot, which provides options to profile dynamic locking behavior. It is as simple as running with an additional Hotspot argument. This information enables the targeting of the atomic regions. In addition, our infrastructure uses a simple model of the data footprint of each code section, which is used to decide when the atomic region should terminate, to minimize cache overflow. We often chop loops into multiple blocks of appropriate sizes in order to put each block inside an atomic region.

For the evaluation, we use the SPECJBB2005 and SPECJVM98 benchmark suites. In addition, we also evaluate two additional applications with substantial synchronization, namely *MonteCarlo* from SPECJVM2008 and *JLex* from [6]. Of these applications, SPECJBB2005 and *MonteCarlo* run with 4 threads, and *Mtrt* of SPECJVM98 runs with 2 threads. The rest of SPECJVM98 and *JLex* run with a single thread, although they have many synchronized blocks. These synchronizations are in the Java library code, which includes synchronization because it

has to be thread safe. Each application runs for at least 1B instructions before being measured.

Finally, among the SPECJVM98 applications, we could not evaluate *Javac* or *MpegAudio* because they are commercial applications with no source code, which we need for source level instrumentation. However, we were able to include *Jack* (another SPECJVM98 commercial application) because it has become open source under the name of JavaCC. The JavaCC source distribution includes an input set which is an identical copy of the input set for *Jack* with a few syntactic modifications.

2.6 Evaluation

In this evaluation, we first describe the optimizations that we enable, then present the simulated speedups, and finally characterize the transformations performed.

2.6.1 Understanding the Optimizations Enabled

To understand the way in which BulkCompiler's transformations enable Hotspot to generate faster code, we analyzed the intermediate representation of the code generated by Hotspot with and without the BulkCompilerchanges. We did not add any new compiler optimization to take advantage of chunk-based execution; conventional Hotspot optimizations perform significantly better once Hotspot is given control of the chunks. The following are some common patterns seen:

Loop unswitching. This transformation involves moving a loop-invariant test out of a loop, and then producing two versions of the loop, one in the if-branch of the test, and the other in the else-branch. With the removal of the test, the two loop bodies have a more streamlined control flow and, therefore, the compiler can optimize them, creating better-quality code. The presence of synchronization within the loop had prevented this optimization, since it would have been in violation of the Java Memory Model. However, after BulkCompiler has wrapped the loop inside an atomic region and replaced the synchronizations with plain accesses, Hotspot performs this

optimization automatically. The Java Memory Model will not be violated because the hardware guarantees that there are no intervening conflicting accesses until the atomic region runs to completion.

Null check elimination. In order to satisfy Java safety guarantees, the compiler needs to insert null checks before every object reference — unless it is able to prove that the reference is non-null. If the compiler can prove that two references point to the same object, it can safely remove the checks on the second reference. This situation occurs often inside a loop, where a reference remains invariant through all the iterations. In this case, the compiler peels off the first iteration of the loop, where it inserts all the checks, and removes the checks from the main body of the loop. Hotspot could not do this optimization if there were intervening synchronizations between the references, since it would be illegal. After BulkCompiler's transformations, Hotspot performs this optimization.

Range check elimination. In addition to performing null checks, the compiler is also required to check that an array reference does not exceed the boundaries of the array. Like for null checks, if the compiler is able to prove that an earlier range check subsumes a later range check, the later check can be removed. Once again, however, the presence of intervening synchronizations prevented Hotspot to perform the same loop-peeling optimization in the code described above. With BulkCompiler's transformations, Hotspot performs the optimization.

Loop invariant code motion. Often, the same expression is computed at every iteration of a loop. A common example is when the range of an array which does not change in size needs to be computed repeatedly within a loop. This transformation involves moving the computation outside the loop. If the loop has synchronizations, Hotspot cannot move the computation. With BulkCompiler's transformations, Hotspot can perform the optimization without violating the Java or SC memory models.

Register allocation. Memory locations that were allocated in registers cannot survive synchronization boundaries. The data needs to be stored to memory and loaded back from it, or the Java Memory Model would be violated. BulkCompiler's transformations result in the removal of

many register allocation restrictions, which often result in much more efficient code.

Besides these types of optimizations, the removal of memory fences done by BulkCompiler gives Hotspot much more room for code scheduling. Scheduling is especially important for potentially long delay loads and stores. However, this effect is not evaluated in our results due to the simplistic timing model used in our simulator.

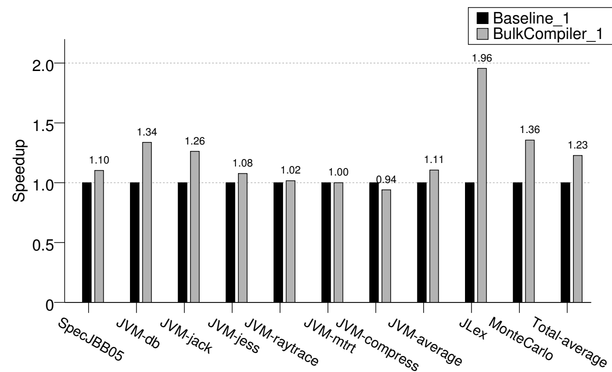
2.6.2 Simulated Speedups

To estimate the performance gains enabled by BulkCompiler, we simulate two environments. The first one (*Baseline*) is unmodified Java running on a conventional (i.e., without chunks) multiprocessor. The second one (*BulkCompiler*) is code transformed by BulkCompiler running on a BulkSC multiprocessor.

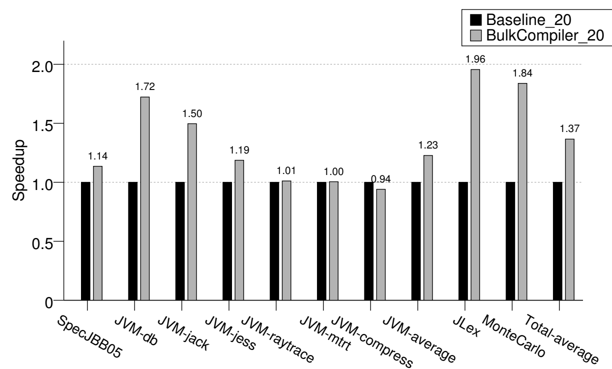
As indicated before, because of the model used in our simulator, we report performance in number of cycles taken by the programs assuming a constant CPI of 1, irrespective of the instruction type. For this reason, we call the two environments above *Baseline_1* and *BulkCompiler_1*. However, it is well known that an important source of overhead in implementations of Java is the actual read-modify-write operations (e.g., CAS) performed in the frequent synchronizations — in the case of Hotspot, potentially two read-modify-write operations for each synchronized block, one at the beginning and one the end. BulkCompiler's transformations replace these operations with plain accesses. Consequently, in our simulations, we also report results for a second scenario, namely one where each instruction takes 1 cycle except for the read-modify-write operations, which take 20 cycles each. The latter is the overhead measured in our workstations for a read-modify-write operation. We call the two environments *Baseline_20* and *BulkCompiler_20* for the two architectures.

Since these environments do not include a high-fidelity architectural model, they do not capture how different memory models use microarchitectures for access overlapping. However, they capture how the compiler can re-order and transform the code under different models, changing the number of instructions executed.

Figure 4.7(a) shows, for each application, the speedup of *BulkCompiler_1* over *Baseline_1*, while Figure 4.7(b) shows the speedup of *BulkCompiler_20* over *Baseline_20*. The bars also include the average for the SPECJVM98 applications, and the average for all the applications. Recall that *BulkCompiler* delivers SC execution, while *Baseline* executes with the relaxed Java Memory Model.



(a)



(b)

Figure 2.5: Speedups of *BulkCompiler_1* over *Baseline_1* (a), and of *BulkCompiler_20* over *Baseline_20* (b).

The figures show that *BulkCompiler* delivers substantial speedups over *Baseline*. In the environment where all the instructions have the same cost, the average speedup of *BulkCompiler_1* across all the applications is 1.23 (or 1.11 if we only consider SPECJVM98). In the environment where the read-modify-write instructions are more costly, which we consider to be more realis-

tic, the speedups are higher. Specifically, the average speedup is 1.37 (or 1.23 if we only consider SPECJVM98). These results show that a *whole-system SC platform*, which guarantees SC at both the compiler and hardware levels, can deliver higher performance than a state-of-the art platform that supports the relaxed Java Memory Model (*Baseline*).

An analysis of the applications shows that most of them get speedups, sometimes quite high. The exceptions are *JVM-raytrace*, *JVM-mtrt*, and *JVM-compress*. We did not get speedups for these applications largely because they do not contain much synchronization in the first place. However, also notice that instrumenting with atomic regions and enforcing SC did not cause them to slow down significantly, either. This is despite the fact that we wrap the *BulkCompiler* assembly instructions in JNI calls (Section 4.3.2), which introduce some overhead. Such overhead would not be present in an implementation that works on the Hotspot intermediate representation. Finally, we note that the speedups of *JLex* are the same for *BulkCompiler_1* and *BulkCompiler_20*. This is because the locks in *JLex* were mostly in the biased [75] state, which does not use any read-modify-write operations.

Application	% of Dyn Instructions in ARs	# of Dynamic ARs	Dyn AR Size	# Sync Blocks per AR	Write Footprint (Lines)	Read Footprint (Lines)	% Instructions in AR Squashed
SPECJBB05	44.5	323086	19117.2	212	489.4	865.6	0.79
JVM-db	75.8	22451	119176.0	2000	84.4	3123.0	0.40
JVM-jack	29.5	2382	30105.2	792	119.7	229.4	1.31
JVM-jess	62.6	33995	43475.6	102	141.1	449.7	0.27
JVM-raytrace	85.8	61419	19771.1	0	51.7	613.9	0.10
JVM-mtrt	77.5	61627	19589.0	0	305.5	1297.0	0.14
JVM-compress	92.7	1632082	5418.6	0	28.1	144.5	0.04
JLex	97.4	45846	131474.0	317	426.9	705.7	0.91
MonteCarlo	99.9	16778	82535.1	2000	11.0	13.0	0.34
Average	74.0	244407	52295.8	602	184.2	826.9	0.48

Table 2.3: Characterizing the dynamic behavior of the code transformed by BulkCompiler. AR stands for Atomic Region.

2.6.3 Characterizing the Transformations

In this section, we characterize the dynamic behavior of the code transformed by BulkCompiler as it runs on the BulkSC architecture. The data is shown in Table 2.3, where AR stands for Atomic Region. In the table, Columns 2–4 show the percentage of dynamic instructions inside atomic regions in the program, the number of dynamic atomic regions, and the average dynamic size of an atomic region in instructions, respectively. We can see from this data that our atomic regions cover the great majority of the execution (74% of the dynamic instructions on average). The remaining execution largely contains private references. We also see that there are many dynamic atomic regions and that they are very large — about 52,000 dynamic instructions on average. These atomic regions are largely loops with small to modest write footprints. The average atomic region size for *JVM-compress* is smaller than the others. This is because system calls interspersed across this application force the creation of smaller atomic regions. At this size, the overhead of our JNI calls becomes more significant and, hence, we suffer a 6% overhead as can be seen in Figure 4.7, even with a negligible squash rate.

Columns 5–7 give more information about these atomic regions, namely the number of synchronized blocks per region in the original code, and their write and read footprints in number of 64-byte lines, respectively. We can see that, on average, each atomic region used to contain about 600 synchronized blocks. By transforming their synchronization operations into plain memory accesses, we enable many optimizations in Hotspot. As mentioned in Section 2.6.2, *JVM-raytrace*, *JVM-mtrt*, and *JVM-compress* do not have much synchronization and, therefore, show no speedups.

We also see that the atomic regions have a small write footprint (184 lines on average). This allows them to fit inside the cache without overflows. The read footprint is larger, but recall that, in BulkSC the read footprint *does not need to remain in the cache* — signatures keep a record of the lines read [15].

For example, *JVM-db* has a large read footprint but a tiny write footprint compared to the size

of its atomic regions. This is due to the fact that *JVM-db* spends the bulk of its time sorting its database index, which involves string comparisons of index entries and swaps when entries are out of order. The index is only updated on swaps, which are much less frequent than the number of read accesses required for the string comparisons. This is the reason for the small write footprint. However, each access to the index is protected by a synchronized block, giving Bulk-Compiler ample optimization opportunities. Other applications follow a similar pattern.

Finally, the last column shows the fraction of dynamic instructions in atomic regions that get squashed. We see that, on average, only 0.48% of the instructions in atomic regions get squashed. This represents a tolerable fraction of work lost.

2.7 Discussion

This research can be applied to to improve the performance of other memory consistency models beyond SC. We are confident that our techniques can improve the performance of relaxed memory models as well. Work by Wenisch *et al* [92] and Blundell *et al* [9] point to the potential of these ideas.

Also, this work is applicable beyond BulkSC to all architectures that support continuous chunked execution such as TCC [35] and, with some extensions, to conventional architectures that support hardware TM.

2.8 Related Work

2.8.1 Software-Only Sequential Consistency

There have been three major software-only efforts to enforce SC in programs that are not well synchronized. The most sophisticated one is the Pensieve Project [85], which provides SC for Java. Their SC compiler uses a combination of escape analysis [85], thread-structure analysis [85], delay set analysis [78, 85], and an optimized fence-insertion algorithm [31]. All but

the fence-insertion algorithm are interprocedural analyses that are fairly complex. Overall, their method induces slowdowns of over 10% on average over the relaxed Java Memory Model.

Liblit *et al* [49] developed an SC version of Titanium [41]. In the same project, Krishnamurthy and Yelick [42] showed how the regular structure of SPMD programs could be exploited to reduce the complexity of delay set analysis in those programs. Finally, Von Praun and Gross [90] used an object-based analysis for delay set analysis to determine reference orders that needed to be enforced because of inter-thread conflicts. Overall, none of these methods reported speedups for applications, and some reported significant slowdowns in one or more applications. In contrast, our combined hardware-software SC scheme delivers speedups over the relaxed Java Memory Model.

2.8.2 Exploiting Support for Atomicity

There has been substantial recent work on exploiting hardware support for atomicity. The Transmeta Code Morphing concept involved aggressively optimizing the code with speculative transformations [26]. It appears that most of the optimizations were for single-thread execution. Neelakantam *et al* [64] sped-up hot sections of the code by developing an optimized, speculative "trace" of the code and running it under hardware atomicity. If the code takes an unexpected control path, the section is squashed and the full version of the code is executed. They largely focus on optimizing single-thread execution, typically in loop iterations, although they mention the application of SLE to critical sections. BulkCompiler differs in its emphasis on grouping many low-contention critical sections in a large atomic region to enable conventional optimizations. It also differs in its goal to support SC.

Carlstrom *et al* [13] take lock-based Java programs and convert them into transactions. They describe how critical sections and other constructs are converted into transactions. However, they neither mention whether this change enables compiler optimizations nor are they focused on SC. Other authors such as Ziarek *et al* [96] and Rossbach *et al* [74] have studied environments that integrate locks and transactions, finding some of the problems we faced.

Re-writing a critical section with a synchronization-free fast path executing under atomic hardware, and a slow path with the complete code has been proposed in SLE [70] and used in TM libraries [28].

2.9 Conclusions

A platform that provides high-performance SC at the hardware and software levels for all codes, including those with data races, will substantially simplify the task of programmers. This work presented the hardware-compiler interface, and the main ideas for *BulkCompiler*, a compiler layer that works with the BulkSC chunking hardware to provide a *whole-system high-performance SC platform*. Our specific contributions included: (i) ISA primitives for BulkCompiler to interface to the chunking hardware, (ii) compiler algorithms to drive chunking and code transformations to exploit chunks, and (iii) initial results of our algorithms on Java programs.

Our results used Java application suites modified with our compiler algorithms and compiled with Sun's Hotspot server compiler. A whole-system SC environment with BulkCompiler and simulated BulkSC hardware outperformed a simulated conventional hardware platform that used the more relaxed Java Memory Model by an average of 37%. The speedups came from code optimization inside software-assembled instruction chunks.

This work is applicable beyond BulkSC to all group-commit architectures and, with some extensions, to conventional architectures that support hardware TM. We are now extending Bulk-Compiler to drive novel compiler optimizations for single- and multi-threading, and to apply them to relaxed memory models as well.

Chapter 3

Software-Exposed Signatures for Code Analysis and Optimization

3.1 Introduction

Many code analysis techniques need to ascertain at runtime whether or not two or more variables have the same address. Such runtime checks are the only choice when the addresses cannot be statically analyzed by the compiler. They provide crucial information that is used, for example, to perform various code optimizations, support breakpoints in debuggers, or parallelize sequential codes.

Given the frequency and cost of performing these checks at runtime, there have been many proposals to perform some of them in hardware (e.g., [32, 40, 43, 50, 69, 81]). Such proposals have different goals, such as ensuring that access reordering within a thread does not violate dependences, providing multiple hardware watchpoints for debugging, or detecting violations of inter-thread dependences in Thread-Level Speculation (TLS). The expectation is that hardware-supported checking (or "disambiguation") of addresses will have little overhead.

A straightforward implementation of hardware-supported disambiguation can be complex and inefficient. A key reason is that it typically works by comparing an address to an associative structure with other addresses. For example, in TLS, when a processor writes, its address is checked against the addresses in the speculative buffers (or caches) of other processors. Similarly, in intra-thread access reordering checkers (e.g., [32]), the address of a write is checked against later reads that have been speculatively scheduled earlier by the compiler. In general, longer windows of speculation require larger associative structures.

To improve efficiency, we would like to operate on sets of addresses at a time, so that, in a

single operation, we compare many addresses. This can be accomplished with low complexity with hardware signatures [8]. In this case, addresses are encoded using hash functions and then accumulated into a signature. If we provide hardware support for signature intersection as in Bulk [14], then address disambiguation becomes simple and fast.

Signatures have been proposed for address disambiguation in various situations, such as in load-store queues (e.g., [77]) and in TLS and Transactional Memory (TM) systems (e.g., [14, 58, 95]). Typically, signatures are managed in hardware or have only a simple software interface [58, 95]. However, to be truly useful for code analysis and optimization techniques, signatures would need to provide a rich interface to the software.

To enable flexible use of signatures for advanced code analysis and optimization, this work proposes to expose a Signature Register File to the software through a sophisticated ISA. The software has great flexibility to decide: (i) what stream of memory accesses to collect in each signature, (ii) what local or remote stream of memory accesses to disambiguate against each signature, and (iii) how to manipulate each signature. We call this architecture *SoftSig*, and describe the processor extensions needed to support it.

In addition, as an example of SoftSig use, this work proposes an algorithm to detect redundant function calls efficiently and eliminate them dynamically. We call this memoization algorithm *MemoiSE*. Our results show that, on average for five popular multithreaded and sequential applications, MemoiSE reduces the number of dynamic instructions by 9.3%, thereby reducing the average execution time of the applications by 9%.

This chapter is organized as follows: Section 3.2 presents a background; Section 3.3 presents the SoftSig idea; Sections 3.4 and 3.5 present SoftSig's software interface and architecture; Section 3.6 describes MemoiSE; Section 3.7 evaluates MemoiSE; and Section 3.8 presents related work.

3.2 Background on Memoization

Memoization is a technique that uses the basic observation that a function (or expression) that is called twice with the same inputs will compute the same result. Consequently, rather than computing the same result again, memoization involves storing the outcome in a lookup table and, on future occurrences of the function, simply returning the answer provided by the lookup table. Michie [57] first proposed memoization as a general way to avoid computing redundant work, and it is routinely applied in dynamic programming [22] and functional programming languages.

For memoization to be profitable, a function must be called with the same inputs often, to ensure a high hit rate in the lookup table. Also, the cost of the lookup must be less than that of executing the function. Not all potentially profitable functions can be memoized, however, since most functions in imperative languages like C have side effects or reads from nonlocal memory which are extremely hard to analyze statically [80]. In these cases, traditional memoization cannot be used.

3.3 Idea: Exposing Signatures to Software

3.3.1 Basic Idea

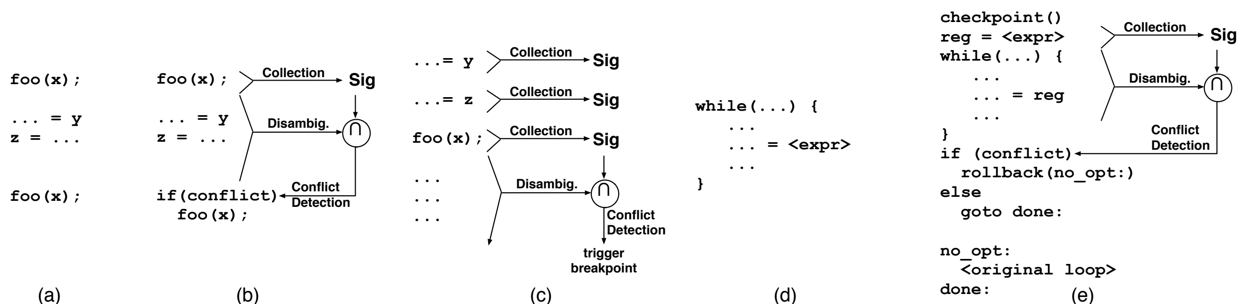


Figure 3.1: Three examples of how to use hardware signatures that are manipulatable in software.

Many code analysis and optimization techniques, debugging schemes, and operations in speculative multithreading require the runtime disambiguation of multiple memory addresses — ei-

ther accessed by a single thread or by multiple threads. We can significantly advance the art in these techniques if we support an environment where hardware signatures are flexibly manipulatable in software.

Such an environment must support three main operations: collection of addresses, disambiguation of addresses, and conflict detection. The software has a role in each of them. In *Collection*, the software specifies the window of program execution whose memory accesses must be recorded in a signature — i.e., the set of program statements to be monitored, possibly with some restriction on the range of addresses to be recorded. Moreover, it specifies whether reads, writes, or both should be collected.

In *Disambiguation*, the software specifies that the addresses collected in a given signature be compared to the dynamic stream of addresses accessed by (i) the local thread, (ii) other threads (visible through coherence messages such as invalidations), or (iii) both. Again, it also specifies whether reads and/or writes should be examined.

Finally, in *Conflict Detection*, the software specifies what action should be taken when the stream being monitored accesses an address present in the signature. The action can be to set a bit that the software can later check, or to trigger an exception and jump to a predefined location — possibly undoing the work performed in the meantime.

3.3.2 Examples

Figure 3.1 shows three examples of how this environment can be used: function memoization (Charts (a) and (b)), debugging with many watchpoints (Chart (c)), and Loop Invariant Code Motion (LICM) (Charts (d) and (e)). Function memoization involves dynamically skipping a call to a function if it can be proved that doing so will not affect the program state. As an example, Figure 3.1(a) shows two calls to function `foo` and some pointer accesses in between. Suppose that the compiler can determine that the value of the input argument is the same in both calls, but is unable to prove whether or not the second call is dynamically redundant — due to non-analyzable memory references inside or outside `foo`. With signatures (Figure 3.1(b)), the com-

piler enables address collection over the first call into a signature, and then disambiguation of accesses against the signature until the next call. Before the second call, the code checks if the signature observed a conflict. If it did not, and no write in `foo` overwrites something read in `foo`, then the second invocation of `foo` can be skipped.

A desirable operation when debugging a program is knowing when a memory location is accessed. Debuggers offer this support in the form of a "watch" command, which takes as an argument an address to be watched, or watchpoint. Some processors provide hardware support to detect when a watchpoint is accessed (e.g., [40]). However, due to the hardware costs involved, only a modest number of watchpoints is supported (e.g., 4). With signatures, a large number of addresses can be simultaneously watched with very low overhead. As an example, Figure 3.1(c) collects addresses y and z in a signature. Then, it collects into the signature all the addresses that are accessed in `foo`. After that, it disambiguates all subsequent accesses against the signature, triggering a breakpoint if a conflict is detected. The system is watching for accesses to any of the addresses collected.

Finally, Figures 3.1(d) and (e) show an example of LICM. Figure 3.1(d) shows a loop that computes an expression at every iteration. If the value of the expression remains the same across iterations, it would offer savings to move the computation before the loop. However, the code may contain non-analyzable memory references that prevent the compiler from moving the code. With signatures and checkpointing support, the compiler can transform the code as in Figure 3.1(e). Before the loop, a checkpoint is generated, and the expression is computed and saved in a register while collecting the addresses into a signature. Then, the loop is executed without the expression, while disambiguating against the signature. After the loop, the code checks if the signature observed a conflict. If it did, the state is rolled back to the checkpoint and execution resumes at the beginning of the unmodified loop.

3.3.3 Design Overview and Guidelines

To expose hardware signatures to software, we extend a conventional superscalar processor with a *Signature Register File* (SRF), which can hold a signature in each of its *Signature Registers* (SRs). Moreover, we add a few new instructions to manipulate signatures, enabling address collection, disambiguation, and conflict detection. We call our architecture *SoftSig*. Before describing SoftSig, we outline some design guidelines that we follow. The guidelines are listed in Table 3.1.

G1	Minimize SR accesses and copies
G2	Manage the SRF through dynamic allocation
G3	Imprecision should never compromise correctness
G4	Manage imprecision to provide the most efficiency
G5	Minimize imprecision and unnecessary conflicts

Table 3.1: Design guidelines in SoftSig.

Signature Registers are Unlike General Purpose Registers

SRs must be treated differently than General Purpose Registers (GPRs) because they are different in two ways. First, an SR is much larger than a 64-bit GPR — SRs are 1 kilobit in SoftSig. Due to their size, SRs are costly to read, move and copy. Second, SRs are persistent. Once a SR begins collecting or disambiguating, it must remain in the SRF for the duration of the operation in order to work as expected. An operation may take a very long time to complete, as can be seen from the examples in Figure 3.1.

These observations motivate two design guidelines:

G1: Minimize SR accesses and copies. Given the size of SRs, it is important to minimize SR accesses and copies. Every move typically takes several cycles, while accessing the SRF consumes power. Consequently, we minimize any negative impact on execution time or power consumption through several measures. First, on a context switch, the system does not save or re-

store SRs; rather, signatures are discarded. Second, the compiler never spills SRs to the stack. Finally, we design the logic to minimize reading SRs from the SRF. While these measures may appear to be severe limitations, our approach works well in spite of them.

G2: Manage the SRF through dynamic allocation. Given the size of SRs, there are few of them. Moreover, given their persistence, their use must be coordinated across an entire program's execution. This introduces the issue of how to assign SRs so that (i) we enable as many uses as possible in the program and (ii) we use them where they are most profitable.

To maximize the number of uses, it is better to allocate the SRs dynamically than to reserve the SRs based on static compiler analysis. For a given number of potential SR uses in a program, it may be difficult for the compiler to determine whether or not the lifetimes of these uses will overlap in time during execution. Consequently, the compiler may have to assume the worst case of maximum lifetime overlap, and refrain from exploiting all opportunities. Dynamic allocation, on the other hand, uses dynamic information on the actual use lifetime to exploit as many opportunities at a time as SRs are available. This approach requires software routines or hardware logic to examine the current state of the SRF and decide whether a SR can be allocated.

As an example, Figure 3.2 shows the case of two SRs and a program with four uses with hard-to-predict lifetimes. If we allocate SRs statically, we can only cover two uses. In practice, these two uses do not overlap in time (Chart (a)). If, instead, SRs are allocated dynamically, since at most two uses overlap in time, we can cover the four uses.

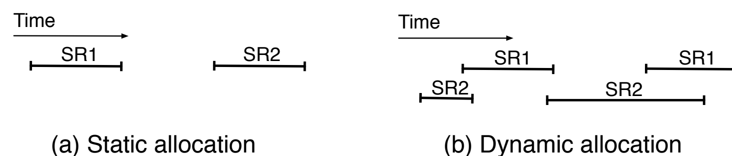


Figure 3.2: Employing SR1 and SR2 in uses whose lifetime (length of the segment) is unpredictable statically.

We leave the problem of deciding which uses of SRs are most worthwhile to the compiler, programmer, or a feedback-directed optimization framework.

System Must Cope with Imprecision

SoftSig must cope with multiple forms of imprecision. One form of imprecision is the encoding of signatures. Instead of an exact list of addresses, only a superset of addresses is actually known. Because of this, conflicts may be reported even when an exact list would show that there were none. Such conflicts are called *false positives*.

Another source of imprecision is the dynamic de/allocation policy of the SRF. Signatures may be silently displaced while in use. Consequently, an optimization can fail simply because of how the SRF is managed at runtime.

The presence of imprecision motivates three design guidelines:

G3: Imprecision should never compromise correctness. The system must be designed such that imprecision hurts at most performance and never correctness. Therefore, any software that uses an SR must be prepared to cope with a conflict that turns out to be a false positive. For instance, consider the watchpoint example in Figure 3.1(c). A conflict may not be the result of an access to a watched location. The software needs to handle this case gracefully.

In addition, to handle the case of the unexpected deallocation of an in-use SR, SoftSig makes this event appear as if a conflict had occurred. Since the code must always work correctly in the presence of false positive conflicts, this approach will always be correct.

G4: Manage imprecision to provide the most efficiency. Some imprecision can be managed in software by controlling how SoftSig is used. Specifically, many false positives may indicate that SRs are too full and do not have enough precision. Using SRs over shorter code ranges or finding a way to filter some of the addresses are effective solutions to manage imprecision. SoftSig provides an instruction for filtering (Section 3.4).

In addition, many SR deallocations indicate competing uses for the SRF. In this case, profiling can help determine the subset of SR uses that are most profitable. Software should judiciously manage both of these effects to provide the most efficiency.

Category	Instruction	Description
Collection	bcollect.(rd,wr,r/w) R1	Begin collecting addresses into SRF[R1]. Depending on the specifier, collect only reads, writes, or both
	ecollect R1	End collecting addresses into SRF[R1]
	filtersig R1,R2,R3	Do not collect or disambiguate addresses between R2 and R3 into/against SRF[R1]
Disambiguation	bdisamb.(rd,wr,r/w).(loc,rem) R1	Begin disambiguating local or remote accesses (depending on the specifier) against SRF[R1]. Depending on the specifier, disambiguate only reads, writes, or both
	edisamb.(loc,rem) R1	End disambiguating local or remote accesses (depending on the specifier) against SRF[R1]
Persistence, Status, & Exceptions	allocsig R1,R2	Allocate register SRF[R2] and return its Status Vector in R1
	dallocsig R1	Deallocate SRF[R1]
	sigstatv R1,R2	Return the Status Vector of SRF[R2] in R1
	exptsig R1,*target*	Except to *target* if a conflict occurs on SRF[R1]
Manipulation	ldsig R1,*addr*	Load from *addr* into SRF[R1]
	stsig R1,*addr*	Store SRF[R1] to *addr*
	mvsig R1,R2	SRF[R1] ← SRF[R2]
	clrsig R1	SRF[R1] ← $\emptyset$, clear Status Vector and set a=1,z=1,x=0
	union R1,R2,R3	SRF[R1] ← SRF[R2]∪ SRF[R3]
	insert_elem R1,R2	SRF[R2] ← R1 ∪ SRF[R2]
	member R1,R2,R3	R1 ← (R2 ∈ SRF[R3]) ? 1 : 0
	intersect R1,R2,R3	SRF[R1] ← SRF[R2] ∩ SRF[R3]

Table 3.2: SoftSig software interface.

G5: Minimize imprecision and unnecessary conflicts. Signatures will always have imprecision due to their hash-based implementation. However, to minimize additional sources of imprecision, the hardware must support address collection and disambiguation at precise instruction boundaries. Also, to minimize unnecessary conflicts, disambiguation should be performed only against the addresses that are strictly necessary for correctness. In so doing, the number of unnecessary conflicts will decrease.

3.4 SoftSig Software Interface

Based on the previous discussion, this section describes SoftSig's software interface.

3.4.1 The Signature Register File (SRF)

A core includes an SRF, which holds a set of SRs. SRs are not saved and restored at function calls and returns. Instead, SRs have persistence — they are allocated when needed and, in normal circumstances, deallocated only when they are not needed anymore. Consequently, when an SR is allocated, it is assigned a *Name* specified by the program. Such a name is used to refer to it until deallocation. The instructions used to manipulate SRs constitute SoftSig's software interface. They are shown in Table 4.1 and discussed next.

3.4.2 Collection

Collection is the operation that accumulates into an SR the addresses of the memory locations accessed during a window of execution. SoftSig supports collection using two instructions, namely `bcollect` and `ecollect`. When `bcollect` is executed, address collection begins. Depending on the instruction suffix, it will collect only reads (rd), only writes (wr), or both reads and writes (r/w). When `ecollect` is executed, address collection ends. Both instructions take as argument a general purpose register (GPR) that contains the name of the SR.

For some optimizations, it is important to skip collection over a range of addresses that the compiler can guarantee need not be considered. This is supported with the `filtersig` instruction. Its inputs are the name of the SR, and the beginning and end of the range — specified using virtual addresses.

3.4.3 Disambiguation

Disambiguation is the operation that checks for conflicts between addresses being accessed and a signature that has been collected or is currently being collected. SoftSig supports disambiguation using two instructions, namely `bdisamb` and `edisamb`. The former begins disambiguation, while the latter ends it. Both instructions take as argument a GPR that contains the name of the SR. They demarcate a code region during which the hardware continually checks addresses for

conflicts with the signature.

Disambiguation can be configured in many ways. One category of specification is whether the signature is disambiguated against accesses issued by the local processor or by remote ones. While the examples in Figure 3.1 all used local disambiguation, remote disambiguation is useful in a multithreaded program to identify when other threads issue accesses that conflict with those in a local signature. In addition, disambiguation can be configured to occur in only reads, only writes, or both reads and writes. As we will see, remote disambiguation relies on the cache coherence protocol to flag accesses by remote processors. Consequently, signatures only observe those remote accesses that cause coherence actions in the local cache — e.g., remote reads to a location that is only in shared state in the local cache will not be seen.

In some cases, it may be desirable to disambiguate accesses performed by remote processors against a local signature that is currently being collected. This often occurs under HTM or TLS. In this case, we first need to use `bdisamb` to begin disambiguation and then `bcollect` to begin collection. Swapping the order of these two instructions is unsafe because it results in a window of time when conflicts can be missed.

When disambiguation is enabled and the hardware detects a conflict with a signature, the hardware records it in a *Status Vector* associated with the signature. Later in this section, we will show how the interface specifies the actions to take on a conflict.

The `filtersig` instruction blocks disambiguation over its range of addresses for the specified signature.

3.4.4 Persistence, Signature Status, and Exceptions

The `allocsig` and `dallocsig` instructions allocate and deallocate, respectively, an SR in the SRF. Each instruction takes as an argument a GPR that holds the name of the SR. `allocsig` further takes a second GPR that returns the Status Vector of the SR. Finally, `allocsig` always allocates an SR, even if it requires silently displacing an existing signature. Consequently, any code optimization that uses SRs must be wary of the hardware displacing a signature it is relying

upon.

The `sigstatv` instruction returns the Status Vector of an SR in a GPR. Figure 3.3 shows the fields of the vector. For now, consider the three 1-bit fields in the figure that are unshaded. They describe whether the signature is currently allocated (*a*), is zero (*z*), or has recorded a conflict (*x*). If `sigstatv` is called on a deallocated signature, a default Status Vector is returned with *a*=0, *z*=0, and *x*=1. Consequently, all code optimizations have to be implemented under the assumption that this default vector means that the signature cannot be trusted to hold meaningful results.

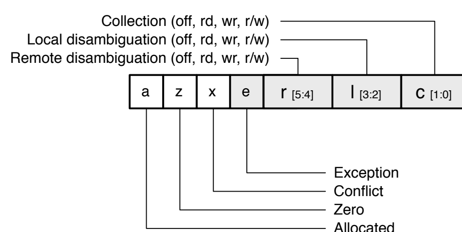


Figure 3.3: Status Vector associated with a signature.

While this makes it possible to generate code that will always function correctly, it would be inefficient to require a Status Vector check before every signature operation. Therefore, SoftSig supplies an additional simplifying policy: a disambiguation or collection operation on a deallocated signature is converted into a NOP.

The `sigstatv` instruction makes it possible to explicitly query for the presence of a conflict. However, it is not always desirable to schedule an instruction to test for a conflict. Consider the case of watchpoints in Figure 3.1(c) — any conflict should be reported immediately when it occurs. To enable such behavior, SoftSig provides the `exptsig` instruction. `exptsig` specifies an exception handler that should be triggered when a conflict occurs on a specific SR. `Exptsig` takes as arguments the SR and the address of the first instruction of the exception handler.

The shaded fields in the Status Vector shown in Figure 3.3 supply additional configuration information that is only valid if the signature is allocated. Starting from the right, the fields show the status of collection, and of local and remote disambiguation. The status can be off, only reads,

only writes, and reads plus writes. The *e* bit indicates whether an exception should be generated on a conflict.

3.4.5 Signature Manipulation

SoftSig provides a set of operations to manipulate signatures. Table 4.1 lists them. Since they are straightforward, we leave it to the reader to understand their use from the description in the table.

3.4.6 Interaction with Checkpointing

As shown Figures 3.1(a)-(c), SoftSig is useful without the need for machine checkpoints. However, if the system supports checkpointing — either in software or in hardware — SoftSig can provide additional functionality. Specifically, it can enable efficient execution of optimizations where the code performs some risky operation speculatively and then tests whether the execution was correct. If it was not, execution is rolled back.

Figures 3.1(d)-(e) showed an example of speculative optimization. The expression is assumed loop invariant and hoisted before the loop. After the loop is executed, there is a check to see if the assumption was correct. If it was not, the checkpoint is restored and the original loop is executed.

Note, however, that SoftSig's applicability is not limited to speculative environments.

3.4.7 Managing Signature Registers

It is necessary that each SR have a unique name. If two SRs had the same name, they could be confused with one another and lead to incorrect programs. Within a thread, the compiler can typically guarantee that each dynamic SR instance has a different name. For example, it can derive the name at allocation time based on the address of the function that allocates the SR.

This approach, however, does not guarantee that names are unique across different threads or processes time-sharing a processor. One possible solution is to include the thread or process ID

as part of the name of the SR. This approach works well for SMT processors. For single-threaded processors, by simply invalidating the SRF at context switches as per guideline **G1**, we eliminate any possible confusion between SR instances.

3.5 SoftSig Architecture

The SoftSig architecture consists of several extensions to a superscalar processor. As shown in Figure 3.4, the extensions are grouped into a *SoftSig Processor Module* (SPM), which contains the Signature Register File (SRF), the Status Vectors, FUs to operate on signatures, the exception vectors, and a module called the In-flight Conflict Detector (ICD), which aids remote disambiguation. The SPM interacts with the Reorder Buffer (ROB) and the Load-Store Queue (LSQ) in support of collection and disambiguation. The rest of this section describes the architecture in detail.

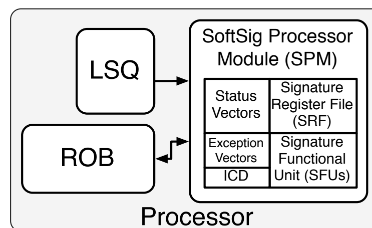


Figure 3.4: SoftSig architecture.

3.5.1 SoftSig Instruction Execution

In our design, SoftSig instructions execute only when they reach the head of the ROB. Therefore, SRs are neither renamed nor updated by speculative instructions or updated out of order. We choose this approach to follow guidelines **G1**, **G2**, and **G5** in Section 3.3.3. Indeed, if we allowed speculative instructions to update SRs, every speculative instruction that updated an SR would have to make a new copy of the SR, in order to be able to support precise exceptions. The additional accesses and copies required would run counter to guideline **G1**.

In addition, allowing speculative instructions to update SRs would induce a larger number of in-use SRs. This is at odds with guideline **G2**, which prescribes that the SRs should be allocated and deallocated dynamically in the most efficient manner. Finally, allowing out-of-order update of the SRs would make it hard to maintain precise boundaries in the code sections where signatures are collected or disambiguated. The signatures would then be more imprecise, which would hurt guideline **G5**.

However, executing SoftSig instructions only when they reach the head of the ROB has two disadvantages. First, some non-SoftSig instructions may have data dependences with SoftSig instructions — for example, instructions that check the Status Vector. Such instructions will have to wait for the SoftSig ones to execute. However, thanks to out-of-order execution, other, independent instructions can continue to execute. The second disadvantage is that remote disambiguation does not work correctly in this environment unless the ICD module is added. Section 3.5.5 presents this problem in more detail and describes our solution.

3.5.2 Signature Register File

As shown in Figure 3.5, the SRF is composed of three modules, namely the Signature Register Array, the Operation Select, and the Signature Encode. The former contains all the SRs, and has a read (*Sig_Out*) and a write (*Sig_In*) port. In turn, each SR has an input (*In*) and an output (*Out*) data port, control signals for union with the input (∪), intersection with the input (∩), read (*Rd*), and write (*Wr*), and output signals that flag a conflict (*Conflict[i]*) or a zero SR (*Zero[i]*).

The Operation Select module generates the control signals for the SRs. Specifically, it can set the Collect (*C[i]*), Disambiguate (*D[i]*), Read (*R[i]*), or Write (*W[i]*) signals for one or more SRs simultaneously. To generate these signals, it takes as inputs the Status Vectors of all the SRs and, if applicable, the type of operation to perform (*Op*), the name of the SR to operate on (*Name*), and the virtual address of the local access (*VirtAddr*). The latter is needed in case we need to filter ranges of addresses.

Finally, the Signature Encode module takes a physical address and transforms it into a signa-

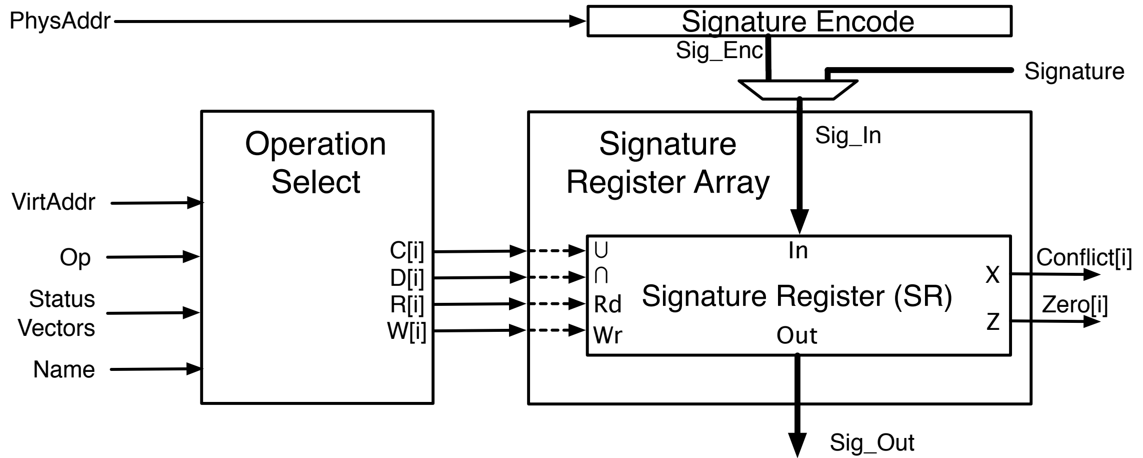


Figure 3.5: The signature register file.

ture (*Sig_Enc*). Either *Sig_Enc* or an explicit signature can be routed into the Signature Register Array for collection, disambiguation, or writing.

3.5.3 Allocation and Deallocation

When an `allocsig` instruction reaches the head of the ROB, the hardware attempts to allocate an SR. If an SR with the same name is already allocated, no action is performed. Otherwise, an SR is cleared, its Status Vector is initialized, and the SR name is stored in the Operation Select module.

If there is no free SR, then one is selected for displacement. The system tries to displace an SR that has its Conflict bit set. If no such SR exists, then an SR is selected at random. In either case, the name of the deallocated SR is removed from the Operation Select module.

When a `dallocsig` instruction reaches the head of the ROB, the hardware deallocates the corresponding SR. This operation involves removing the SR name from the Operation Select module.

3.5.4 Collection and Local Disambiguation

When a `bcollect` or a `bdisamb.loc` instruction reaches the head of the ROB, the hardware notifies the LSQ to begin sending to the SoftSig Processor Module (SPM) the address (virtual and physical) and type of access of all memory operations as they retire. In addition, the appropriate bits in the corresponding Status Vector are set. As addresses are streamed into the SPM, they are handled by the SRF as described previously.

If no conflict is detected on a memory operation, the ROB is notified that the corresponding instruction can retire; otherwise, the Conflict signal is raised and, depending on the configuration, an exception may be generated (Section 3.5.6).

When an `ecollect` or an `edisamb.loc` instruction reaches the head of the ROB, the corresponding Status Vector is updated. When both collection and local disambiguation have terminated for all SRs, the LSQ does not forward state to the SPM any longer.

3.5.5 Remote Disambiguation

The `bdisamb.rem` instruction enables the SPM to watch the addresses of external coherence actions, while the `edisamb.rem` terminates this ability — if no other SR is performing remote disambiguation. Both instructions also update the Status Vector of the corresponding SR. As usual, `edisamb.rem` performs its actions when it reaches the head of the ROB. However, `bdisamb.rem` is different in that, for correctness, it needs to perform some of its actions earlier. In the following, we consider why this is the case and how we ensure correct remote disambiguation.

Correctly Supporting Remote Disambiguation

The challenging scenario occurs when SoftSig performs address collection and remote disambiguation on the same SR simultaneously. This is a common situation in HTMs. In this case, to eliminate any window of vulnerability where a conflicting external coherence action could be

missed, we must enclose the `bcollect` and `ecollect` instructions inside the region bounded by `bdisamb.rem` and `edisamb.rem` instructions. This is shown in Figure 3.6(a), which also includes a load to variable *X* inside the code section being collected and remotely disambiguated.

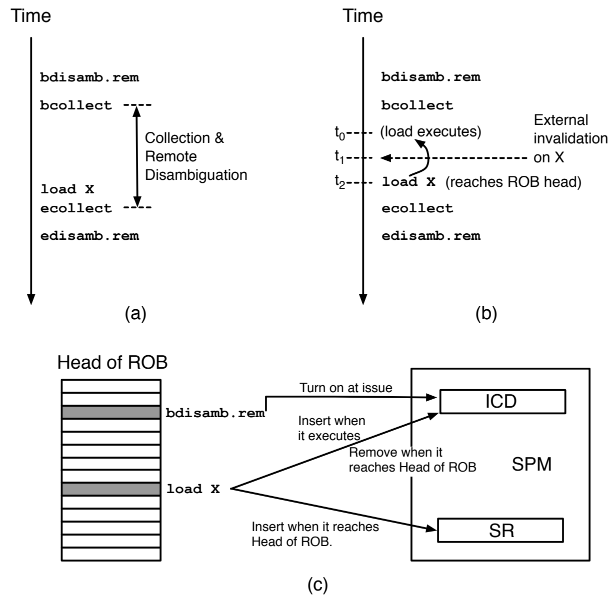


Figure 3.6: The ICD prevents missing a remote conflict.

However, as shown in Figure 3.6(b), due to out-of-order execution, the load may execute at time t_0, which is before it reaches the head of the ROB (and updates the SR) at time t_2. Unfortunately, if an external invalidation on *X* is received at time t_1 — in between the time the load reads at t_0 and the time it updates the SR at t_2 — the conflict will be missed. Note that we cannot assume that the consistency model supported by the processor will force the retry of the load to *X*.

This inconsistency occurs because loads read data potentially much earlier than they update the SR. To solve this problem, we add the *In-flight Conflict Detector* (ICD) to the SPM, and require that `bdisamb.rem` perform most of its actions in the (in-order) issue stage. More specifically, the ICD is a counter-based Bloom filter that automatically accumulates the addresses of all *in-flight* loads. As shown in Figure 3.6(c), when the load executes, *X* is inserted into the ICD. When the load reaches the head of the ROB, *X* is inserted into the SR and, since the ICD is counter-based, it is removed from the ICD. If remote address disambiguation is enabled, any ex-

ternal coherence action is disambiguated against both the SR and the ICD. If a conflict is found on either the ICD or the SR, then the SR is flagged as having a conflict. In the example shown, the conflict will be detected by the ICD as soon as the invalidation is received.

Moreover, we must ensure that `bdisamb.rem` turns the ICD on before any subsequent load could be executed. Consequently, we conservatively require that `bdisamb.rem` turn the ICD on and start directing external coherence addresses to the SPM as soon as the `bdisamb.rem` instruction goes through the (in-order) issue stage. This is shown in Figure 3.6(c). However, `bdisamb.rem` does not update the Status Vector until it reaches the head of the ROB. This is because only then can we guarantee that the corresponding `allocsig` instruction has retired.

Based on this design, the full behavior of the ICD is as follows. When `bdisamb.rem` is issued, the ICD is turned on. When a load executes, its address is added to the ICD; when a load reaches the head of the ROB or is found to be misspeculated, its address is removed from the ICD. If an external coherence action has a conflict with the ICD, the ICD sets a flag indicating a conflict and remembers the ROB index of the youngest load instruction i that has executed so far. All SRs that are collecting and performing remote disambiguation from this point until i retires will have their Conflict bit set in their Status Vector. Once i retires, the ICD clears its conflict flag — since any SR that starts collection after i should not be affected by this conflict. The ICD remains active from the time the first `bdisamb.rem` is issued until no signatures perform remote disambiguation anymore.

Handling Cache Displacements under Remote Disambiguation

A final challenge to supporting remote disambiguation involves cache displacements. The problem is that a cache is only guaranteed to see external coherence actions on those addresses that it caches. If the cache displaces a line, the cache may not see future coherence actions by other processors on that particular line. Therefore, consider a processor that performs both collection and remote disambiguation on an SR. Suppose that the processor references a line, inserts its address in the SR, and then displaces the line from the cache. Future coherence actions by other proces-

sors on that line may not be seen by the cache and, therefore, remote disambiguation cannot be trusted to identify all remote conflicts.

To prevent this case, when remote disambiguation against an SR is in progress, the hardware takes a special action when a line is displaced from the cache. Specifically, the line's address is disambiguated against the SR, as if the cache had received an external invalidation on that line. This approach may conservatively generate a non-existing conflict. However, it will never result in missing a real conflict.

A more expensive, alternative approach would involve explicitly preventing the displacement of lines whose address are collected in the SR — for as long as remote disambiguation is in progress. This is the approach used in HTM and TLS systems. We do not support this approach.

3.5.6 Exceptions

When a conflict is detected on an SR, an exception may be triggered. The SPM supports registering exception handlers in a table, as shown in Figure 3.4 as *Exception Vectors*. If an exception is registered for a given SR, it is triggered when a conflict is detected. For a conflict caused by a local access, a precise exception is generated. For a conflict caused by an external coherence action, the handler is called as soon as the ICD or the SR detect the conflict.

When an exception is raised, the SPM notifies the processor's front end of the target address and informs the ROB of the instructions that need to be flushed. The exception handler then pushes the return address into a register and disables the handling of additional exceptions. Since other SRs may still be under disambiguation, additional exceptions are buffered and serviced sequentially.

3.6 MemoiSE: Signature-Enhanced Memoization

Memoization has been used to replace redundant or precomputed function calls with their outputs [57]. However, in languages such as C and C++, function memoization is hard to apply

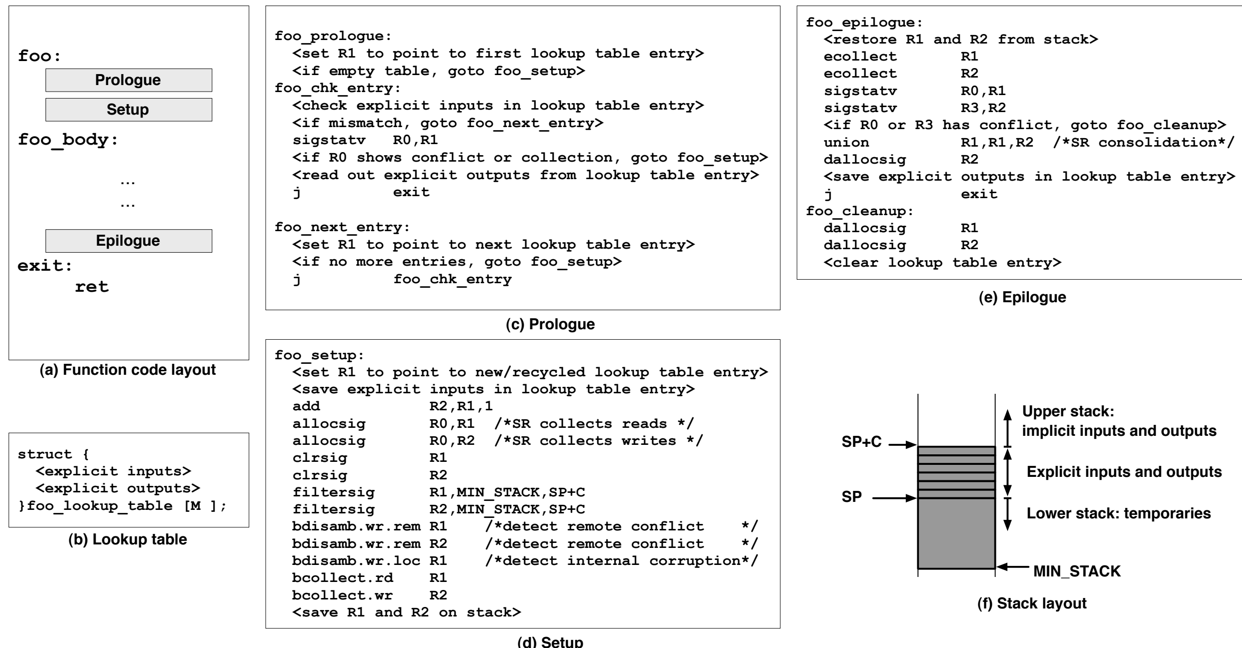


Figure 3.7: Applying the MemoiSE algorithm to function `foo`: function code layout (a), lookup table (b), Prologue (c), Setup (d), Epilogue (e), and stack layout (f).

because memory state is often changed through non-analyzable pointer accesses. Using SoftSig, however, we propose a very general, low-overhead, and effective approach to increasing the number of function calls that can be memoized. We call our approach *MemoiSE*, for Signature-Enhanced memoization. In this section, we describe MemoiSE's general approach, the MemoiSE algorithm, and some optimizations to reduce its overhead.

3.6.1 A General Memoization Framework

Memoization algorithms work by caching the values of the inputs and outputs of a function in a lookup table. When the function is next invoked, the lookup table is searched for an entry with an identical set of input values. If such an entry is found, the output values are copied out of the lookup table into the appropriate locations (memory or registers), and the function execution is skipped.

Unfortunately, a function's inputs and outputs are not just the explicit input arguments passed to the function and the explicit output arguments returned by the function. They also include im-

plicit inputs and outputs. These are other variables that the function reads from memory or writes to memory. To build a generic memoization algorithm, both explicit and implicit inputs and outputs need to be considered.

A naive approach would log the values of all implicit inputs and outputs in the table, in the same way as explicit inputs and outputs are logged. Unfortunately, implicit inputs and outputs cannot always be determined statically by the compiler, and there can be a very large number of them.

With MemoiSE, we do not log implicit inputs and outputs. Instead, we note that, if none of the implicit inputs or outputs have been written to since the end of the previous invocation of the function, then they have the same values. Such a condition can be easily checked using SoftSig. Indeed, during the initial execution of the function, we collect the addresses of all the implicit inputs and outputs in signatures. After the function completes, as the processor continues execution, the hardware enables the disambiguation of these signatures against all processor accesses. If, by the time execution reaches another call to the function, no conflicts have been discovered, it is safe to assume that the implicit inputs and outputs have not changed.

It is possible that, during the execution of the function, an implicit output overwrites a location read by an implicit input. In this case, since an input has changed, memoization should fail. We call this case *Internal Corruption*, and it must be detected to guarantee correctness of the optimization. Fortunately, detecting this case is easy with SoftSig signatures.

In summary, MemoiSE works by recording the explicit inputs and outputs of a function call in the lookup table, collecting the addresses of the implicit inputs and outputs of the function using signatures and, after the function is executed, disambiguating these signatures against the addresses accessed by the code that follows. When we reach the next call to the function, we successfully memoize it if: (1) the explicit inputs match an entry in the lookup table, (2) during function execution, implicit outputs did not overwrite any implicit inputs, and (3) the implicit inputs and outputs have not been modified since the previous call as determined by signature disambiguation.

3.6.2 The MemoiSE Algorithm

MemoiSE is implemented by intercepting function calls using code inserted in functions. Figure 3.7 shows the application of MemoiSE to function `foo`. Part (a) shows the resulting layout of `foo`'s code. MemoiSE inserts three code fragments: Prologue, Setup, and Epilogue. Part (b) shows the statically-allocated lookup table for `foo`. An entry in the table records the values of the explicit inputs and the explicit outputs of a call to `foo`. Different entries correspond to different values of the explicit inputs. In a multithreaded program, each thread has its own private copy of the lookup table to avoid the need to synchronize on access to a shared table. In the following, we explain the Prologue, Setup, and Epilogue code fragments. Note that we have skipped some code optimizations in the figure to make the code more readable.

Prologue

The Prologue is shown in Figure 3.7(c). It determines whether the call can be memoized and, if so, it reads out the explicit outputs stored in the lookup table and immediately jumps to the function return. To understand the code, note that each entry in the lookup table is logically associated with an SR. This SR was used to collect the function's memory accesses when the function was called with the explicit inputs stored in the entry. Moreover, this SR has been disambiguated against all local and remote accesses since that function call was executed. Finally, the name of this SR was set to be the virtual address of the lookup table entry.

Based on this organization, the code in Figure 3.7(c) first sets register *R1* to point to the first entry in the lookup table, which is also the name of the associated SR. Then, the function's explicit inputs are compared to the values stored in the table entry. If they are the same, then this entry's explicit outputs can potentially be reused. However, we first need to check that the associated SR has not recorded a conflict since the function was last called with these explicit inputs. To perform the check, we first use the `sigstatv` instruction to read out the SR's Status Vector into register *R0*. If the bits in the Status Vector show both that there has been no conflict and that

this SR is not currently collecting addresses (if it is still collecting, it would mean that the function is recursive and, therefore, cannot be memoized), then memoization succeeds. In this case, the explicit outputs are read out from the table entry and control transfers to the function return. If, instead, memoization fails, the function needs to be executed. Also, if the explicit inputs did not match, we check subsequent table entries until a match is found or the table is exhausted.

Setup

If the function call is not memoized, the Setup code fragment initializes the necessary structures to record the effects of this call. The code is shown in Figure 3.7(d). It involves three operations, namely obtaining a new entry in the lookup table (or recycling the entry that has the same explicit inputs, if it already exists), saving the explicit inputs in the entry, and starting-up SRs to collect the addresses of the implicit inputs and outputs.

The instructions for the third operation are shown in Figure 3.7(d). We allocate two SRs — one for addresses read and one for addresses written. In the figure, the name of the SR for reads is the address of the table entry, and it is stored in *R1*; the name of the SR for writes is obtained by adding 1 to the entry's address, and it is stored in *R2*.

The next step is to skip the collection of (and the disambiguation against) the addresses of local accesses to memory-allocated variables that are neither implicit inputs nor implicit outputs. These are temporaries that are created on the stack for use during the call, or are explicit inputs or outputs passed on the stack. The stack locations where such variables are allocated is shown in Figure 3.7(f): we store explicit inputs or outputs between *SP* and *SP+C*, and temporaries between *MIN_STACK* and *SP*. In Figure 3.7(d), the `filtersig` instruction ensures that accesses to these variables are neither collected nor disambiguated against.

Next, we initiate disambiguation of remote writes against both SRs, and of local writes against the SR that collects reads. The latter operation will detect internal corruption (Section 3.6.1). Note that remote disambiguation is only necessary for multithreaded programs. Then, we start address collection for both SRs. Finally, we save *R1* and *R2* in the stack, since function `foo` may

write these registers, and we will need them later.

Epilogue

After `foo` executes, we can fill the entry in the lookup table. This process is performed by the Epilogue as shown in Figure 3.7(e). This code first restores *R1* and *R2* from the stack and ends collection for both SRs. It then obtains the Status Vectors of the SRs and checks that they have not recorded a conflict. If either one has recorded a conflict, then memoization is not possible; we discard the entry in the lookup table and deallocate the two SRs.

Otherwise, the two SRs are consolidated into one SR (whose name is in *R1* in the example) to save space. Moreover, the explicit outputs of the call are saved in the entry of the lookup table. Note that the remaining SR is currently under disambiguation against local and remote writes.

3.6.3 Optimizations for Lower Overhead

Since only some functions can benefit from memoization, a profiler should identify which functions are amenable to memoization and apply MemoiSE only to them, as per **G4**. We leverage an analytical model proposed by Ding and Li [29] to identify which functions are most likely to benefit from memoization.

Furthermore, searching a large lookup table usually adds significant overhead. Consequently, we use profiling to discover which functions mostly need a single-entry table. For these functions, we restructure the table while providing space for only a single entry, so that the table can be accessed with very low overhead.

3.7 Evaluation

3.7.1 Experimental Setup

To estimate the potential of MemoiSE, we implemented an analysis tool that uses Pin [55], a software framework for dynamic binary instrumentation. The output of Pin is connected to a simulator of a multiprocessor memory subsystem based on SESC [72]. The simulator models per-processor private L1 caches attached to a shared L2 cache. Some parameters of the architecture are shown in Table 4.2. With this setup, we can estimate MemoiSE's reduction in number of instructions executed and in execution time. The latter is obtained assuming that, when memory accesses do not stall the processor, the average IPC of non-memory instructions is 1. We model the overlap of instructions with L2 misses.

Reorder buffer	50 entries
Signature register file	16 signature registers, 1Kbit each
L1 cache: size, line, assoc, lat.	64 KB, 64B, 4, 1 cycle
L2 cache: size, line, assoc, lat.	2 MB, 64B, 8, 10 cycles
Max. outstanding L2 misses	16
Memory latency	500 cycles

Table 3.3: Parameters of the architecture simulated.

For our experiments, we run the applications shown in Table 3.4. They are Firefox, Gaim, Impress, SESC and Supertux. The first three are popular applications used on many personal computers. SESC is an architectural simulator [72] and Supertux is an open-source arcade game. Of these applications, Firefox, Impress, and Supertux are multithreaded, and run with 6, 6, and 2 threads, respectively. For each application, we trace an execution of over 400 million instructions.

We study MemoiSE in the context of the four environments of Table 3.5. By default, the results are normalized to *Baseline*.

App. (Num Threads)	Description	Section Analyzed
Firefox (6)	Popular web browser	Begins after initialization, while it loads the *ia-coma.cs.uiuc.edu* webpage
Gaim (1)	Open source instant messaging program	Begins once a client is running. It consists of opening a new message window, sending a message, and receiving a message
Impress (6)	OpenOffice presentation software	Begins with opening a sample presentation and continues while a user interacts with it
SESC (1)	Architectural simulator available from SourceForge.net	Performs a functional simulation of the *mcf* program using the default simulator configuration from SourceForge.net
Supertux (2)	"Jump'n run" arcade sidescroller game like Mario Brothers	Performed during game play. It begins when the penguin drops to the ground. It continues until the penguin dies and respawns

Table 3.4: Applications studied.

Environ.	Description
Baseline	No MemoiSE
Plain (P)	MemoiSE applied selectively to some functions using a cost-benefit analysis as in [29]. Lookup table size is limited to 10 entries
Optimized (O)	*P* optimized by reducing the lookup table size to a single entry for functions that get little benefit from larger tables. It has low overhead
Ideal (I)	*O* with unlimited number of SRs and no false positive conflicts. It approximates an ideal hardware behavior

Table 3.5: Environments analyzed.

3.7.2 Impact of MemoiSE

Figure 3.8 shows the dynamic instruction counts of the *P*, *O*, and *I* environments relative to *Baseline*. The figure shows data for each application and the average. Each bar is divided into two segments. The segment above zero (*Gains*) is the fraction of application instructions eliminated by memoizing function calls. The segment below zero (*Overhead*) is the additional instructions added by the memoization algorithm. The difference between *Gains* and *Overhead* is the savings achieved by MemoiSE, and is shown above each bar. A better optimization will have a taller *Gains* and a shorter *Overhead*.

From the figure, we see that, on average, *P* eliminates 13% of the application instructions.

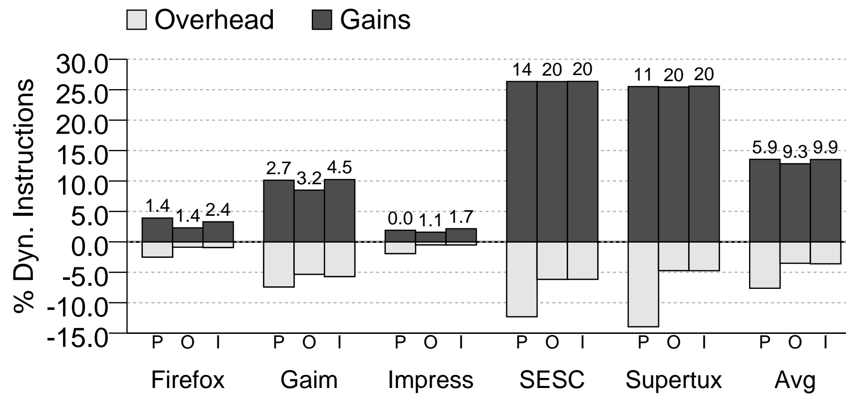


Figure 3.8: Dynamic instruction count relative to *Baseline*.

However, it adds overhead instructions, resulting in an average net instruction reduction of only 5.9%. For SESC and Supertux, the reduction in application instructions is especially significant, reaching over 25%. Moving now to the *O* bars, we see that keeping most of the tables to a single entry results in much lower overheads, while, in most cases, still eliminates a similar number of application instructions. The result is that the average net instruction reduction is lifted to 9.3% — and about 20% for SESC and Supertux.

Interestingly, having an unlimited number of SRs with no false positive conflicts (*I* bar) offers little additional advantage. The average net instruction reduction is slightly under 10%. Therefore, 16 SRs of 1Kbit each appear to be enough.

Figure 3.9 shows the execution times of the *P*, *O*, and *I* environments relative to *Baseline*. The figure shows that the execution time reductions closely follow the reductions in instruction count. On average, *O* offers a 9% reduction in execution time. Moreover, the reduction reaches 19% for SESC and Supertux. This is a significant reduction in execution time on challenging applications. In addition, the average reductions are nearly identical to those for the *I* environment. They are slightly better than for the *P* environment.

Our execution time analysis provides insights into the overheads of MemoiSE. Because MemoiSE adds a lookup table for each memoized function, the overheads of memoization can vary depending on the cache behavior of the accesses to the lookup tables. Fortunately, we observed

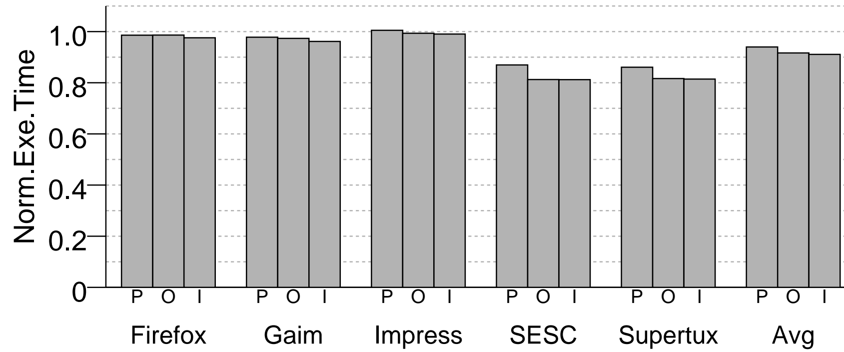


Figure 3.9: Execution time relative to *Baseline*.

that the accesses to the lookup tables rarely caused L2 misses due to the temporal locality of function calls.

Figure 3.10 shows the contention on the SRF. For each application and for the average, the figure shows the average number of accesses per cycle to the SRF for collection and disambiguation. Each bar is broken into four categories of accesses: *C-Rd* is the collection of read addresses, *C-Wr* is the collection of written addresses, *D-Loc* is local disambiguation, and *D-Rem* is remote disambiguation. There are also additional accesses due to allocating, deallocating, and manipulating SRs. However, they are not seen in the figure because they account for a very small fraction of the total accesses.

From the figure, we see that the average number of SRF accesses per cycle is about 0.11. This means that the SRF is only accessed roughly once in 10 cycles. This is a tolerable access frequency. In addition, for the multithreaded applications (Firefox, Impress, and Supertux), remote disambiguation causes most of the accesses. In all the applications, collection and local disambiguation are less significant, in part due to the filtering of many stack accesses.

3.7.3 Function Characterization

To further understand MemoiSE, we analyze in detail one function that is frequently memoized from each application. The functions are shown in Table 3.6. From left to right, the columns of

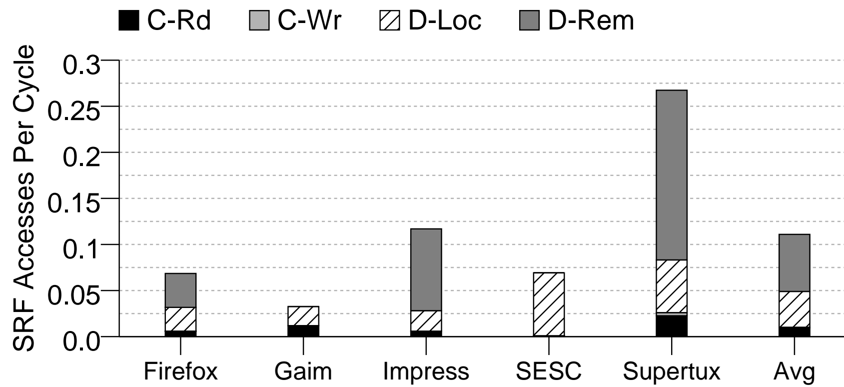


Figure 3.10: Mean number of SRF accesses per cycle of execution.

the table show: (1) function name; (2) application name; (3) explicit inputs; (4) type of the explicit output; (5) number of calls to the function in the execution analyzed by Pin; (6) average dynamic size of each call in instructions; (7) fraction of the total application instructions eliminated by memoization (*Gains* in Figure 3.8) that are contributed by this function; (8) fraction of the calls to the function that are memoized; (9) fraction of the failed memoizations of this function due to conflicts that are caused by false positives; and (10) average read and (11) write set size, respectively, of the function when it is successfully memoized. The read (or write) set size is the number of reads (or writes) to different locations.

`g_value_type_compatible` from Firefox is a function in the GLib GTK+ core library that checks whether two object types are compatible with each other. The check is done by accessing a type table and examining the inheritance tree. As shown in the *Mem* column, memoization is typically successful (75% of the times). This is because the data structures are only updated when a type is first registered. However, the large variation in the input values sometimes causes misses in the lookup table.

`pango_fc_font_get_glyph` from Gaim is a Pango GTK+ font library function that gets the glyph index of a given Unicode character for a font. The properties of a glyph within a font do not change once the font is loaded into memory and are requested frequently in Gaim as each

Function Name	App.	Explicit Inputs	Explicit Output	#Calls (Thous.)	Size (Inst.)	Weight (%)	Mem (%)	FP (%)	R Set
g_value_type_compatible	Firefox	GType src_type, GType dest_type	gboolean	17.8	212	19	75	0	7
pango_fc_font_get_glyph	Gaim	PangoFcFont* font, gunichar wc	guint	25.1	322	5	21	1	29
_dl_name_match_p	Impress	const char* __name, struct link_map* __map	int	41.1	80	31	95	0	8
OSSim::enoughMTMarks1	SESC	this, int pid, bool justMe	bool	33481.0	35	80	100	0	2
Sector::collision_static	Supertux	this, collision::Constraints* constraints, const Vector& movement, const Rect& dest, GameObject& object	void	7.7	5023	14	29	10	551

Table 3.6: Five functions that are frequently memoized from the different applications.

character is processed. Memoization is often successful (21% of the times, as shown in the *Mem* column), as characters are repeated. However, a larger lookup table would be desirable for a longer history of characters.

`_dl_name_match_p` from Impress is a function used internally in the GNU C library to test whether the given name matches any of the names of the given object. It is often used to resolve the name of a dynamically-loaded object such as a shared library object. Since the list of names for an object is updated only when it is first registered (e.g., when a shared library is first loaded), this function behaves much like a pure function at runtime. Consequently, as shown in the *Mem* column, it is memoized 95% of the times. Moreover, as shown in the *Weight* column, it contributes 31% of the application instruction reduction.

`OSSim::enoughMTMarks1` from SESC monitors several conditions to determine when the program should begin and end detailed timing simulation. While the condition checks are optimized, they are performed frequently — some kind of check is required after each instruction is simulated. The function is only 35 instructions, but it is called millions of times. Its memoization

is practically 100% successful, and it accounts for 80% of the application instruction reduction.

`Section::collision_static` from Supertux is part of the game logic that detects when collisions occur. It is called from three sites in the same function. The first two sites are in loops, with each iteration changing one of the input parameters. As a result, calls from these sites are not memoized. However, between the second and third sites, no change typically occurs to the parameters, allowing memoization. For these reasons, memoization is only 29% successful. Nonetheless, the function contributes 14% of the application instruction reduction because it has a large size (5023 instructions). However, a read set size of 551 addresses, as shown in the *R Set* column, leads to an increase in false positives. Indeed, Column *FP* shows that, of all failed memoizations due to conflicts, 10% are caused by false positives.

One additional observation is that all these functions have a zero write set when they are successfully memoized. We find that written locations are typically read by the same function, causing internal corruption and memoization failure.

3.8 Related Work

3.8.1 Signatures & Bloom Filters

SoftSig builds on the body of work that uses hardware signatures and Bloom filters for efficient disambiguation (e.g., [14, 15, 58, 60, 68, 77, 95]). Bulk [14], LogTM-SE [95], and SigTM [58] are closely related to SoftSig. Each system uses signatures for the explicit purpose of supporting TM or TLS. For the case of Bulk and LogTM-SE, these signatures are hardware registers that are used for the sole purpose of logging memory accesses and performing conflict detection. LogTM-SE can save and restore signatures to support virtualization, but in no other way are they manipulated by software.

SigTM employs a limited set of signature operations in software to implement Software Transactional Memory. Software can insert an address into a signature or do a membership test to sup-

port read and write barriers. Software can also do remote disambiguation to detect conflicts between transactions. SigTM, like LogTM-SE, has the means to save or restore a signature.

However, none of these schemes provide a comprehensive ISA to manipulate multiple signatures in a register file that enables the wide variety of tasks discussed in Section 3.3. Furthermore, SoftSig can be used even without support for speculative execution as in MemoiSE. Per **G1**, we opted not to save and restore SRs for the scenarios we considered. However, SoftSig does support save and restore, and can use them if SoftSig were employed in a TM system.

3.8.2 Memoization

Memoization has been studied at the granularity of instructions [52, 53, 79, 80] and coarse-grained regions [20, 21, 38, 76, 94]. Sodani *et al.* [80] empirically characterized the sources of instruction-level repetition and some characteristics of function-level behavior. They found that a large number of dynamic function calls are called with repeated arguments, and that most of these calls had either implicit inputs or side effects. This lead them to conclude that few functions could be memoized. However, with SoftSig, implicit inputs and side effects are easily coped with.

Connors *et al.* [20, 21] studied memoization of coarse-grained regions of code using a compiler (augmented with profiling information) to identify profitable regions. During execution, compiler-inserted instructions direct the hardware to record the explicit inputs and outputs for a region in a hardware table. Then, when the region is encountered again, the table is checked for a solution. If one exists, the outputs are written into registers directly by the hardware, and the region is skipped. To account for memory inputs, each table entry has a memory valid bit which is cleared anytime a memory input for that entry is potentially updated. The compiler is responsible for scheduling invalidate instructions in the code. Wu *et al.* [94] built on top of [20] by combining speculation and memoization to exploit more region-level reuse.

MemoiSE differs from all of these in that it only targets functions, as opposed to arbitrary regions of code. MemoiSE does have an advantage, in particular over [21], in that it can dynamically detect any memory access that invalidates an entry in the lookup table. In addition, the

memory accesses of a function do not need to be analyzed statically for correct memoization. MemoiSE incurs overhead for table lookups in software. Such lookups are done in hardware in [21]. If MemoiSE lookups were done in hardware, the overheads could be significantly reduced.

Ding and Li [29] proposed a compiler-directed memoization scheme implemented fully in software. The compiler identifies coarse-grained regions of code for reuse and then generates the necessary code to store the inputs in a lookup table and check the table on future calls. The compiler must prove that all inputs are invariant for a memoized region. Also, because there is no hardware support, the compiler must perform a cost-benefit analysis to decide when a region of code is worth memoizing. MemoiSE is similar to this approach in that the lookup table is a software structure and the compiler/profiler must decide which functions to transform using a similar cost analysis. MemoiSE, however, can more aggressively select functions since implicit inputs and outputs are checked dynamically.

3.9 Conclusion

This work proposed the SoftSig architecture to enable flexible use of hardware signatures in software for advanced code analysis, optimization, and debugging. SoftSig exposes a Signature Register File to the software through a rich ISA. The software has great flexibility to decide: (i) what stream of memory accesses to collect in each signature, (ii) what local or remote streams of memory accesses to disambiguate against each signature, and (iii) how to manipulate each signature. We also described the processor extensions needed for SoftSig.

In addition, this work proposed to use SoftSig to detect redundant function calls efficiently and eliminate them dynamically. We called our memoization algorithm MemoiSE. Our results showed that, for five multithreaded and sequential applications, MemoiSE reduced the number of dynamic instructions by 9.3% on average, thereby reducing the average execution time of the applications by 9%.

SoftSig can be used for many other optimizations. Several proposals for runtime-disambiguation based optimizations can be revisited, with potentially new applications or more general use [7, 50, 69, 83]. Also, aggressive speculative optimizations based on checkpointing [64] may benefit from SoftSig's ability to record information about a program's dependences. Of course, SoftSig can integrate into environments that already use signatures [14, 58, 95] to enhance the software's or the programmer's control over signature building and disambiguation.

Chapter 4

Alias Speculation Using Atomic Region Support

4.1 Introduction

Alias analysis is a compiler analysis pass that attempts to prove whether two different accesses in a program reference the same memory location. It is a crucial component in all modern compilers because many popular optimizations that need dataflow analysis, such as Loop Invariant Code Motion (LICM), Global Value Numbering (GVN), Dead Store Elimination (DSE), and Partial Redundancy Elimination (PRE) rely on alias analysis to tell them which data flows where.

In spite of this, proving the aliasing properties of a program is a notoriously difficult problem, especially in the presence of pointers. Multiple pieces of work have shown that alias analysis is hard even from a theoretical point of view [45, 71, 73, 16, 37]. Nonetheless, the importance of the problem has impelled many to work on it over the years[82, 4, 47, 93, 30, 33]. The difficulty has been that there is a trade-off between precision and efficiency, and no single solution has arisen that is both satisfactorily precise and efficient enough to gain general use [36]. The most precise algorithms that involve interprocedural analysis have high time and space complexity, which grow with the size of the program. This has prevented their use with large programs with many lines of code, because of memory space constraints. Moreover, precise alias analyses tend to be very complex to implement and are a well-known source of bugs. As a result, most popular compilers today such as GCC and LLVM [46] resort to simpler intraprocedural analyses.

Recently, there has been a flurry of activity in the hardware world to integrate support for transactional execution into processors, such as Intel's Transactional Synchronization Extensions [1], AMD's Advanced Synchronization Facility [18], Azul's Vega [19], IBM's Bluegene/Q,

and Sun's ROCK [17], and the Transmeta processors [26]. This was mostly in response to the trend of packing increasing numbers of cores inside a single processor [18, 19, 17]. To fully utilize these cores, programs necessarily have to become more and more parallel. Hence, many hardware vendors decided to invest on transactional hardware to make the programmer's job easier by enabling such techniques as transactional memory and speculative multithreading. Another trend in processors has been to perform on-the-fly binary optimizations on the uops of instructions before executing them, to improve on performance and energy efficiency [26]. Transactional hardware is useful in this setting because it allows speculative optimizations even when the processor has a limited view of the program due to being only able to see instructions that have executed so far. If the processor has made a wrong assumption, it can simply throw away the transaction and start anew.

Whatever the original motivation, this trend is of interest to alias analysis because of two reasons. First, transactions, or atomic regions, provide the ability to rollback to a time in the past, specifically a time prior to the point when a wrong assumption was made. Second, atomic regions naturally store the memory locations it has accessed in hardware, for the purposes of conflict detection and buffering speculative data in case of rollback. If this information can be exposed to the upper stacks, software could make assumptions about aliasing properties to perform speculative optimizations and do checks to validate them at runtime.

In this chapter, we apply these insights into building a speculative form of alias analysis that can be used by the compiler to do aggressive optimizations that were not previously possible. Unlike conventional alias analyses which need to prove aliasing properties, we only need to determine that the aliasing properties are true "most of the time". We can detect the cases where they are incorrect at runtime and just discard the executions in those cases.

This is not the first time that speculation has been adopted for alias analysis. Speculation has been used in the past in a purely software setting [65] or with the help of hardware to perform checks such as the Advanced Load Address Table(ALAT) [50, 51, 25]. However, the ability to rollback execution in hardware provides great advantages over these past schemes in multiple

ways.

First, it frees the compiler from having to generate recovery code to fix up the state in software, which is very complex and a source of bugs. Sometimes it is not even feasible to perform recovery in software because old values in memory have been overwritten already. What's more, the recovery code hampers subsequent optimization and code generation passes, potentially resulting in suboptimal code.

Second, the ability to rollback lets us drastically decrease the number of correctness checks that need to be inserted to enable speculative optimizations. Without the support of atomic regions, the compiler must place a check at each location in the code where the original memory access happened before speculatively moving an access. In the case of the ALAT, a chk.a or ld.c would have to be inserted for each original access [39]. This is to enable an immediate jump to recovery code for the cases when speculation fails. However, with atomic region support, the compiler only needs to place a check for each *address* that has been speculated upon in an atomic region. Because recovery is handled completely by hardware, it does not matter when the check happens, as long as it happens before the atomic region commits. One can imagine a scenario where a speculative code motion happens on a load in a loop that loads from a single address. With the ALAT, a check would have to be inserted at every loop iteration, whereas you can insert a single check at the end of the loop if you encase the loop in an atomic region. Delaying the check can increase the cost of recovery but this is a good trade-off when failure rate is low, as is often the case.

Other contributions in the chapter include: 1) proposing an efficient alias profiler that leverages the same hardware to determine the profitability of speculation, 2) describing how the novel alias analysis can be applied to an LICM optimization pass, and 3) evaluating the enhanced LICM optimization pass on the LLVM compiler [46], which resulted in a 14% speedup on average in SPEC FP2006 programs and 2% speedup on average in SPEC INT2006 programs over the baseline LLVM alias analysis.

The chapter is organized as follows: Section 4.2 presents some basic ideas, the application

to LICM, and required hardware extensions; Section 4.3 describes implementation; Section 4.4 presents examples of optimizations and experimental results; Section 4.5 discusses applying the analysis to optimizations other than LICM; Section 4.6 lists some relevant work.

4.2 Alias Speculation

4.2.1 Basic Idea

The main goal of this proposal is to leverage existing hardware support for atomic regions with some ISA extensions for the purposes of alias analysis. Pairs of references that do not alias "most of the time" are speculated upon as not aliasing and checks are inserted in the code to verify these assumptions at runtime. The speculation allows the compiler to perform code movements and optimizations that were previously impossible. In the rare case of a check failure, the atomic region is aborted and code without speculation is executed in place of the atomic region. We call this novel style of alias analysis Speculative Alias Analysis, or SAA.

The hardware support that is required of SAA consists of:

- **The Atomic Region (AR) primitive.** This primitive allows a compiler to demarcate a region of code that either gets committed atomically at the end or gets discarded if speculation fails. This requires the hardware to take a checkpoint of all architectural registers at the beginning of the atomic region and buffer all writes to memory so that they can be discarded in the event of failure. Hardware implementation of atomic regions typically do not impose any performance penalties on the application unless rollback and re-execution occurs.

- **The Address Check primitive.** This primitive allows a quick check if an address belongs to a group of addresses. More specifically, it checks whether an address is among the set of addresses written to in an atomic region (the write-set), or the set of addresses read from in an atomic region (the read-set). This can be done by accessing the speculative read and

speculative written bits that are available in most hardware implementations of atomic regions for the purposes of conflict detection. Alternatively the read-sets and write-sets can be encoded separately in signatures [14].

4.2.2 Compiler Transformations

While there are numerous compiler optimizations that can benefit from SAA, we focus on Loop Invariant Code Motion (LICM) for the purposes of demonstrating its usefulness.

The code motion in LICM refers to either *hoisting* or *sinking* [91]. A brief description and requirements of each is given below:

- **Hoisting.** This optimization involves the movement of an expression in the body of a loop to the preheader of the loop. In order for this to happen, the following must be true: **(Safety Rule)** the expression is guaranteed to be computed at least once during execution of the loop (or else it might cause an extraneous exception or side effect) and **(Loop Invariant Rule)** all the live-ins to the expression are loop invariant (or else the expression could not have been pre-computed).

- **Sinking.** This optimization involves the movement of an expression in the body of a loop to the exit block of the loop. In order for this to happen, the following must be true: **(Safety Rule)** expression is guaranteed to be computed at least once (same as for hoisting), **(Not Used Rule)** the value generated by the expression must not be used inside the loop (or else its evaluation cannot be delayed), and **(Not Stale Rule)** the expression should not have any output dependencies on live-out memory locations (or else the expression might store a stale value).

Load instructions and computation instructions are most often hoisted because the values generated by these instructions are usually needed inside the loop. Store instructions are most often sunk because they often store the outgoing results of a loop and there are no instructions dependent on the stored value.

Alias analysis is deeply involved in guaranteeing the Loop Invariant Rule, the Not Used Rule, and the Not Stale Rule of LICM. Here is how:

- **Loop Invariant Rule.** If a load instruction is to be proven loop invariant, it must not alias with any store instruction that might write to the same memory location. Moreover, the hoisting of many computation instructions are dependent on hoisting the load instructions that generate the live-ins to them.
- **Not Used Rule.** If a value written by a store instruction is to be proven as not being used inside the loop, it must not alias with any load instruction that might read from the same memory location.
- **Not Stale Rule.** If a value written by a store instruction is to be proven not to write a stale value, it must not alias with any store instruction that might write to the same memory location.

Many LICM optimizations are prevented because alias analysis cannot prove the above properties when using only static software alias analyses. Using SAA however, the compiler can go ahead with these optimizations by speculating that these properties will hold at runtime and simply insert checks to verify that assumption. In the case of load instructions and potentially exception generating computation instructions, it can even speculate on the **Safety Rule**. The hardware can monitor whether an exception actually occurs at runtime and abort the atomic region in that event. The **Safety Rule** cannot be relaxed for store instructions however, since a store instruction that should not have executed can silently store a wrong value to a memory location undetected.

An example of a code transformation of a loop using SAA and LICM is shown in Figure 4.1(a). The dotted line indicates where the atomic region(s) would be placed. The *check_load* and *check_store* are special instructions placed at the exit block of the loop to check that the **Loop Invariant Rule**, the **Not Used Rule**, and the **Not Stale Rule** have not been violated. A *check_store* instruction is placed for each load hoisted to verify the **Loop Invariant Rule** and check that nobody

stored to the same location. A *check_load* instruction and a *check_store* instruction are placed for each store sunk to verify the **Not Used Rule** and the **Not Stale Rule**. A more detailed description of these instructions are provided in Section 4.2.3. If any of the checks fail, the atomic region is aborted and the *safe version* of the loop is executed. The safe version is simply a replica of the loop that has been compiled without the help of SAA. Since there is no speculation done in the safe version, it is always safe to execute and no atomic regions are required. Once done with the safe version, execution jumps back to the main body of code after the loop.

Hardware resources for supporting atomic regions is limited and there is always a possibility that an atomic region might overflow the resources that buffer accessed data. The solution to this problem is well known, which is to perform *loop blocking* on the loop. After blocking, an atomic region is placed around the inner blocked loop. An example of this is shown in Figure 4.1(b). The *check_load* and *check_store* instructions must be placed at the exit block of the inner loop to check that the aliasing rules have been followed before committing each atomic region. If the checks fail, the atomic region is aborted and the safe version of the inner loop is executed. This safe version must recreate the hoisted values after it is done with re-execution of the inner loop because they are stale now. This involves nothing more than the recomputation of the hoisted code. After the recomputation, execution jumps back to next iteration of the outer loop.

In addition, we add a second atomic region around the hoisted code in the preheader to enforce the **Safety Rule**. If an exception is generated by any of the hoisted code, the atomic region is aborted and execution jumps to a safe version which is a non-speculative version of the entire loop. Once done with the safe version, execution jumps back to the main body of code after the outer loop has finished. We did not put an atomic region around the sunk code because all sunk code consisted of stores and the **Safety Rule** for these cannot be speculated upon.

4.2.3 ISA Extensions

The hardware has to be extended with the instruction set described in Table 4.1 in order to enable SAA. Note that the *store_check* and *load_check* instructions can only check that an access

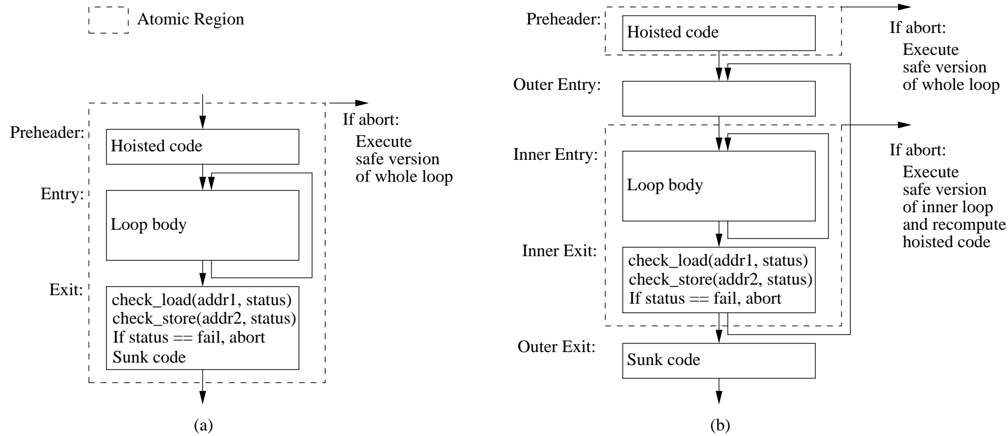


Figure 4.1: Using atomic region support for Loop Invariant Code Motion (a) without loop blocking and (b) with loop blocking.

has no aliases. It cannot check whether a pair of accesses *must alias* with each other. However, the ability to turn a *may alias* relationship into a *no alias* relationship is what is useful for the vast majority of compiler optimizations that require code motion or redundancy elimination, including LICM.

Instruction	Description
begin_atomic(PC)	Begins an atomic region, which stores a new register checkpoint and starts buffering memory accesses. PC holds an instruction pointer to the beginning of the safe version.
end_atomic	Ends an atomic region, which commits all buffered accesses.
abort_atomic	Aborts the atomic region and discards all buffered state, after which execution jumps to the safe version of the atomic region.
check_load(addr, status)	Checks if the address has been loaded in this atomic region. If so, the register designated by status is set to 1.
check_store(addr, status)	Checks if the address has been stored in this atomic region. If so, the register designated by status is set to 1.

Table 4.1: Extensions to the ISA to enable SAA.

4.2.4 Example of LICM using SAA

Figure 4.2 shows an example of a simple loop optimized by LICM using SAA. Assume a, b, c, and d may alias with $*p$ or $*q$ but they do not alias with each other. Without SAA, LICM cannot

```
for(i=0;i<100;i++) {
  a = b * c
  d = i * i
  *p = ...
  ... = *q
}
```

```
register int t1, t2;
begin_atomic(PC)
t1 = b * c;
for(i=0;i<100;i++) {
  t2 = i * i
  *p = ...
  ... = *q
}
a = t1
d = t2
check_store(&b, fail)
check_store(&c, fail)
check_store(&a, fail)
check_store(&d, fail)
check_load(&a, fail)
check_load(&d, fail)
if(fail) abort_atomic;
end_atomic
```

(a) (b)

Figure 4.2: Loop (a) before and (b) after performing LICM using SAA.

perform any optimizations in this situation. With SAA, LICM can use an atomic region to speculate that a, b, c, and d do not alias with $*p$ and perform 1) the hoist of $b * c$ and 2) the sink of the stores to a and d. The *check_stores* for b and c are executed to check the **Loop Invariant Rule** for $b * c$ against the store to $*p$. The *check_stores* for a and d are executed to check the **Not Stale Rule** for the stores $a = t1$ and $d = t2$ against the store to $*p$. The *check_loads* for a and d are executed to check the **Not Used Rule** for the stores agains the load to $*p$. If any of the checks fail, the atomic region is aborted. Note that the **Not Used Rule** for a and d are provably satisfied without needing any runtime checks.

4.3 Implementation

4.3.1 The Cost-Benefit Model

Until now, we have focused exclusively on *how* optimizations are enabled using SAA. But an equally important question is *when* are these optimizations beneficial? One could use heuris-

tics for this purpose [23], but a more precise method would be to actually profile the application and take real measurements. Fortunately, we can leverage the same hardware support we use for speculation to collect this information with high efficiency. We use the equation in Figure 4.3 to decide whether to speculate on a particular atomic region.

*(AR Abort Rate) * (# of Dyn. Insts in AR) + (SAA Inst. Overheads) < (# of Dyn. Insts Reduced by Optimizations)*

Figure 4.3: Cost-benefit model for an atomic region(AR).

The number of dynamic instructions in an atomic region and the number of dynamic instructions reduced by optimizations can be readily obtained by well-known methods such as path profiling [5] or some simple heuristic can be used.

The key is to estimate the abort rate of the atomic region. There are three reasons an atomic region can abort: buffer overflows, exceptions, and alias speculation failures. We can eliminate virtually all buffer overflows through techniques such as loop blocking. The number of aborts due to exceptions can be counted easily by running the program after instrumenting every potentially exception-throwing instruction that has been hoisted with an exception handler, which is available in most language runtimes (e.g. the try/catch block in C++). On an exception, the exception handler records that fact by incrementing a per-instruction counter. Alias speculation failures can be measured in a similar way by using the same *check_load* and *check_store* instructions used for speculation.

In fact, the code generated for the purposes of profiling looks almost identical to the one in Figure 4.1. The only difference is that checks are added pretending we have performed all speculative optimizations but the actual optimizations themselves are not performed, so that we won't have to abort if we encounter a check failure. Instead, the abort is recorded in a per-instruction counter for the checked instruction. Hence, at the end of the profiling run, we obtain a set of per-instruction counters that tells us how many times the speculative movement on the instruction would have caused aborts. Once we get that number, we can plug it into the cost-benefit model in Figure 4.3 and decide if that movement was beneficial.

The SAA profiler takes the place of conventional alias analyses in deciding when to perform code optimizations across potentially aliasing accesses. While we do not provide a quantitative comparison of the space and time complexity between the profiler and conventional alias analyses, a few facts become salient:

- The space complexity of the profiler is equal to the number of store and load instructions in the frequently executed loops in the program. All the profiler needs is locations to store the per-load and per-store failure counters for the accesses speculated upon. Typically, there are only a handful of important loops in a program so the space overhead is negligible in most cases. In the rare cases that it becomes a problem, an incremental approach can be taken where you profile a set of loops first and then move on to the next set of loops, in the order of importance.

- The time complexity of the profiler is equal to the time needed to run an program natively, after some instrumentation using SAAinstructions and try/catch blocks. The time needed to profile each loop in the program increases only linearly with the size of the program. The overhead caused by instrumentation is typically insignificant since only the preheader and exit blocks of a loop need instrumentation and not the loop body where most of the execution time is spent.

4.3.2 Compiler Support

We modified the LLVM compiler [46] release 2.8 to implement SAA. Considering how much effort it takes to correctly implement a complex alias analysis, the modifications were quite straightforward. Here is what is needed:

- Write a loop blocking pass that uses a profile of loop iteration counts for each loop.

- Write an instrumentation pass for the profiler. The profiler produces a profile in the end that contains the per-instruction contribution to the abort rate for each potentially moved

instruction.

- Write an instrumentation pass for the actual code that inserts the atomic regions and the necessary checks to validate the speculative code movements that were performed based on the per-instruction abort rate generated by the profiler and the cost-benefit model in Figure 4.3. This pass also needs to create a *safe version* for each atomic region.
- Modify the existing LICM pass to actually perform the speculative movements that have been decided upon.

4.4 Evaluation

4.4.1 Experimental Setup

Using the modified LLVM compiler [46], we produced three types of binaries: *BaselineAA*, *DSAA*, and *SpecAA*.

BaselineAA was built by running LICM using the default *Basic Alias Analysis* provided with the LLVM compiler, after running the standard -O3 set of optimizations. Basic Alias Analysis is what most applications compiled by LLVM use today. It is an aggressive yet scalable alias analysis that uses knowledge about heap, stack, and global memory allocations and structure field information among others.

DSAA was built by running LICM using *Data Structure Alias Analysis*, which was first proposed by Lattner et al. [47]. Data Structure Alias Analysis is the most advanced alias analysis that has been implemented on LLVM and is shown to be more precise [47] than either Steensgaard's [82] and Andersen's [4] alias analyses. It is a heap cloning algorithm that is flow-insensitive but context-sensitive and field-sensitive. It does unification to reduce memory usage but we found it is still not able to compile certain programs due to memory overflows. In fact, it failed to compile four benchmarks 400.perlbench, 403.gcc, 483.xalancbmk, and 445.gobmk with 4 GB of memory (the limit on our systems) and hence we are not able to show results using *DSAA* for

any of them. This lack of scalability for complex applications prevents *DSAA* from gaining widespread usage beyond the research setting.

SpecAA was built by running LICM using the SAA alias analysis. The *Basic Alias Analysis* was run before running SAA to filter out the easily provable alias relationships. Only the remaining *may aliases* after running *Basic Alias Analysis* were candidates for speculation. Also, loops with many iterations were blocked every 50 iterations. We found that a uniform 50 iterations for all benchmarks was large enough to amortize instrumentation overheads but small enough so that the atomic regions did not overflow speculation resources.

These binaries were run on a set of SPEC CPU2006 benchmarks. We ran both integer and floating point with the exception of 416.gamess and 481.wrf, which are both Fortran benchmarks that LLVM had trouble compiling correctly even before adding the SpecAApass. The performance of the applications was measured on a simulator built on top of the Pin binary instrumentation tool [55] that models a single issue processor extended with SAA instructions. We assume uniform latencies for all instructions and we simulate the speculation resources needed to enable atomic regions faithfully. Table 4.2 shows the parameters of the hardware.

Speculative L1 Cache: size, line, assoc.	64KB, 32B, 4
Speculative Victim Cache: size, line, assoc.	512B, 32B, Full

Table 4.2: Parameters of the architecture simulated.

The speculative L1 cache buffers all speculatively accessed data within an atomic region and when it overflows, the cache line is stored in the 16 entry fully associative victim cache. If the victim cache overflows, the atomic region is aborted and the safe version is executed. Speculatively written cache lines are either marked non-speculative when the atomic region commits or are invalidated if the atomic region aborts. Speculatively read cache lines are marked as non-speculative when the atomic region either commits or aborts. The cache lines have read and write speculative access bits which are accessible by the *check_load* and *check_store* instructions.

The alias profiling phase for *SpecAA* was also run on the same Pin-based simulator mentioned

above. We used the *train* input set provided in SPEC CPU2006 for the purposes of profiling and the the *ref* input set for the purposes of performance measurement. We found that the results of alias analysis profiling is insensitive to the input set and relatively small input sets can be used to generate the profiles, for the programs we tested.

4.4.2 Optimization Examples

Benchmark	Dynamic Instructions				Code Hoisted Out of the Loop	
	BaselineAA	SpecAA		Reduction (%)	BaselineAA	SpecAA
		Total	Overhead			
433.milc	448	362	21	19	Calculations of addresses for fields $real$ and $imag$ in $b \rightarrow e[0][0]$, $b \rightarrow e[1][0]$, $b \rightarrow e[2][0]$.	All hoists in BaselineAA. Loads of fields $real$ and $imag$ in $b \rightarrow e[0][0]$, $b \rightarrow e[1][0]$, $b \rightarrow e[2][0]$.
437.leslie3d	1863	540	16	71	Loads of $IADD$, $IBDD$. Computations for $IADD + IBDD$, $IADD - IBDD$.	All hoists in BaselineAA. Loads of dimensions for QAV and Q. Partial calculation of addresses for $QAV(I, J, K, L)$, $Q(IBD, J, K, L, N)$, $Q(II, J, K, L, N)$, $Q(ICD, J, K, L, N)$.

Table 4.3: Analysis of examples shown in Figure 4.4 and Figure 4.5.

In this section we show two examples in real code where *SpecAA* was able to produce more efficient code compared to either *BaselineAA* or *DSAA*. Figure 4.4 shows one loop in 433.milc, a C program, and Figure 4.5 shows another in 437.leslie3d, a Fortran program, before and after applying *SpecAA*. *Check_load* and *check_store* instructions were omitted for brevity.

For both loops, you can see that applying *SpecAA* resulted in a significant reduction in the number of instructions, 19% for 433.milc and 71% for 437.leslie3d.

For 433.milc in Figure 4.4, *SpecAA* unrolled the loop three times to produce a singly nested loop before an atomic region was placed around the entire loop. *BaselineAA* also unrolled the

```
for(i=0;i<3;i++) {
   for(j=0;j<3;j++) {
      ...
      br=b->e[j][0].real;  bi=b->e[j][0].imag;
      ...
      c->e[i][j].real=cr;  c->e[i][j].imag=ci;
   }
}
```

(a)

```
begin_atomic(PC)
real0 = b->e[0][0].real;  imag0 = b->e[0][0].imag;
real1 = b->e[1][0].real;  imag1 = b->e[1][0].imag;
real2 = b->e[2][0].real;  imag2 = b->e[2][0].imag;
for(i=0;i<3;i++) {
   ...
   br0=real0;  bi0=imag0;
   br1=real1;  bi1=imag1;
   br2=real2;  bi1=imag2;
   ...
   c->e[i][0].real=cr0;  c->e[i][0].imag=ci0;
   c->e[i][1].real=cr1;  c->e[i][1].imag=ci1;
   c->e[i][2].real=cr2;  c->e[i][2].imag=ci2;
}
end_atomic
```

(b)

Figure 4.4: Example loop in function mult_su3_na(...) of 433.milc (a) before and (b) after applying *SpecAA*.

loop three times. After doing so, *BaselineAA* was able to hoist out the address computations for the fields in $b{\rightarrow}e[0][0]$, $b{\rightarrow}e[1][0]$, and $b{\rightarrow}e[2][0]$, but not the loads of the fields themselves. The address computations could be hoisted because the dimensions of b were statically defined but the loads could not be moved because neither *Baseline* nor *DSAA* could prove that they do not alias with the updates to the fields in $c{\rightarrow}e[i][0]$. *SpecAA* was able to determine that b and c are always distinct locations through profiling and speculatively hoisted the loads.

```
DO I = I1, I2
   II   =   I + IADD
   IBD = II - IBDD
   ICD = II + IBDD
   QAV(I,J,K,L) = R6I * (2.0D0*Q(IBD,J,K,L,N) +
>                          5.0D0*Q( II,J,K,L,N) -
>                                Q(ICD,J,K,L,N))
END DO
```

(a)

```
t1 = IADD - IBDD
t2 = IADD + IBDD
begin_atomic(PC1)
offset1 = QAV_d1*J + QAV_d2*K + QAV_d3*L
offset2 = Q_d1*J + Q_d2*K + Q_d3*L + Q_d4*N
end_atomic
DO b = I1, I2, 50
begin_atomic(PC2)
DO I = b, MIN(b + 50, I2)
   II   = I + IADD
   IBD = I + t1
   ICD = I + t2
   QAV(I+offset1) = R6I * (2.0D0*Q(IBD+offset2) +
>                          5.0D0*Q( II+offset2) -
>                                Q(ICD+offset2))
END DO
end_atomic
END DO
```

(b)

Figure 4.5: Example loop in subroutine EXTRAPI() of 437.leslie3d (a) before and (b) after applying *SpecAA*.

For 437.leslie3d in Figure 4.5, *SpecAA* performed loop blocking every 50 iterations and one

atomic region was placed around the blocked inner loop and another around part of the hoisted code in the preheader of the outer loop. The latter atomic region was placed to speculate on the **Safety Rule**. Note that the hoisted loads and computations for $IADD - IBDD$ and $IADD + IBDD$, which are used to compute values IBD and ICD, needed not to be placed inside the atomic region because $IADD$ and $IBDD$ are global variables and loading from them provably cannot cause an exception. *BaselineAA* did not need to block the loop because it does not use atomic regions.

Both QAV and Q in the code are dynamically allocated arrays, and hence the dimensions have to be loaded from memory in order to calculate the address of each element. QAV_d1, QAV_d2, QAV_d3, QAV_d4 and Q_d1, Q_d2, Q_d3 refer to the dimensions of matrices QAV and Q respectively. We used C-style pointer arithmetic to illustrate how the matrix element addresses were computed since direct address calculations cannot be shown in Fortran.

BaselineAA could easily hoist the loads and computation for $IADD - IBDD$ and $IADD + IBDD$ because $IADD$ and $IBDD$ are locations in global memory whereas the updated value $QAV(I, J, K, L)$ is a dynamically allocated location in the heap. However, neither *BaselineAA* nor *DSAA* was able to differentiate between the updated location $QAV(I, J, K, L)$ and the locations that store the dimensions for QAV and Q and so could not hoist anything more. Using speculation, *SpecAA* could hoist the loads of the dimensions with the parts of the address calculations for the elements in Q and QAV that were loop invariant.

Table 4.3 presents an analysis of optimizations performed on these two loops. Columns 2-5 show what effect optimizations had on the average dynamic instruction count of each atomic region. Column 2 refers to the dynamic instruction count when compiled using the *BaselineAA* configuration (*BaselineAA* does not actually contain atomic regions but we count the corresponding region of code). Column 3 refers to the dynamic instruction count when compiled using the *SpecAA* configuration. Column 4 shows how many instructions in column 3 were for the purposes of enabling atomic regions, such as *check_load* and *check_store* instructions, *begin_atomic* and *end_atomic* instructions, and instructions needed to block the loops. Column 5 shows the per-

centage of instruction reduction for *SpecAA* when compared to *BaselineAA*. Columns 6-7 show the actual code movements that occurred for the *BaselineAA* and *SpecAA* binaries. The results for *DSAA* were exactly the same as *BaselineAA*.

4.4.3 Experimental Results

In this section, we measure the impact of the optimizations we have described on benchmarks, using the setup described in Section 4.4.1.

Alias Analysis Results

Figure 4.6 measures the quality of alias analysis responses generated by each alias analysis pass, when paired with the LICM pass. The results shown are the percentage of queries that returned a *may alias*, *no alias*, or *must alias* response among the total set of queries thrown by the LICM client pass. *B*, *D*, and *S* refer to the *BaselineAA*, *DSAA*, and *SpecAA* configurations respectively.

We are not able to show bars for 400.perlbench, 403.gcc, 483.xalancbmk, and 445.gobmk for *DSAA* due to reasons mentioned in Section 4.4.1. For these two benchmarks, we use the results of *BaselineAA* when we calculate the averages. This methodology was used throughout the evaluation.

Alias analyses return a *may alias* response when they can neither prove that a pair of references must alias nor not alias, and hence the fraction of *may alias* responses is a measure of their precision. Note that *SpecAA* consistently outperforms both *BaselineAA* and *DSAA* in all benchmarks, leading to a drastic reduction in the fraction of *may alias* responses. All of the reduction in *may alias* responses are translated into *no alias* responses, thereby aiding in the code movement of LICM. Note that even *SpecAA* is not able to get rid of all *may alias* responses because some pairs of references actually do *may alias* during execution. That is, sometimes they reference the same location and at other times they do not. In addition, SpecAA is not able to check *must alias* relationships as noted in Section 4.2.3 and hence some of the remaining *may aliases*

might be the result of SpecAA returning *may alias* responses to pairs of references that never alias.

Performance

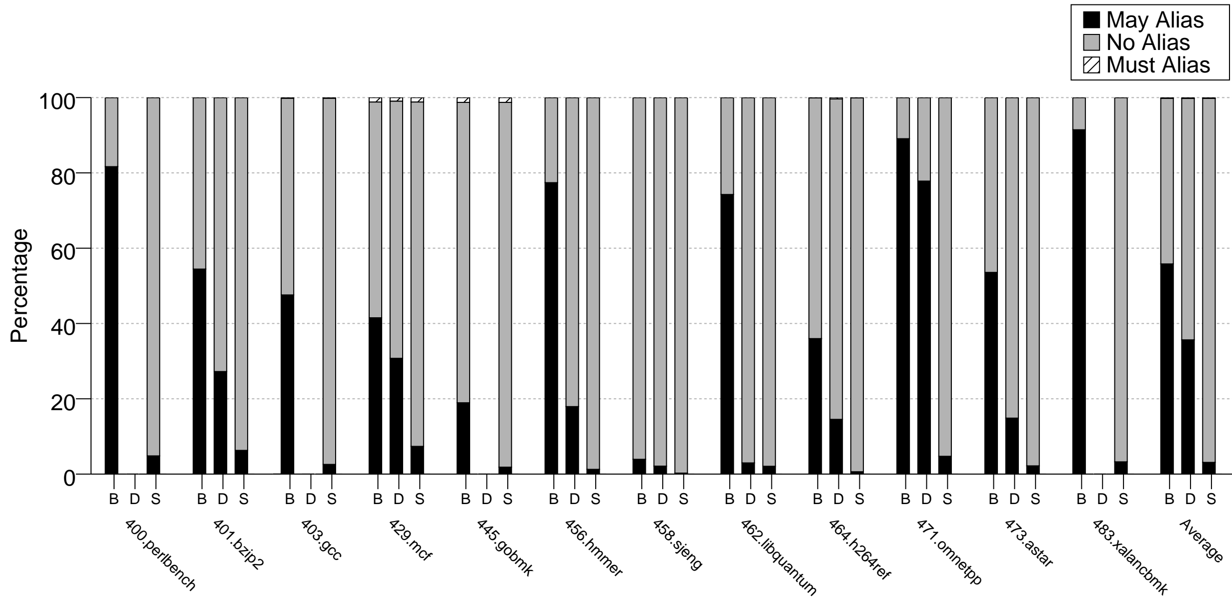
Figure 4.7 shows the impact the speculative alias analysis had on application performance. Performance was measured using instruction count and the results are normalized to *BaselineAA*. The first thing to note is that better alias analysis does not necessarily translate to better performance. This is because even a single alias inside a loop can prevent code from being moved outside the loop. And even if the analysis can prove that all references do not alias with each other, there simply might not be loop invariant values in that loop.

However, *SpecAA* does give significant speedups in some benchmarks, beyond what can be provided using *DSAA*. On average, *SpecAA* was able to speedup SPEC FP2006 benchmarks by 14% and SPEC INT2006 benchmarks by 2% over *BaselineAA* on average, compared to 5% for SPEC FP2006 and 2% for SPEC INT2006 when using *DSAA*.

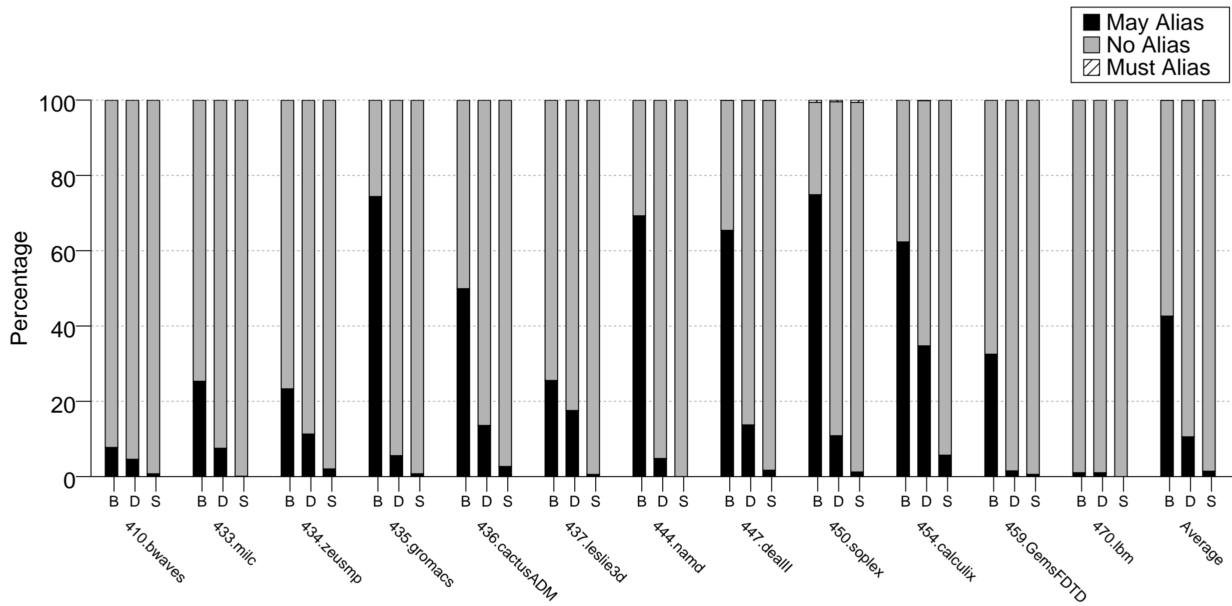
Figures 4.8 show reduction in the dynamic number of instructions when running the entire program categorized by non-memory, load, and store instructions. *B*, *D*, and *S* again refer to the *BaselineAA*, *DSAA*, and *SpecAA* configurations. The most marked reduction can be seen in the number of load instructions. The hoisting of load instructions that read loop invariant values from memory was responsible for the lion's share of performance improvement. You can also see some reduction in non-memory instructions, especially in 437.leslie3d and this is due to the movement of the calculation of addresses for matrix element accesses, such as was shown in Section 4.4.2.

The slight increase in non-memory instructions in some benchmarks is due to operations on memory being converted to operations on registers which hold the value of loads that have been hoisted out of the loop.

Store instructions also saw some reduction in 437.leslie3d and 433.milc. This was due to two reasons: 1) the sinking of stores to the exit of the loop and 2) the reduction in register pres-

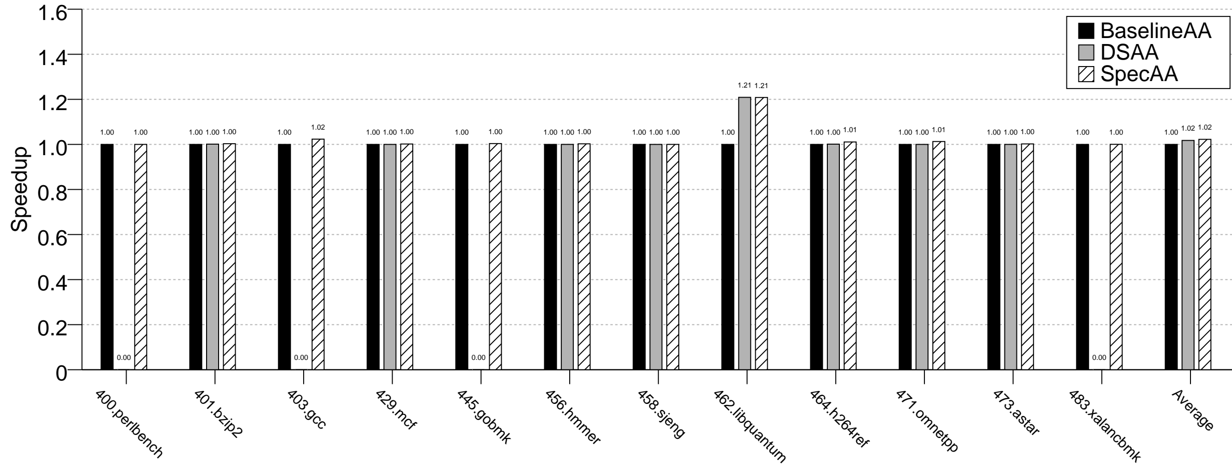


(a)

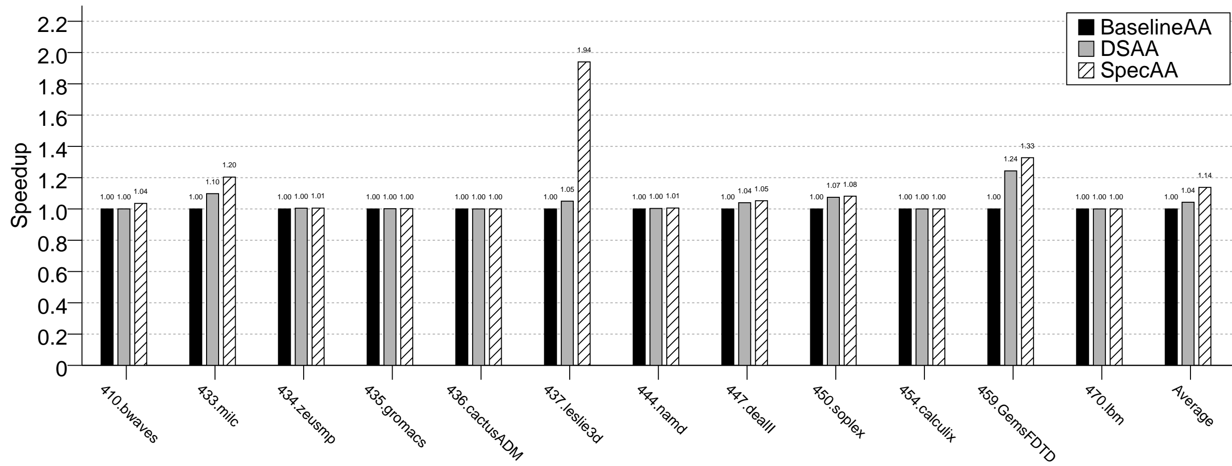


(b)

Figure 4.6: Fraction of alias queries performed by LICM that returned a *may alias*, *no alias*, or *must alias* response for (a) SPEC INT2006 and (b) SPEC FP2006. B, D, and S stand for *BaselineAA*, *DSAA*, and *SpecAA*.



(a)



(b)

Figure 4.7: Speedups normalized to baseline for (a) SPEC INT2006 and (b) SPEC FP2006.

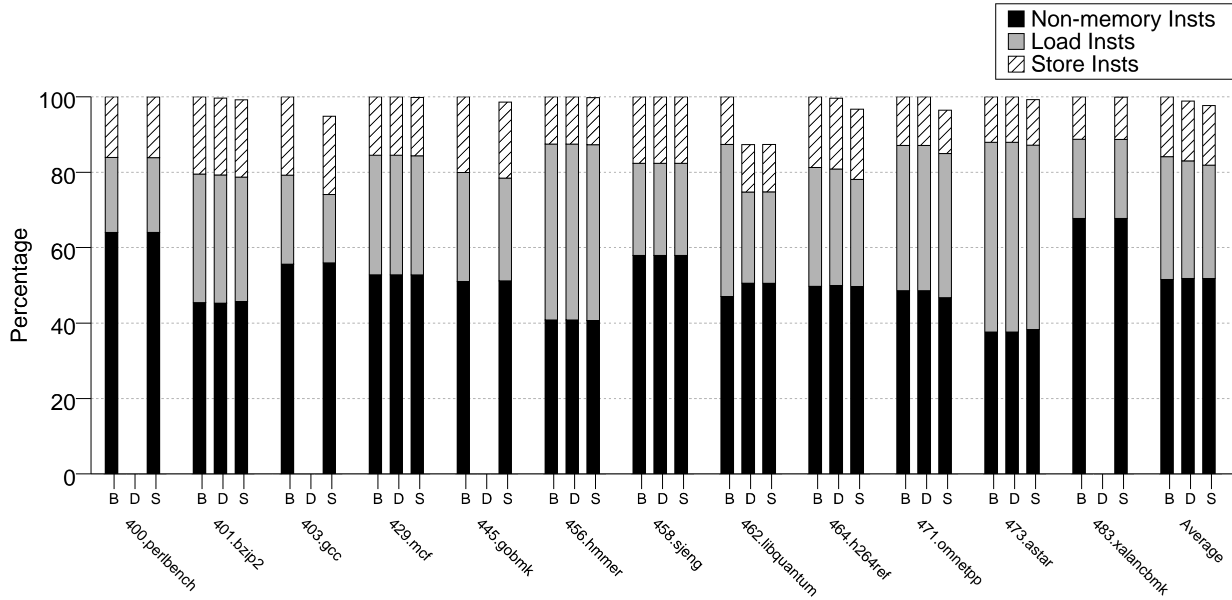
sure due to having computation moved out of the loop, leading to less spills. However, register spills can also be increased because of LICM. If you hoist loads of values that are used inside the loop, that ends up increasing the live ranges of the registers that contain the values of those loads, which can increase register pressure and lead to more spills. This was the case in 459.GemsFDTD. But the reduction in load instructions more than made up for the difference.

Limit Study

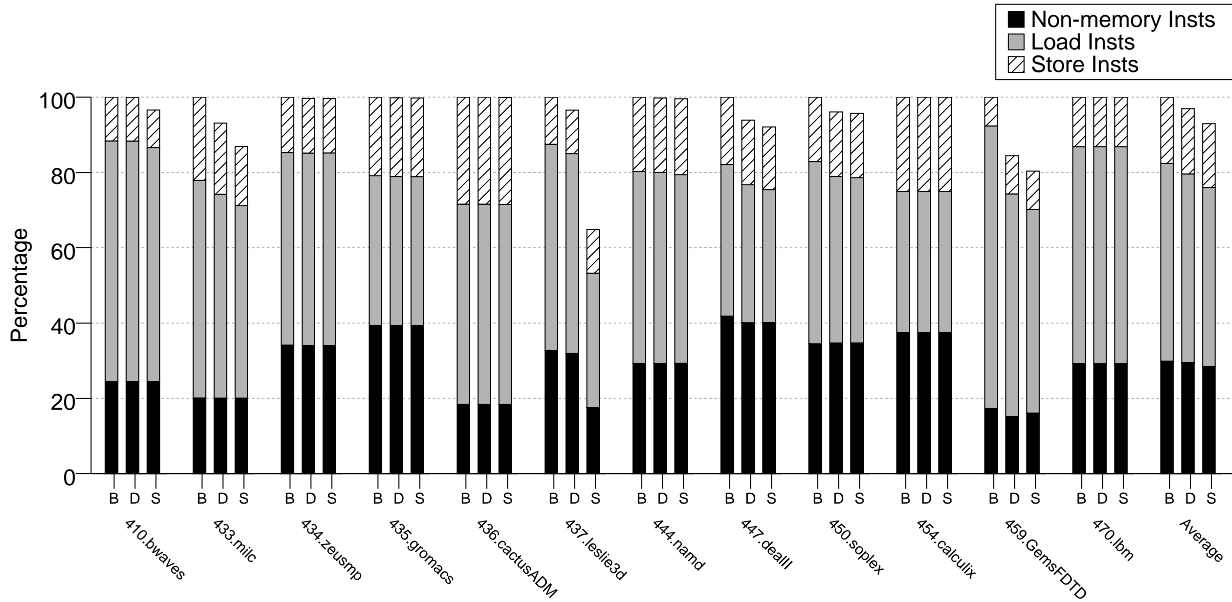
One surprising thing about the set of benchmarks that we studied was that we did not observe any atomic region aborts even in code optimized with *SpecAA*. Initially we thought we were being too conservative and assigning too much cost to the cost-benefit function given in Figure 4.3, to the point where we were speculating only on completely safe optimizations. The logic was that if we could decrease the abort cost, perhaps by decreasing the size of atomic regions or by decreasing the abort rate by using an abort predictor, we could potentially enable more optimizations. So we did a limit study on how much we would gain in terms of performance if we were able to remove all cost of aborts. The results of this experiment is shown in Figure 4.9. The figure shows speedups normalized to *BaselineAA*. The binary for the new configuration *SpecAANoCost* was built by performing all possible speculative optimizations regardless of cost. And when running *SpecAANoCost* on our simulator, we assumed that all aborts could be avoided. The results show that *SpecAANoCost* has little advantage over the original *SpecAA*. In other words, we've already performed all optimizations that are accessible through speculation on aliases, regardless of the cost-benefit model.

Characterization

In this section, we characterize the behavior of atomic regions instrumented by *SpecAA*, during runtime of the applications. Table 4.4 characterizes dynamic atomic regions in terms of instruction count. Table 4.5 characterizes dynamic atomic regions in terms of the hardware speculation and conflict detection resources needed per atomic region on average and at maximum.



(a)



(b)

Figure 4.8: Dynamic instructions normalized to baseline broken down by category for (a) SPEC INT2006 and (b) SPEC FP2006. B, D, and S stand for *BaselineAA*, *DSAA*, and *SpecAA*.

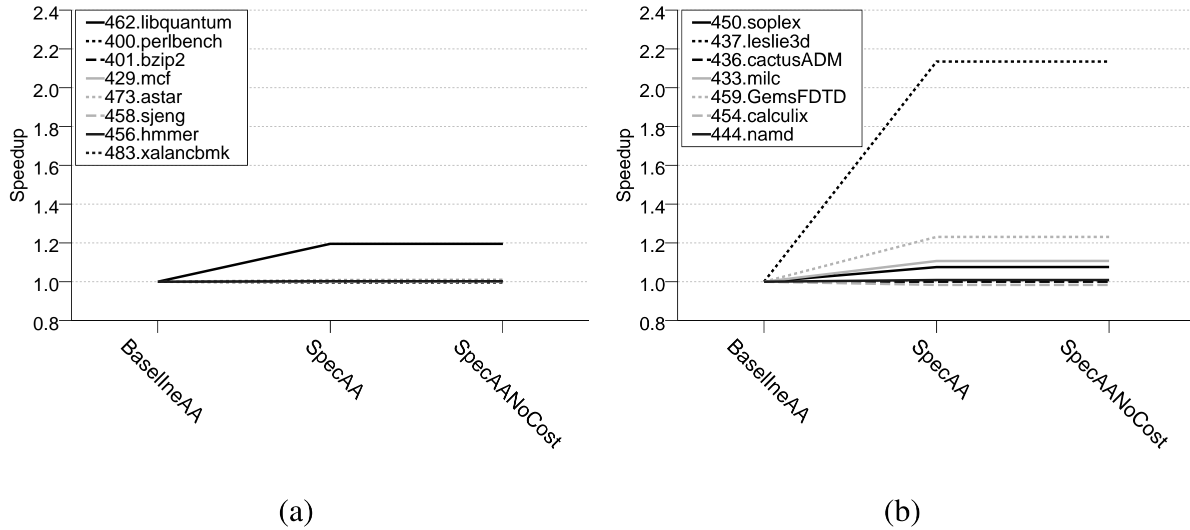


Figure 4.9: Speedups when varying the speculation threshold for (a) SPEC INT2006 and (b) SPEC FP2006.

Here is an explanation of each column in Table 4.4. Colummn 2 shows what percentage of dynamic instructions were covered with atomic regions. Columns 3-5 show the dynamic instruction count of atomic regions on average for each configuration after performing LICM. As mentioned before, *BaselineAA* and *DSAA* don't actually contain atomic regions but we counted the number of instructions in the corresponding region of code. Column 6 shows how many instructions in column 5 were due to overhead. Column 7 shows the percentage of instruction reduction for *SpecAA* when compared to *BaselineAA* on average per atomic region.

We can see in the *BaselineAA* numbers that we start with sizable atomic regions, using a loop blocking factor of 50 as was described in Section 4.4.1. This is desirable to amortize the overhead of SAA instructions but certain applications such as 400.perlbench did not have enough loop iterations to start with which resulted in small atomic regions, and it ended up having meager instruction reduction in the end. In some benchmarks like 464.h264ref, you see a significant reduction in the number of instructions in the region of code optimized using atomic regions but it didn't have much impact on performance because of low coverage. Both high coverage and instruction reduction rate are needed to achieve good speedups as in the case 462.libquantum and

437.leslie3d.

Here is an explanation of each column in Table 4.5. Each pair of columns indicates how much hardware resources would be needed to encode addresses or buffer speculative data on average and at maximum. Columns 2-3 show how many words were read during execution of the atomic region Columns 4-5 show the same information for words written. Columns 6-7 show the occupancy rate of the L1 cache while buffering all the reads and writes in the atomic region. Columns 8-9 show the occupancy rate of the victim cache.

You can see that some benchmarks only use a small portion of the 64KB L1 cache and do not even use the victim cache at all. However for benchmarks with larger atomic regions, you can see that both the L1 and victim caches are needed. Benchmarks such as 410.bwaves and 459.GemsFDTD start overflowing the L1 cache long before its full capacity is reached because accesses are not evenly distributed among sets in the 4-way associative cache. In these cases, having a fully associative victim cache to store overflowed lines really helps, even with just 16 entries.

We could downsize the cache space needed to buffer speculative data if the read set of an atomic region is encoded in a structure separate from the cache such as a signature [14]. In this case, we'll only have to buffer the speculatively written dirty lines of an atomic region (i.e. the write set), for the purposes of rollback.

4.5 Discussion

In this chapter, we showed how SAA could be used for a single widely used compiler optimization: LICM. However, SAA can also be used for optimizations such as Partial Redundancy Elimination (PRE), Global Value Numbering (GVN), Common Subexpression Elimination (CSE), and Dead Store Elimination (DSE). All of these optimizations involve removing some form of redundant computation, but in the process, they require code motion in the same way LICM does.

Benchmark	Coverage(%)	Dynamic Instructions per Atomic Region				
		BaselineAA	DSAA	SpecAA		Reduc. (%)
				Total	Overhead	
400.perlbench	0	488	0	478	2	2
401.bzip2	1	509	509	410	3	19
403.gcc	14	314	0	262	3	16
429.mcf	1	459	459	392	3	14
445.gobmk	3	688	0	616	3	10
456.hmmer	1	4296	4273	3472	4	19
462.libquantum	95	767	628	628	4	18
464.h264ref	3	476	451	322	7	32
471.omnetpp	31	1875	1875	1792	4	4
473.astar	1	313	313	262	4	16
483.xalancbmk	0	635	0	585	3	7
410.bwaves	68	3180	3179	3019	12	5
433.milc	60	8063	6829	5745	11	28
434.zeusmp	17	4552	4394	4378	6	3
435.gromacs	2	1385	1319	1251	4	9
436.cactusADM	0	1573	1573	661	3	57
437.leslie3d	79	2712	2603	1052	21	61
447.dealII	22	1397	1153	1088	6	22
450.soplex	41	757	628	617	5	18
454.calculix	0	547	544	488	3	10
459.GemsFDTD	74	14533	10698	9705	112	33

Table 4.4: Average instruction statistics per atomic region for each benchmark.

Benchmark	ReadSet (# words)		WriteSet (# words)		L1 Occ.(%)		Victim Occ.(%)	
	Avg	Max	Avg	Max	Avg	Max	Avg	Max
400.perlbench	6	6	3	3	0	0	0	0
401.bzip2	17	25	10	18	0	0	0	0
403.gcc	43	280	44	229	1	5	0	0
429.mcf	121	237	2	2	2	4	0	0
445.gobmk	11	33	20	51	0	1	0	0
456.hmmer	43	78	77	104	0	1	0	0
462.libquantum	104	107	24	101	1	1	0	0
464.h264ref	32	170	20	82	0	7	0	0
471.omnetpp	29	43	6	13	0	1	0	0
473.astar	44	64	19	51	1	2	0	0
483.xalancbmk	6	18	2	6	0	0	0	0
410.bwaves	1288	2408	524	1035	11	19	6	12
433.milc	98	225	48	52	1	1	0	0
434.zeusmp	680	1565	158	462	5	11	0	0
435.gromacs	157	508	113	301	1	5	0	0
436.cactusADM	25	30	4	4	0	0	0	0
437.leslie3d	247	678	119	643	2	9	0	0
447.dealII	43	106	28	75	0	1	0	0
450.soplex	140	253	77	101	1	2	0	0
454.calculix	51	108	26	101	0	2	0	0
459.GemsFDTD	1074	1473	319	489	15	34	16	87

Table 4.5: Hardware resources required per atomic region for each benchmark.

For example, if you are to perform CSE, you have to be able to move the redundant expression to the site of the original expression in order to perform the elimination. If there are any aliasing accesses in the path of code motion, the optimization is denied. SAA can be used here to speculate across those accesses, turning *may aliases* into *no aliases*.

However, compared to LICM, the distances that the code moves in other optimizations are relatively short, in terms of dynamic instructions. This means that if we were to cover each path of code motion in an atomic region, we would end up having small atomic regions which would increase overheads. What is more, differing paths of code motion might overlap with each other making it hard to set atomic region boundaries. In this sense, we had an easier time with LICM since all code motion was uniform: across the entire loop.

Hence, to enable these other optimizations, it is important that we decouple the Atomic Region primitive from the Address Check primitive (see Section 4.2.1). That is, we must allow address checks against a subset of addresses that the atomic region reads and writes. And, we need potentially multiple subsets to support various paths of code motion inside a single large atomic region. You don't necessarily need as many subsets as the number of code motion paths since the compiler might be able to merge certain subsets. For example, even if the code motion involved speculation on only a subset of reads or writes in an atomic region, if the compiler can determine either by proof or profiling that the code does not alias with the rest of the atomic region, you can just do address checks against the entire read set or write set. However, this is not always the case.

Architectures that leverage signatures to perform conflict detection such as Bulk [86], SoftSig [87], and SigTM [58] naturally decouple the Atomic Region primitive from the Address Check primitive. Multiple signatures can be assigned to a single atomic region to encode various subsets of addresses, decoupled from the mechanism to store a checkpoint. While we did not evaluate other compiler optimizations, we believe that this more advanced hardware support can result in a better alias analysis and more efficient code.

4.6 Related Work

4.6.1 Optimizations using Atomic Regions

Work on leveraging hardware atomicity for speculative code optimizations was initially done for the purposes of doing control speculation to enable optimizations over a hot trace of code [64, 63, 26, 67, 11]. When these traces of code were generated to enable on-the-fly optimization of uops translated from machine instructions [63, 26, 11, 10], the atomic regions were also used to support precise exceptions at the machine level. In addition, atomic regions were a good way of hiding code movements, which might violate the machine memory model, from other processors. Recently, it was shown that a Java runtime compiler could enforce sequential consistency, a much stricter memory model compared to the Java Memory Model, all the while improving performance by using atomic regions [3]. Continued interest in atomic regions have also led to work that proposes ways to deal with the limitation on hardware speculation resources either through short rollbacks [11] or smart monitoring of the resources [10].

4.6.2 Alias Analysis

There is a vast body of work concerning static alias analysis [36]. Work has been done and is still on-going to prove that alias analysis is in fact a hard problem [45, 71, 73, 16, 37], but researchers have tackled the problem nonetheless, trying to achieve precision and efficiency at the same time [82, 4, 47, 93, 30, 33].

A newer direction of work was to perform alias analysis speculatively, relying on runtime checks to verify them in similar ways as SAA. These checks can either be done purely in software [65] or with the assistance of a hardware table that monitors accesses to memory locations such as the Advanced Load Address Table(ALAT) [50, 51, 25]. The ALAT is used for hoisting load instructions across potentially aliasing stores. The use of monitoring hardware can significantly decrease the frequency of checks in the code, but it does not alleviate the need of perform-

ing speculation failure recovery in software.

This recovery code can be very complex because it tries to "undo" work that has been done and sometimes is not even feasible because old values have been overwritten already. Also, the check and recovery code must be inlined at every location where the original load happened. For example, to hoist a load speculatively in LICM, the compiler needs to insert a load checkling instruction inside the loop body where the load instruction used to be. Delaying the check as in SAA increases the state that needs to be compensated for by the recovery code and quickly becomes infeasible. Hence, load hoisting through the ALAT is often successful in reducing memory latency but not the number of instructions. Moreover, the inlined check and recovery code becomes an obstacle in future passes of code optimization and generation. All of the above limits the scope and aggressiveness of speculation. SAA is free of such problems since the hardware is responsible for the rollback and recovery of the atomic region.

Another interesting approach has been to have the hardware monitor alias speculation violations and silently patch up the state when it happens, unbeknownst to software and even without having to rollback execution, such as CRegs [24] and the Store-Load Address Table(SLAT) [69]. While it is an attractive idea, the complexity of patching up the state is analogous to the job of generating recovery code in the compiler and doing this in hardware clearly has its limits. And hence, the optimization that they support is limited to speculative register promotion. SAA lets hardware and software do what they are capable of doing best: hardware is good at taking checkpoints and restoring them, and software is good at recreating state from a known checkpoint.

4.7 Conclusions

This chapter presents a new way of performing speculative alias analysis through the use of atomic regions, which is gaining increasing support from industry for the purposes of parallelization. In using atomic regions, we leverage what hardware does much better than software: checkpoint and recovery. This precludes the need for the compiler to generate recovery code which can be

error-prone, sometimes infeasible, and overhead-imposing. Also, the use of atomic regions allow us to delay instructions that check the speculated accesses to the very end of the atomic regions, which greatly decreases the overhead. We also showed how the same hardware that is needed to support atomic regions can be used to monitor aliasing, if exposed appropriately to software. Lastly, we showed how a profiling pass to determine the merits of speculation could be accelerated greatly by the same hardware support.

We implemented our ideas on LLVM, a commercial grade compiler, and tested the precision of the alias analysis using the LICM optimization pass. This resulted in a 14% speedup for SPEC FP2006 benchmarks and a 2% speedup for SPEC INT2006 benchmarks on average over the baseline LLVM alias analysis. We also showed detailed characterization of atomic regions formed by SAA and examples of real code where the optimizations happened.

Our future work will involve applying SAA to other popular compiler optimizations such as Global Value Numbering (GVN), Dead Store Elimination (DSE), and Partial Redundancy Elimination (PRE).

Chapter 5

Conclusion

Continuous chunk-based execution is an interesting idea that can improve the experience of parallel programming in many ways. The recent adoption of transactional execution in industry has brought this idea one step closer to reality.

This thesis focused on the compiler aspects of chunk-based execution, investigating ideas that can expose th various hardware mechanisms needed by chunk-based execution to the compiler. First, it investigated the possibility of exposing chunks to the compiler in the form of atomic regions for the purposes of memory ordering speculation. It proved that, with some help from the compiler, chunked-based execution can expose an SC memory model to the programmer which is much easier to understand than traditional weak memory models, all the while offering improved performance. Next, it proposed a general API that exposes signatures to software for the purposes of analyzing memory aliasing behavior and proposed a novel form of function memoization based on it. Lastly, it showed how to improve alias analysis for all traditional compiler optimizations in general by using alias speculation enabled using atomic regions. A fully functioning alias analysis pass based on the idea was implemented upon LLVM, an advanced compiler, after which Loop Invariant Code Motion was tested as a client pass.

References

[1] *Intel Architecture Instruction Set Extensions Programmig Reference*. Intel Corporation, February 2012.

[2] Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. *Western Reseach Laboratory-Compaq. Research Report 95/7*, September 1995.

[3] Wonsun Ahn, Shanxiang Qi, Jae-Woo Lee, Marios Nicolaides, Xing Fang, Josep Torrellas, David Wong, and Samuel Midkiff. BulkCompiler: High-performance sequential consistency through cooperative compiler and hardware support. In *Inter. Symp. on Microarchitecture*, December 2009.

[4] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994.

[5] Thomas Ball and James R. Larus. Efficient Path Profiling. In *International Symposium on Microarchitecture*, December 1996.

[6] Elliot Berk. JLex: A Lexical Analyzer Generator for Java. http://www.cs.princeton.edu/ appel/modern/java/JLex/.

[7] David Bernstein, Doron Cohen, and Dror E. Maydan. Dynamic Memory Disambiguation for Array References. In *International Symposium on Microarchitecture*, pages 105–111, New York, NY, USA, November 1994. ACM Press.

[8] B. Bloom. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 11(7):422–426, July 1970.

[9] Colin Blundell, Milo M.K. Martin, and Thomas F. Wenisch. InvisiFence: Performance-Transparent Memory Ordering in Conventional Multiprocessors. In *International Symposium on Computer Architecture*, June 2009.

[10] E. Borin, Youfeng Wu, M. Breternitz, and Cheng Wang. LAR-CC: Large Atomic Regions with Conditional Commits. In *International Symposium on Code Generation and Optimization*, April 2011.

[11] Edson Borin, Youfeng Wu, Cheng Wang, Wei Liu, Mauricio Breternitz, Jr., Shiliang Hu, Esfir Natanzon, Shai Rotem, and Roni Rosner. TAO: Two-level Atomicity for Dynamic Binary Optimizations. In *International Symposium on Code Generation and Optimization*, April 2010.

[12] Harold Cain and Mikko Lipasti. Memory Ordering: A Value-Based Approach. In *International Symposium on Computer Architecture*, June 2004.

[13] B. Carlstrom et al. Transactional Execution of Java Programs. In *Workshop on Synchronization and Concurrency in Object-Oriented Languages*, October 2005.

[14] Luis Ceze, James Tuck, Calin Cascaval, and Josep Torrellas. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *International Symposium on Computer Architecture*, pages 227–238, Washington, DC, USA, June 2006. IEEE Computer Society.

[15] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. BulkSC: Bulk Enforcement of Sequential Consistency. In *International Symposium on Computer Architecture*, June 2007.

[16] Venkatesan T. Chakaravarthy. New Results on the Computability and Complexity of Points–to Analysis. In *International Symposium on Principles of Programming Languages*, January 2003.

[17] S. Chaudhry et al. Simultaneous Speculative Threading: A Novel Pipeline Architecture Implemented in Sun's ROCK Processor. In *International Symposium on Computer Architecture*, June 2009.

[18] Jaewoong Chung, L. Yen, S. Diestelhorst, M. Pohlack, M. Hohmuth, D. Christie, and D. Grossman. ASF: AMD64 Extension for Lock-Free Data Structures and Transactional Memory. In *International Symposium on Microarchitecture*, December 2010.

[19] Cliff Click. Azul's Experiences with Hardware Transactional Memory. In *HP Labs Bay Area Workshop on Transactional Memory*, January 2009.

[20] Daniel A. Connors, Hillery C. Hunter, Ben-Chung Cheng, and Wen-Mei W. Hwu. Hardware Support for Dynamic Activation of Compiler-Directed Computation Reuse. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, New York, NY, USA, November 2000. ACM.

[21] Daniel A. Connors and Wen-Mei W. Hwu. Compiler-Directed Dynamic Computation Reuse: Rationale and Initial Results. In *International Symposium on Microarchitecture*, pages 158–169, Washington, DC, USA, November 1999. IEEE Computer Society.

[22] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 2001.

[23] Jeff Da Silva and J. Gregory Steffan. A Probabilistic Pointer Analysis for Speculative Optimizations. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2006.

[24] Peter Dahl and Matthew O'Keefe. Reducing Memory Traffic with CRegs. In *International Symposium on Microarchitecture*, December 1994.

[25] Xiaoru Dai, Antonia Zhai, Wei-Chung Hsu, and Pen-Chung Yew. A General Compiler Framework for Speculative Optimizations Using Data Speculative Code Motion. In *International Symposium on Code Generation and Optimization*, March 2005.

[26] J. Dehnert et al. The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges. In *International Symposium on Code Generation and Optimization*, March 2003.

[27] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. DMP: Deterministic shared memory multiprocessing. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2009.

[28] D. Dice et al. Applications of the Adaptive Transactional Memory Test Platform. In *Workshop on Transactional Computing*, February 2008.

[29] Yonghua Ding and Zhiyuan Li. A Compiler Scheme for Reusing Intermediate Computation Results. In *International Symposium on Code Generation and Optimization*, page 279, Washington, DC, USA, March 2004. IEEE Computer Society.

[30] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based Alias Analysis. In *International Conference on Programming Language Design and Implementation*, May 1998.

[31] Xing Fang, Jaejin Lee, and Samuel P. Midkiff. Automatic Fence Insertion for Shared Memory Multiprocessing. In *International Conference on Supercomputing*, June 2003.

[32] David M. Gallagher, William Y. Chen, Scott A. Mahlke, John C. Gyllenhaal, and Wen-Mei W. Hwu. Dynamic Memory Disambiguation Using the Memory Conflict Buffer. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–193, October 1994.

[33] Rakesh Ghiya, Daniel Lavery, and David Sehr. On the Importance of Points-to Analysis and Other Memory Disambiguation Methods for C Programs. In *International Conference on Programming Language Design and Implementation*, June 2001.

[34] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *International Symposium on Computer Architecture*, May 1999.

[35] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional Memory Coherence and Consistency. In *International Symposium on Computer Architecture*, June 2004.

[36] Michael Hind. Pointer Analysis: Haven't We Solved this Problem Yet? In *Workshop on Program Analysis for Software Tools and Engineering*, June 2001.

[37] Susan Horwitz. Precise Flow-insensitive May-alias Analysis is NP-hard. *ACM Transactions on Programming Language System*, January 1997.

[38] Jian Huang and David Lilja. Exploiting Basic Block Value Locality with Block Reuse. In *International Symposium on High Performance Computer Architecture*, page 106, Washington, DC, USA, January 1999. IEEE Computer Society.

[39] Jerry Huck, Dale Morris, Jonathan Ross, Allan Knies, Hans Mulder, and Rumi Zahir. Introducing the IA-64 Architecture. *IEEE Micro*, September 2000.

[40] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 3B: System Programming Guide, Part II*, November 2007.

[41] Amir Kamil, Jimmy Su, and Katherine A. Yelick. Making Sequential Consistency Practical in Titanium. In *International Conference on Supercomputing*, November 2005.

[42] Arvind Krishnamurthy and Katherine A. Yelick. Analyses and Optimizations for Shared Address Space Programs. *Journal of Parallel and Distributed Computing*, November 1996.

[43] V. Krishnan and J. Torrellas. A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Trans. on Computers*, pages 866–880, September 1999.

[44] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, September 1979.

[45] William Landi. Undecidability of Static Analysis. *ACM Letters on Programming Language System*, December 1992.

[46] C. Lattner and V. Adve. LLVM: a Compilation Framework for Lifelong Program Analysis Transformation. In *International Symposium on Code Generation and Optimization*, March 2004.

[47] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World. In *International Conference on Programming Language Design and Implementation*, June 2007.

[48] Kyungwoo Lee and Samuel P. Midkiff. A Two-Phase Escape Analysis for Parallel Java Programs. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2006.

[49] Ben Liblit, Alexander Aiken, and Katherine A. Yelick. Type Systems for Distributed Data Sharing. In *International Static Analysis Symposium*, June 2003.

[50] Jin Lin, Tong Chen, Wei-Chung Hsu, and Pen-Chung Yew. Speculative Register Promotion Using Advanced Load Address Table (ALAT). In *International Symposium on Code Generation and Optimization*, pages 125–134, Washington, DC, USA, March 2003. IEEE Computer Society.

[51] Jin Lin, Tong Chen, Wei-Chung Hsu, Pen-Chung Yew, Roy Dz-Ching Ju, Tin-Fook Ngai, and Sun Chan. A Compiler Framework for Speculative Analysis and Optimizations. In *International Conference on Programming Language Design and Implementation*, June 2003.

[52] M.H. Lipasti, C.B. Wilkerson, and J.P. Shen. Value Locality and Load Value Prediction. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[53] Mikko H. Lipasti and John Paul Shen. Exceeding the Dataflow Limit Via Value Prediction. In *International Symposium on Microarchitecture*, pages 226–237, Washington, DC, USA, December 1996. IEEE Computer Society.

[54] Brandon Lucia, Joseph Devietti, Karin Strauss, and Luis Ceze. Atom-Aid: Detecting and surviving atomicity violations. In *Inter. Symp. on Computer Architecture*, June 2008.

[55] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *International Conference on Programming Language Design and Implementation*, pages 190–200, New York, NY, USA, June 2005. ACM.

[56] Milo Martin, Colin Blundell, and E. Lewis. Subtleties of Transactional Memory Atomicity Semantics. *IEEE Computer Architecture Letters*, July 2006.

[57] Donald Michie. ”Memo” Functions and Machine Learning. In *Nature*, pages 19–22, April 1968.

[58] 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 *Inter. Symp. on Computer Architecture*, June 2007.

[59] P. Montesinos, L. Ceze, and J. Torrellas. DeLorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Inter. Symp. on Computer Architecture*, June 2008.

[60] Andreas Moshovos, Gokhan Memik, Alok Choudhary, and Babak Falsafi. JETTY: Filtering Snoops for Reduced Energy Consumption in SMP Servers. In *International Symposium on High-Performance Computer Architecture*, page 85, Washington, DC, USA, January 2001. IEEE Computer Society.

[61] M. Musuvathi and S. Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In *International Symposium on Programming Language Design and Implementation*, June 2007.

[62] G. Naumovich and G. Avrunin. A Conservative Data Flow Algorithm for Detecting All Pairs of Statements that May Happen in Parallel. In *International Symposium on Foundations of Software Engineering*, November 1998.

[63] Naveen Neelakantam, David R. Ditzel, and Craig Zilles. A Real System Evaluation of Hardware Atomicity for Software Speculation. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2010.

[64] Naveen Neelakantam, Ravi Rajwar, Suresh Srinivas, Uma Srinivasan, and Craig Zilles. Hardware Atomicity for Reliable Software Speculation. In *International Symposium on Computer Architecture*, June 2007.

[65] A. Nicolau. Run-Time Disambiguation: Coping with Statically Unpredictable Dependencies. *IEEE Transactions on Computer*, May 1989.

[66] Michael Paleczny, Christopher Vick, and Cliff Click. The Java HotspotTM Server Compiler. In *Symposium on JavaTM Virtual Machine Research and Technology Symposium*, April 2001.

[67] S.J. Patel and S.S. Lumetta. rePLay: A Hardware Framework for Dynamic Optimization. *IEEE Transactions on Computers*, June 2001.

[68] Jih-Kwon Peir, Shih-Chang Lai, Shih-Lien Lu, Jared Stark, and Konrad Lai. Bloom Filtering Cache Misses for Accurate Data Speculation and Prefetching. In *International Conference on Supercomputing*, pages 189–198, New York, NY, USA, June 2002. ACM.

[69] Matthew Postiff, David Greene, and Trevor Mudge. The Store-load Address Table and Speculative Register Promotion. In *International Symposium on Microarchitecture*, December 2000.

[70] R. Rajwar and J. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *International Symposium on Microarchitecture*, December 2001.

[71] G. Ramalingam. The Undecidability of Aliasing. *ACM Transactions on Programming Language System*, September 1994.

[72] Jose Renau, Basilio Fraguela, James Tuck, Wei Liu, Milos Prvulovic, Luis Ceze, Smruti Sarangi, Paul Sack, Karin Strauss, and Pablo Montesinos. SESC Simulator, January 2005. http://sesc.sourceforge.net.

[73] N. Rinetzky, G. Ramalingam, M. Sagiv, and E. Yahav. On the Complexity of Partially-flow-sensitive Alias Analysis. *ACM Transaction on Programming Language System*, May 2008.

[74] Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Bhandari Aditya, and Emmett Witchel. TxLinux: Using and Managing Hardware Transactional Memory in an Operating System. In *Symposium on Operating Systems Principles*, October 2007.

[75] Kenneth Russell and David Detlefs. Eliminating Synchronization-Related Atomic Operations with Biased Locking and Bulk Rebiasing. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 2006.

[76] S. Sastry, R. Bodik, and J. Smith. Characterizing Coarse-Grained Reuse of Computation. In *Workshop on Feedback-Directed and Dynamic Optmization*, 2000.

[77] S. Sethumadhavan, R. Desikan, D. Burger, C.R. Moore, and S.W. Keckler. Scalable Hardware Memory Disambiguation for High ILP Processors. In *International Symposium on Microarchitecture*, pages 399–410, December 2003.

[78] Dennis Shasha and Marc Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. *Transactions on Programming Languages and Systems*, April 1988.

[79] Avinash Sodani and Gurindar S. Sohi. Dynamic Instruction Reuse. In *International Symposium on Computer Architecture*, pages 194–205, New York, NY, USA, June 1997. ACM Press.

[80] Avinash Sodani and Gurindar S. Sohi. An Empirical Analysis of Instruction Repetition. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 35–45, New York, NY, USA, October 1998. ACM.

[81] G.S. Sohi, S.E. Breach, and T.N. Vijayakumar. Multiscalar Processors. In *International Symposium on Computer Architecture*, pages 414–425, June 1995.

[82] Bjarne Steensgaard. Points-to Analysis in Almost Linear Time. In *International Symposium on Principles of Programming Languages*, January 1996.

[83] Bogong Su, Stanley Habib, Wei Zhao, Jian Wang, and Youfeng Wu. A Study of Pointer Aliasing for Software Pipelining Using Run-time Disambiguation. In *International Symposium on Microarchitecture*, pages 112–117, New York, NY, USA, November 1994. ACM Press.

[84] Sun Microsystems. OpenJDK. http://openjdk.java.net/.

[85] Zehra Sura, Xing Fang, Chi-Leung Wong, Samuel P. Midkiff, Jaejin Lee, and David Padua. Compiler Techniques for High Performance Sequentially Consistent Java Programs. In *Symposium on Principles and Practice of Parallel Programming*, June 2005.

[86] Josep Torrellas, Luis Ceze, James Tuck, Calin Cascaval, Pablo Montesinos, Wonsun Ahn, and Milos Prvulovic. The Bulk Multicore Architecture for Improved Programmability. *Communications of the ACM*, December 2009.

[87] James Tuck, Wonsun Ahn, Luis Ceze, and Josep Torrellas. SoftSig: Software-Exposed Hardware Signatures for Code Analysis and Optimization. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, January 2008.

[88] E. Vallejo, M. Galluzzi, A. Cristal, F. Vallejo, R. Beivide, P. Stenstrom, J.E. Smith, and M. Valero. Implementing Kilo-Instruction Multiprocessors. *International Conference on Pervasive Services*, July 2005.

[89] Virtutech. Simics. http://www.simics.net/.

[90] Christoph von Praun and Thomas R. Gross. Static Conflict Analysis for Multi-Threaded Object-Oriented Programs. In *Conference on Programming Language Design and Implementation*, June 2003.

[91] Kevin R. Wadleigh and Isom L. Crawford. *Software Optimization for High Performance Computing: Creating Faster Applications*. Prentice Hall, 2000.

[92] Thomas F. Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Mechanisms for Store-Wait-Free Multiprocessors. In *International Symposium on Computer Architecture*, June 2007.

[93] Robert P. Wilson and Monica S. Lam. Efficient Context-sensitive Pointer Analysis for C programs. In *International Conference on Programming Language Design and Implementation*, June 1995.

[94] Youfeng Wu, Dong-Yuan Chen, and Jesse Fang. Better Exploration of Region-level Value Locality with Integrated Computation Reuse and Value Prediction. In *International Symposium on Computer Architecture*, pages 98–108, New York, NY, USA, June 2001. ACM Press.

[95] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *Inter. Symp. on High Performance Computer Architecture*, February 2007.

[96] L. Ziarek et al. A Uniform Transactional Execution Environment for Java. In *European Conference on Object-Oriented Programming*, July 2008.