HARDWARE ATOMICITY FOR COMPILER-DIRECTED CONTROL SPECULATION

BY

NAVEEN NEELAKANTAM

DISSEETATION

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, 2011

Urbana, Illinois

Doctoral Committee:

Professor Craig B. Zilles, Chair & Director of Research
Professor Vikram S. Adve
Professor Wen-mei W. Hwu
Doctor Ravi Rajwar, Intel Corporation
Doctor Suresh Srinivas, Intel Corporation
Professor Marc Snir
Abstract

This dissertation introduces the atomic region as a novel compiler abstraction which eases the development of speculative compiler optimizations. As this dissertation will show, speculation enables a compiler writer to exploit dynamically occurring opportunities which would otherwise be difficult or even impossible to expose. Despite their potential, speculative optimizations typically involve complex implementation and significant compiler re-engineering. In comparison, the atomic region abstraction is both simple to incorporate into a compiler infrastructure and also exposes speculative opportunity to existing and unmodified optimizations.

The utility of the atomic region abstraction largely derives from its use of hardware atomicity—the execution of a region of code either completely, and as if all operations in the region occurred at one instant, or not at all. Hardware atomicity is an architectural primitive which provides software with a simple and intuitive model of execution, namely the ability to either explicitly commit or rollback a region of code.

The atomic region abstraction leverages hardware atomicity to enable a compiler writer to easily reason about and implement speculative optimization. In the atomic region abstraction, the compiler encapsulates commonly executed regions of the program using the hardware atomicity primitive. This permits the compiler to generate a speculative version of the code where uncommonly executed code paths are completely removed such that they do not need to be considered in (and hence do not constrain) a region’s optimization. Pruned paths are converted into assert operations that trigger an abort in the uncommon case that one of these paths is needed. On an abort, hardware reverts state back to the beginning of
the region and transfers control to a non-speculative version of the code.

Two implementations of the atomic region abstraction are also presented, the first of which demonstrates potential for a 10-15% average performance improvement in the context of a Java Virtual Machine, albeit running on simulated hardware. Incorporation of the atomic region abstraction is shown to be a simple and, thereby, a cost-effective means for exposing speculative optimization opportunities to the compiler.

The second implementation leverages a real system to demonstrate that at least some of these gains are achievable in practice. By incorporating the atomic region abstraction into the dynamic translator of the Transmeta Efficeon processor, practical concerns related to the identification of speculative opportunities and adaptation to mis-speculations are explored. In the context of this real system, a straightforward implementation using simple control mechanisms is shown to be sufficient to achieve a 3% average performance improvement.
Acknowledgments

Writing a dissertation is a personal journey, but it is not undertaken alone. Along the way, I have been honored to gain the advice, feedback, encouragement, and support of a number of people. Not all influenced the technical content of this dissertation, but all were essential to its completion. To everyone involved, I would like to extend my heartfelt gratitude.

Throughout, I have benefited from the guidance and intellectual inspiration provided by my adviser, Prof. Craig Zilles. His hard work and dedication to his students is clear for all to see. It is something more to experience it firsthand. This dissertation, as well as the work that fueled it, had as an embarkation point a stimulating and inviting research environment, the creation of which can be attributed solely to Craig.

In this environment, I had the opportunity to observe the research process in an exemplary form. Specifically, I watched Pierre Salverda develop, under Craig’s advisement, his research ideas into several sound publications and a superb dissertation. Imitation is the sincerest form of flattery, and in that regard I have striven to emulate the balance of exploration and completeness that I observed in Pierre’s work.

I have also benefited from several important experiences that each shaped the path I would follow. Early on, I had the pleasure of working closely with Prof. Sanjay Patel and his research group. I gained much from those experiences, and my interests in computer architecture truly began there.

Later, the committee for my first qualifying exam provided me with what, in hindsight, has turned out to be an inflection point in my intellectual development. I have come to believe that failure merely provides an opportunity for growth. It motivates change, it
motivates increased effort, and it motivates self-determination. I am proud of my failure, it led me to all of my successes.

Along the way, I was fortunate for the many discussions and shared academic experiences that the excellent students attending the University of Illinois have to offer. From reading groups to ad-hoc exchanges at a whiteboard, these experiences shaped my intellectual development in ways too numerous to mention. As peers, we also provided one another with much needed support, encouragement, and compassion. I will always reflect fondly on my time at UIUC. There are surely numerous students that I owe my thanks, but I would like to explicitly acknowledge Mayank Agarwal, Lee Baugh, Luis Ceze, Jeffrey Cook, Mike Fertig, John Kelm, Alex Li, Pablo Ortego, Pradeep Ramachandran, Nicholas Riley, James Roberts, Pierre Salverda, Francesco Spadini, Sam Stone, and Karin Strauss.

A special thanks also goes to Reverend Dr. Tipp Moseley, and the $924 bet that I lost to him. In his own way, Tipp tried to encourage me to complete my dissertation more quickly and move onto the next stages of my life. In the end, I am happy to have given this dissertation the time I felt it needed, despite the growing sum of our bet.

I would like to thank my Ph.D. committee members, Prof. Craig Zilles, Prof. Vikram Adve, Prof. Wen-mei Hwu, Dr. Ravi Rajwar, Dr. Suresh Srinivas, and, in particular, Prof. Marc Snir, for their patience, attention to detail and helpful feedback. I would also like to thank my managers at Intel, David Ditzel, Stephen Lee, and Suresh Srinivas, who have always supported my academic pursuit. They simultaneously encouraged my ideas and exposed me to the real world concerns that are often overlooked in our field. This dissertation and my research benefited from the exposure.

Finally, I would like to thank my family and my friends for the unwavering support that they have provided. Above all, I would like to thank Kyle Hartzell. A Ph.D. is not only a very personal journey but is also a very selfish one. Kyle has excused my absences without question, and she has always encouraged my efforts. I am a lucky man.
# Table of Contents

List of Tables ................................. ix
List of Figures ................................. x
List of Abbreviations ......................... xii
Preface ........................................ 1

Chapter 1 Introduction .............................. 9

Chapter 2 The Atomic Region Abstraction ............... 13
  2.1 Hardware Atomicity ........................... 13
  2.2 Software Speculation Using Atomic Regions ......... 16

Chapter 3 Limitations of Classical Optimization ......... 21
  3.1 Aggressive Compiler Optimization Opportunities .... 22
    3.1.1 Partial Redundancy Elimination .................. 22
    3.1.2 Partial Dead Code Elimination .................. 23
    3.1.3 Procedure Inlining .............................. 26
    3.1.4 Control Flow Restructuring ...................... 28
    3.1.5 Partially Redundant Array Bounds Check Elimination 32
  3.2 Opportunities in Multithreaded Systems ............ 34
    3.2.1 Synchronization Elimination ...................... 35
    3.2.2 Synchronization Optimization .................... 36
    3.2.3 Constraints on Binary Optimization .............. 37

Chapter 4 Speculative Compiler Optimization ............ 39
  4.1 Region Shape ............................... 39
  4.2 Misspeculation Recovery ....................... 41
  4.3 Speculation Strategy .......................... 45
    4.3.1 Data Speculation .............................. 47

Chapter 5 Study of Control Bias in Integer Programs .... 49
  5.1 Cost-benefit Tradeoff for Control Speculation ....... 49
  5.2 Previous Techniques for Detecting Branch Bias ........ 51
  5.3 Characterization of Changing Branches .............. 54
Chapter 6 Atomic Regions for Managed Languages

6.1 Opportunities in Managed Languages

6.1.1 DaCapo Xalan example

6.1.2 DaCapo Jython example

6.2 Providing Hardware Atomicity

6.2.1 Checkpoints and Hardware Atomicity

6.2.2 Microarchitectural Implications

6.3 Forming and optimizing regions

6.4 Experimental method

6.5 Results

6.5.1 Understanding the Variation

6.5.2 Architectural Analysis of Atomic Regions

6.5.3 Microarchitectural sensitivity

6.5.4 Limitations of the existing compiler

6.6 Conclusion

Chapter 7 Atomic Regions for Dynamic Translation

7.1 SPECint 2000 Vortex example

7.2 SPECint 2000 GCC example

7.3 Background

7.3.1 Efficeon Processor Architecture

7.3.2 CMS Overview

7.4 Atomic Regions in CMS

7.4.1 Hardware Atomicity

7.4.2 Incorporating Atomic Regions into CMS

7.4.3 Monitoring Speculations

7.4.4 Eliminating Redundant Asserts

7.5 Evaluation

7.5.1 System Configuration

7.5.2 Experimental Results

7.6 Conclusions

Chapter 8 Atomic Region Memory Model

8.1 Formal Specification of Multithreaded Programs

8.2 Formal Specification of Atomic Regions

8.3 Constraints on Atomic Region Formation

8.4 Reorderings Permitted with Atomic Regions

8.4.1 Control Speculation and Safe Reordering

Chapter 9 Conclusion
List of Tables

5.1 Simulation data sets and run length. ........................................... 52
5.2 Model Parameters. ................................................................. 57
5.3 Model Transition Data. ............................................................ 60
5.4 Model Sensitivity. ................................................................. 63

6.1 Baseline processor parameters .................................................. 84
6.2 DaCapo benchmarks used in evaluation. ..................................... 85
6.3 Atomic region statistics. .......................................................... 89

7.1 Efficeon atomicity primitives. .................................................. 109
7.2 Evaluation system configuration ................................................. 117
7.3 Atomic region statistics. .......................................................... 119
7.4 Static code statistics. ............................................................... 121
## List of Figures

1. Normalized SPECint\_rate scores for Intel x86 processors. ................................................. 3
2. Normalized SPECint scores for Intel x86 processors. ................................................................. 4

2.1 Software usage of hardware atomicity. ..................................................................................... 17
2.2 Atomic region formation by the compiler. .................................................................................. 18

3.1 Example of an unsafe partial redundancy elimination opportunity. ................................. 23
3.2 Example of partial dead code elimination opportunities. ......................................................... 24
3.3 Example of subroutine inlining opportunities. ......................................................................... 26
3.4 Example of partially-redundant conditional control flow. ....................................................... 29
3.5 Partial dead code elimination that requires control flow restructuring. ............................... 30
3.6 Partial redundancy elimination that requires control flow restructuring. ............................ 31
3.7 Fully and partially redundant bounds check elimination opportunities. .............................. 33
3.8 Example of optimizations disallowed by memory model constraints. .................................... 38

4.1 Example of trace scheduling bookkeeping complexity. ......................................................... 40

5.1 Speculative removal of biased branches versus misspeculation rate. ................................ 51
5.2 Five static branches with initially invariant behavior. ............................................................... 55
5.3 A finite-state machine model for branch behavior characterization. ..................................... 56
5.4 Reactive control performs comparably with self-training. ..................................................... 59
5.5 Misprediction rate when a biased branch transitions from being biased. ............................ 62

6.1 An example Java method with hot and cold paths. ................................................................. 66
6.2 Compiler-based redundancy removal. ....................................................................................... 68
6.3 Atomic region optimization. ...................................................................................................... 69
6.4 Complexity of Compiler Optimizations. .................................................................................... 70
6.5 Region formation: ...................................................................................................................... 80
6.6 Performance analysis infrastructure. .......................................................................................... 83
6.7 Execution time speedups. ........................................................................................................... 87
6.8 Micro-operation (uop) reduction. ............................................................................................... 88
6.9 Sensitivity to hardware atomicity implementation. ................................................................. 92

7.1 Potential for atomic region optimizations. ................................................................................ 96
7.2 CMS baseline optimization. ...................................................................................................... 97
7.3 Optimizations enabled by atomic regions. ................................................................................ 98
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFG</td>
<td>Control Flow Graph</td>
</tr>
<tr>
<td>CMS</td>
<td>Code Morphing Software</td>
</tr>
<tr>
<td>DFG</td>
<td>Dataflow Graph</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
</tr>
<tr>
<td>DRLVM</td>
<td>Dynamic Runtime Layer Virtual Machine</td>
</tr>
<tr>
<td>ILP</td>
<td>Instruction Level Parallelism</td>
</tr>
<tr>
<td>ISA</td>
<td>Instruction Set Architecture</td>
</tr>
<tr>
<td>IPC</td>
<td>Instructions Per Cycle</td>
</tr>
<tr>
<td>IR</td>
<td>Intermediate Representation</td>
</tr>
<tr>
<td>JIT</td>
<td>Just-In-Time</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>LIT</td>
<td>Long Instruction Trace</td>
</tr>
<tr>
<td>LOC</td>
<td>Lines Of Code</td>
</tr>
<tr>
<td>OOO</td>
<td>Out-Of-Order</td>
</tr>
<tr>
<td>PC</td>
<td>Program Counter</td>
</tr>
<tr>
<td>PDE</td>
<td>Partial Dead-code Elimination</td>
</tr>
<tr>
<td>PRE</td>
<td>Partial Redundancy Elimination</td>
</tr>
<tr>
<td>SLE</td>
<td>Speculative Lock Elision</td>
</tr>
<tr>
<td>SMT</td>
<td>Simultaneous Multithreading</td>
</tr>
<tr>
<td>SSA</td>
<td>Static Single Assignment</td>
</tr>
</tbody>
</table>

xii
<table>
<thead>
<tr>
<th>TLP</th>
<th>Thread Level Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLB</td>
<td>Translation Lookaside Buffer</td>
</tr>
<tr>
<td>VLIW</td>
<td>Very Long Instruction Word</td>
</tr>
</tbody>
</table>
Preface

High-performance processors enable software innovation and increased program functionality. Historically, software has relied on hardware to provide continual performance improvements through technology, frequency, and architectural innovations. However, shrinking power envelopes and diminishing returns from the pursuit of instruction-level parallelism (ILP) have raised serious questions about where such performance will come from in the future. The computer industry envisions a future of concurrency and increased core counts per die. Unfortunately, the lack of a viable concurrent programming model for general-purpose programs means that, at least for the foreseeable future, achieving high performance by improving single-thread execution remains critical. Furthermore, efficiency in achieving performance has become a first-class constraint, and hardware research has re-focused towards efficient processor designs and will continue to innovate in that direction.

As a result, I believe that software and in particular the compiler will play an increasingly critical role in efficiently enabling performance, especially as part of the trend toward richer, safer, and more dynamic programming environments. Likewise, hardware will play a complementary role by providing features to enable simpler and more powerful compiler designs. To that end, this dissertation demonstrates that one such feature, hardware atomicity both improves the effectiveness of classical compiler optimizations and simplifies the implementation of new compiler optimizations. In particular, hardware atomicity permits the compiler to trivially and effectively employ speculative optimizations, an otherwise complex undertaking.
Recent Trends in Processor Performance

The semiconductor manufacturing trend predicted by Gordon Moore in 1965 [78], the oft quoted Moore’s Law—that the number of transistors per integrated circuit will double every two years\(^1\)—has been sustained by the semiconductor industry for the past four decades. This doubling of transistor density has enabled designers of high-performance processors to translate Moore’s Law into a similar performance trend in which processor performance roughly doubled every eighteen months\(^2\)—equivalent to a growth rate of 60% per year.

Until recent years, these gains were achieved by improving the performance of an individual processor core through a combination of frequency scaling and an increased number of executed instructions per cycle (IPC). However, semiconductor scaling trends and the quadratic circuit complexity of further exploiting ILP have now made it impractical to sustain as rapid a pace of performance improvements per processor core. The computer industry has instead chosen to increase the number of processor cores contained on a single die thereby relying on thread-level parallelism (TLP) to sustain historical performance trends.

As a demonstration, Figure 1 shows SPECint\(_{\text{rate}}\) scores collected in recent years by Intel for their x86 processors. All data was taken from the SPEC website. Scores through 2006 are from the SPECint 2000 benchmark suite [96], and scores after 2006 are from the SPECint 2006 benchmark suite [97]. All scores have been normalized using a conversion factor computed by comparing SPECint 2000 and 2006 scores of identical systems. Also shown on the graph is a 60% per year improvement trendline and transition points where core counts were increased or Simultaneous Multithreading (SMT) [100] was added in the tested systems.

The SPECint\(_{\text{rate}}\) score measures the throughput capabilities of a system by measuring the time it takes to run multiple concurrent instances of each benchmark. Therefore, the

---

\(^1\)His original prediction was that the number of transistors per integrated circuit would double every year but was later revised [56].

\(^2\)First mention of this trend is often attributed to Moore himself. However, Moore attributes it to Dave House [56].
Figure 1: Normalized SPECint rate scores for Intel x86 processors. The vertical lines depict a change of core count or multithreading support in the measured systems.

“rate” score benefits both from increases in single-thread performance as well as increases in TLP. As the figure shows, Intel processors have been able to sustain historical performance trends by increasing core count and by adding multithreading features.

Unfortunately, this shift in focus has a tradeoff. Whereas increasing the performance of a single processor core enables unmodified programs to execute more quickly, increasing the number of processor cores per die only improves the performance of parallel programs. In general this requires converting a single-thread program into a multithreaded, parallel implementation, a far from trivial undertaking.

Figure 2 shows SPECint scores collected for Intel x86 processors over the past decade to demonstrate recent single-thread performance trends. As in Figure 1, data was taken from the SPEC website, all scores through 2006 are from the SPECint 2000 benchmark suite, and all scores have been normalized using a conversion factor computed by comparing SPECint 2000 and 2006 scores of identical systems. In addition to illustrative trendlines, the figure also includes SPECint scores after performance benefits attributable the compiler
have been factored away (using a conversion factor computed by comparing the performance of identical hardware systems evaluated using different Intel compiler versions).

The figure shows that recent improvements in single-thread performance have indeed fallen short of the 60% per year trend of the past. On the other hand, performance has not stagnated and single-thread performance continues to improve at a rate of 30% per year. Such a performance trend—equivalent to a doubling in performance every three years—is worth sustaining especially considering its generality: single-thread performance benefits both sequential and parallel applications.

After adjusting SPECint scores by removing benefits provided by the compiler it becomes apparent that this recent trend in single-thread performance improvements results from more than just changes in the hardware. More specifically, hardware changes alone appear to have been providing a 25% per year improvement until 2007 and only a 10% per year improvement afterward. If it were not for changes in the compiler, two-thirds of the single-thread performance improvements achieved in the course of the past decade would have been
lost. The performance lost in the past two years would have been even more severe.

Therefore, the compiler has already become a key factor to providing sustained performance improvements. Furthermore, it seems that in the past two years hardware has become even more constrained in its ability to improve single-thread performance and Intel, at least, has relied on compiler advancements to fill the gap. Some of these performance improvements result from specialized hardware extensions, such as new vector instructions, but even these require improvements in compiler technology, such as auto-vectorization, to expose their performance potential. I believe this trend will continue and that the compiler will play an increasingly central role in the quest for ever greater single-thread performance.

Factors Influencing Processor Performance

The recent emphasis by the computing industry on multicore processors and concurrent programming has largely been borne of necessity not choice. There are two major reasons for this paradigm shift: unfavorable technology trends and diminishing returns from ILP.

Technology trends: Improvements in semiconductor manufacturing technology have historically enabled dramatic and consistent frequency increases every process generation [22]. Unfortunately, recent years have witnessed a fundamental shift in these trends for two major reasons. First, wire delay is becoming an increasingly dominant and constant factor to determining cycle time. Even if transistors were to continue to scale and exhibit reduced delays, the delay of the wires used to interconnect them is not scaling proportionally. Second, sub-threshold leakage has further constrained designs. Processors are designed to operate below fixed power dissipation ceilings, and, in the past, transistor scaling enabled a reduction of gate threshold voltages which thereby provided the power headroom necessary for increasing the processor frequency. Recently, however, gate threshold voltages have been kept relatively constant to keep sub-threshold leakage in control. The causes are quite different, but the result is the same: transistor technology scaling can no longer enable dramatic increases in
Diminishing returns from ILP: The pursuit of instruction-level parallelism drove many of the architectural innovations of the past decades. This pursuit manifested itself in three complementary ways each sharing the same goal: execute an increasing number of instructions concurrently and thereby reduce the effective latency of each individual instruction. However, all three of these techniques are plagued by diminishing returns and have limited ability to further improve performance. First, lengthening a processor’s pipeline depth is increasingly difficult to accomplish without also lengthening performance critical loops such as the branch-mispredict penalty and load-to-use penalty [21]. Second, wide-issue processors are infeasible due to the quadratic interconnect complexity of widening bypass and scheduling logic and adding cache and register ports. Third, building larger scheduling windows for out-of-order processors is difficult because doing so increases the size of already timing-critical structures such as the re-order buffer, scheduler, load-store queue and register file. Therefore, dramatic increases in ILP are unlikely to be achievable on future processors.

As a result of these trends, the computing industry has refocused away from single-thread performance to multithreaded computing. The industry has chosen to invest growing transistor budgets by incorporating multiple cores onto a single die and improving single core throughput with hardware multithreading [68, 100].

However, I believe this hardware-centric view is overly pessimistic. There are two software factors which bear mention: a growing interest in managed languages and dynamic binary optimizations.

Managed Languages: Languages such as Java and C# have become pervasive over the past decade. These languages are typically compiled to an intermediate binary representation, or bytecode, and are executed inside of a virtual machine [35, 70]. These languages are managed because they require sophisticated runtime support from the virtual machine to support features such as portability, garbage collection, dynamic class loading and reflection.
These features, in addition to others also found in non-managed languages such as bounds checking and polymorphism, enable programmers to write expressive and maintainable applications quickly and more easily. Furthermore, the underlying virtual machines have recently been adapted to support dynamic languages such as Python, Ruby and Javascript which add yet more programmability features such as dynamic typing [28,60].

However, these features impose a performance cost onto the managed language implementation. To compensate, high performance implementations of managed languages use just-in-time (JIT) compilation techniques to enable a runtime compiler to optimize for common case behavior while still retaining programmability features. Though already sophisticated, JIT compilers continue to innovate increasingly aggressive techniques to improve performance. Efficient hardware features to ease managed language implementation or enable more aggressive JIT compilation could enable significant single-thread performance improvements, potentially avoiding the hardware complexity issues mentioned above.

**Dynamic Binary Optimization:** Several recent systems have integrated a dynamic compiler, also known as a *translator*, to enable runtime program optimization. Similar to the JIT compilers mentioned above, dynamic translators such as Transmeta’s Code Morphing Software (CMS) [32], IA-32 EL [8] and the Godson x86 translator [52] also incorporate sophisticated runtime optimizers to enable high-performance execution of x86 binaries on dissimilar hardware. These systems are capable of identifying frequently executed pieces of a running program, analyzing control flow and dataflow structure, optimizing for observed common case behavior, and deploying optimized codes into the running system. Here too, the opportunity exists for single-thread performance improvement by providing efficient hardware features to enable more effective runtime compiler optimizations.

Given these software factors, I believe that hardware architects should embrace the needs of the language and nurture the optimization potential of a runtime compiler. Even with limited hardware features and without the ability to adapt to runtime behavior, static compilers have been instrumental in sustaining single-thread performance growth, particularly
in the past few years. It stands to reason that with more general hardware support and with runtime information the compiler would be capable of continued performance improvements.

To that end, the focus of my dissertation is on one such feature: *hardware atomicity*. The dissertation introduces hardware atomicity as a powerful feature which enables a compiler to easily and effectively optimize for common case program behavior. It is viable in any runtime optimization system, and I will describe how it was incorporated into both a JIT compiler and dynamic binary optimizer.

To be clear, I do believe that industry and academia have made sound and pragmatic decisions. If we are to sustain historical trends of 60% performance growth per year, then multicore and multithreaded processors demand attention. That being said, I do not believe that the quest for single-threaded performance has reached an impasse, and this dissertation proposes one means to furthering the journey.
Chapter 1

Introduction

In his landmark paper “Compilers and Computer Architecture,” William Wulf identifies three principles (regularity, orthogonality, and composability) that instruction sets should adhere to in order to simplify compiler implementations, thereby improving the code quality that is practically achievable [106]. Each time these principles are not observed, an additional set of special cases must be considered during compilation in order to generate the best possible code for a given program. While architectures that ignore these principles do not, in theory, preclude the building of compilers that generate the highest performance code, in practice the quality of code suffers as many compiler implementations will be unable to justify the additional software complexity required.

Akin to the simplifying principles set forth by Wulf, I see hardware atomicity—the execution of a region of code either completely, and as if all operations in the region occurred at one instant, or not at all—as a fundamental architectural feature that enables a range of sophisticated uses by software. Hardware atomicity provides a simple and intuitive model of execution which enables software writers to reason about implementations that would otherwise be too complex to consider.

This dissertation focuses on one possible use of hardware atomicity: the atomic region. An atomic region is a compiler abstraction in which hardware atomicity primitives are used to encapsulate a program region being optimized. In doing so, the region appears to execute completely and instantaneously or not at all. This permits the compiler to generate a speculative version of the code where uncommonly executed code paths are completely removed, so that they need not be considered in (and hence do not constrain) a region’s optimization.
If one of these pruned paths needs to be executed, the region will be aborted—reverting back to the state at the beginning of the region—and control will be transferred to a non-speculative version of the code.

Speculative optimizations are important for achieving high performance in many integer and enterprise applications because the control flow intensive nature of these programs prevents non-speculative compiler approaches from generating efficient code. In fact, the presence of frequent control flow can be a significant inhibitor of compiler optimizations, even when a significant fraction of the control flow is strongly-biased and compilation is performed with an accurate profile.

This dissertation will demonstrate both the ability of atomic regions to expose speculative opportunity and the simplicity in which the atomic region abstraction can be incorporated into software optimization frameworks. The atomic region abstraction not only exposes speculative opportunities to unmodified classical compiler optimizations but it can also form the basis of new optimizations—some which would otherwise be prohibitively difficult to implement and others that require atomicity guarantees. In Chapter 2, I will describe how the atomic region abstraction is used by software and the necessary requirements of a hardware atomicity implementation.

In Chapter 3, I will present an informal survey of optimization opportunities that confound classical compiler techniques. Many specific strategies have been proposed to exploit these opportunities but they are often computationally complex and non-trivial to implement. The atomic region abstraction offers the ability to exploit the practically occurring instances of these opportunities.

At a cursory level, the atomic region abstraction shares many similarities with previously proposed speculative optimization opportunities. However, by characterizing the atomic region abstraction and previously proposed speculative optimization according to three dimensions—region shape, misspeculation recovery, and speculation strategy—Chapter 4 will show that atomic regions provide a novel, powerful, and simple tool to the compiler
Nevertheless, the utility of the atomic region abstraction depends upon the occurrence of strongly-biased control flow in programs. The manner and frequency of such occurrences is the subject of Chapter 5. As the chapter will show, biased control flow is common but can be short lived. Exploiting biased control flow therefore requires mechanisms to react to changing program behaviors. Reacting to changing program behaviors also enables a dynamic compilation system to identify biased control flow as effectively as, and sometimes better than, a self-trained static compilation system.

The simplicity of the atomic region abstraction makes it well suited for a variety of compiler frameworks and processor architectures. In demonstration of this point, I incorporated the atomic region abstraction into two different dynamic compilation systems.

Chapter 6 describes how the atomic region abstraction was incorporated into the JIT compiler of a Java virtual machine (JVM) for an out-of-order processor. To evaluate the resultant hardware and software co-designed system a new simulation methodology needed development: a description of this methodology is provided as a secondary contribution of this dissertation.

Chapter 7 describes how the atomic region abstraction was also incorporated into the dynamic binary translator of a very long instruction word (VLIW) processor. Evaluated on a real system, the chapter also describes an implementation of the reactive control mechanisms called for in Chapter 5. The control mechanism described is sufficient to achieve speedups on a real system.

The execution guarantees of the atomic region abstraction also enable its use in multiprocessor systems. However, any such use must still satisfy the constraints specified by the memory consistency model of a system. Chapter 8 formally specifies the execution semantics of the atomic region abstraction. It also shows that the atomic region is compatible with modern memory consistency models and enables optimization.
A retrospective of this work is provided in Chapter 9, including a discussion of open questions left for future research.
Chapter 2

The Atomic Region Abstraction†

The central focus of this dissertation is the atomic region and its effectiveness as an abstraction for speculative compiler optimization. The atomic region abstraction simplifies many optimizations, and this simplification derives from the execution guarantees and clean interface provided by hardware.

In this chapter, I describe these hardware requirements, which are collectively referred to as hardware atomicity, and mention potential implementations that satisfy them. I then described how a software optimizer leverages hardware atomicity to implement the atomic region abstraction and provide an abstract example of the optimization opportunities exposed.

2.1 Hardware Atomicity

An implementation of hardware atomicity must satisfy three important criteria. First, hardware must provide the illusion of atomic execution—that a region of code either executes completely and instantaneously or does not execute at all. Second, hardware must provide this illusion without introducing performance overheads in the common case. Third, a set of simple hardware atomicity primitives should be provided that enables intuitive software control.

Providing atomic execution: The illusion of atomic execution provides important guarantees to software. Foremost, it guarantees that a region of code, as specified by software,
will either completely execute or will appear as if it never executed. This enables trivial recovery in case of a misspeculation because software simply relies on hardware to discard all updates from the current region and then restarts execution elsewhere (e.g., in a non-speculative version of the same region).

In addition, hardware must guarantee that the same region of code will appear to execute instantaneously or, in case of a misspeculation, will have no observable effects. This enables a software optimizer to easily satisfy the memory ordering rules of the system. To maintain this illusion, hardware must not permit other processors or devices to observe any of the effects of an atomically executed region until it has committed (see Chapter 8). Likewise, none of the effects of a region should be observable in the case of a misspeculation.

The implementations discussed in this dissertation provide these guarantees through the following steps: 1) creating a register checkpoint upon entering the region, 2) tracking all memory addresses accessed by instructions in the region, 3) buffering all updates performed by these instructions, 4) using an ownership-based cache coherence protocol to detect conflicting accesses from other agents, 5) discarding updates on a conflict, and 6) committing the updates in the cache atomically.

The mechanisms necessary for each of these steps have much in common with prior hardware proposals ranging from efficient management of resources in out-of-order processors [4, 29, 55, 75] to hardware transactional memory [48, 67]. A number of implementations are possible, of which this dissertation explores two. The first is based on hardware similar to that proposed for speculative lock elision [88] and is described in Section 6.2. The second leverages hardware already provided by the Transmeta Efficeon processor [89, 90] and is described in Section 7.3.1.

**Good common case performance:** Though various implementations of atomic execution are possible, not all are suitable for the atomic region abstraction. Because the atomic region abstraction is intended to improve single-thread performance, hardware must introduce as little performance overhead as possible. Commonly occurring overheads could overwhelm
the performance opportunities exposed by the abstraction.

Most importantly, the performance overheads of taking and committing atomic region checkpoints should have minimal overhead. The speculative opportunities exposed by the atomic region abstraction may only provide a small performance improvement on each execution of a region; if the overhead of using the atomic execution hardware is significant, the net result could be a performance penalty.

Fortunately, some aspects of the atomic execution hardware are tolerant of performance overhead. For example, the atomic region abstraction is intended to exploit highly-biased opportunities and misspeculation are assumed to be rare. Therefore, the performance of a hardware rollback (and the corresponding redirection to non-speculative code) can tolerate some performance overheads.

Likewise, a hardware implementation of atomic execution need not support all possible uses. Best effort hardware is sufficient, as long as it covers common uses of the atomic region abstraction. For example, hardware does not need to support unbounded region sizes, I/O operations or exceptions. In these situations, hardware is expected to implicitly abort the region. This hardware flexibility is afforded by software control mechanisms that rein in excessive misspeculation and also enable software to reoptimize regions that exceed hardware support.

**Simple primitives:** Atomic execution hardware should ideally be exposed using a set of simple and concise primitives. This simplicity has two purposes: it eases the design of the software that must use it, and it provides an interface to hardware that allows for implementation flexibility.

These primitives should be both intuitive and general. For example, hardware could expose three simple instructions: one for beginning an atomic region, another for completing it, and a final instruction to abort the region in case of a misspeculation:

- **aregion.begin** `<alternate PC>`: This instruction signals the start of a speculatively-
optimized region and creates a recovery checkpoint. Subsequent register and memory updates are speculative. The alternate PC (program counter) specifies the code address at which execution will resume after an abort.

- **aregion_end**. This instruction ends the region and atomically commits all speculative register and memory updates.

- **aregion_abort**. This instruction permits software to explicitly rollback to the previously taken recovery checkpoint by discarding all speculative register and memory updates. Execution resumes at the alternate PC specified by the most recently executed **aregion_begin**.

Though intended for the atomic region abstraction, these instructions merely manage the underlying mechanisms of atomic execution. They presume knowledge of neither how software might utilize nor how hardware might implement atomic execution. Their generality provides design flexibility to both hardware and software.

Figure 2.1 illustrates the use of these new instructions. Causes for atomic region aborts are communicated to the software via two additional registers. The first register encodes the reasons for an abort (*e.g.*, explicit abort, interrupt, data conflict, exception, etc.). The second register records the program counter of the instruction responsible for an abort (if any). This information enables software to diagnose the cause of aborts and adaptively recompile when necessary.

### 2.2 Software Speculation Using Atomic Regions

This section describes how the compiler exploits hardware atomic regions to improve the generated code quality using an illustrative, but representative example. Figure 2.2(a) shows the control flow graphs (CFGs) of two example methods that depict a common Java idiom: method **foo** calls the synchronized method **bar**, and the compiler considers inlining **bar** into
Figure 2.1: **Software usage of hardware atomicity.** If a speculation succeeds, no abort conditions will be invoked and the execution will reach an `aregion_end` that commits the atomic region. Speculation fails when an abort condition evaluates to true, causing a branch to be taken to an unconditional abort instruction. The abort instruction restores register state, invalidates speculatively written cache lines, and transfers control to the address specified by the `aregion_begin` instruction.

The example depicts several common optimization obstacles. In both methods, extremely rare execution paths (C→B, X→call, and Y→exception) limit the optimization of more frequently executed paths because these infrequent paths tend to use or redefine variables that obscure optimization opportunity. The static size of infrequently executed paths can also prevent the compiler from inlining methods for example, `bar` into `foo`. The
Figure 2.2: **Atomic region formation by the compiler.** (a) Initial control flow graphs (CFGs) for the methods annotated with a control flow profile, (b) a replicated version of the hot paths after partial inlining and trimming of cold paths, (c) and the final CFG for the method foo.

`monitor_enter` and `monitor_exit` intrinsics will expand into the relatively complex CFGs required of high-performance lock implementations [61]. Furthermore, `monitor_enter` and `monitor_exit` are synchronization actions in the Java memory model, which restrict the compiler’s ability to perform optimizations across them [74].

The atomic region abstraction provides the compiler writer with a simple, yet effective means of overcoming these common performance obstacles. It enables the compiler writer to reason about exploiting available optimization opportunity without being concerned about infrequently executed paths or multiprocessor memory models. In essence, the abstraction provides a compiler with the ability to “undo” a region of speculatively optimized code.

For example, in Figure 2.2, the compiler can replace each infrequently executed path with an operation to assert\(^1\) that the path is not followed. Similarly, the compiler can replace the

\(^1\)An assert can be implemented with an unconditional abort, as depicted in Figure 2.1
balanced pair of synchronization actions with an operation to assert that no other thread holds the lock. The all-or-nothing property of atomic regions aids both transformations; if any assertion does not hold, the processor discards the updates performed by the speculative optimizations, making it appear as if the speculative execution had never happened. Furthermore, the instantaneous commit provided by an atomic region allows the compiler to remove synchronization actions (i.e., a software controlled implementation of speculative lock elision [88]) because hardware prevents illegal interleavings of memory operations from other threads.

To exploit these opportunities, the compiler first selects an optimization region as depicted by the CFG in Figure 2.2(b). The region is a subset of the CFG in Figure 2.2(a), where bar has been partially inlined (i.e., infrequent paths have not been inlined) into foo and infrequently executed paths have been removed. The compiler converts the optimization region into an atomic region by duplicating the CFG and placing an aregion_begin at each region entry and an aregion_end at each region exit. The atomic region may contain arbitrary control flow, such as the depicted branch-over, but any path through the region must encounter a single balanced pair of aregion_begin and aregion_end instructions. Specifically, any path through the region must encounter exactly one aregion_begin and exactly one aregion_end.

The resulting atomic region is reconnected into the flow graph for foo by pointing all of A’s in-edges to the aregion_begin and adding an exception edge from the aregion_begin to A. For each infrequent path that has been removed, the compiler ensures correctness by replacing the branch to the removed path with an assert operation to verify that the expected path was taken. By traversing the CFG for this region, the compiler can identify the balanced pair of monitor_enter and monitor_exit intrinsics and convert them into an assertion that the monitor is free. Figure 2.2(c) shows the atomic region formed and the complete CFG for a speculatively optimized foo.

The simplified CFG contained within the atomic region enables the compiler to transform
and schedule the common program paths without having to generate compensation code. Importantly, the compiler uses the existing exception handling mechanisms in its intermediate representation to represent an atomic region. To the compiler, an atomic region abort appears as if an exception occurred in the `aregion_begin` block and transferred control to block A. By inserting the exception edge between these two blocks, the compiler preserves the values needed by the abort path and performs register allocation appropriately.

The assert operations are a simple addition to the intermediate representation and are represented as arithmetic operations that have source values but produce no output. In particular, an assert is neither a control nor an exception producing operation in the compiler intermediate representation—if an assert triggers an abort, hardware will rollback such that an exception appears to occur at the `aregion_begin`. As a result, existing compiler passes can eliminate redundant assertions and schedule them. By using existing compiler mechanisms to represent atomic regions, existing optimization passes do not require modification to exploit the exposed speculative optimization opportunity.
Chapter 3

Limitations of Classical Optimization

Throughout this dissertation, I use the term classical optimization to refer to compiler techniques which consider all possible program paths in their analysis. These techniques are fundamentally non-speculative in that they generate code which is safe for all program paths and does not need recovery code to maintain correctness. They rely instead on proofs and heuristics (possibly augmented by profiling information) to identify and exploit optimization opportunities. These techniques are commonly accepted into standard practice and form an essential part of most compilers. Classical optimizations readily exploit many of the optimization opportunities available in a program, but this chapter will show that other opportunities remain.

I first present a set of examples which demonstrate optimization opportunities that are often difficult to exploit using classical compiler techniques. For each example, I will refer to previously proposed techniques to exploit the available opportunities. In a few cases, a prior proposal has proven to be effective and is used in standard practice. However, many of the examples remain difficult to exploit.

I then describe optimization obstacles that are introduced by multiprocessor memory consistency models. Memory model requirements of both modern programming languages and hardware systems place stringent requirements upon the observability of memory values specified by a programmer and by an application binary. Operations that might be safe to remove or reorder in a single-thread program might not be safe to remove or reorder in a multithreaded program.

To improve readability, abstract and simplified examples are used throughout this chap-
3.1 Aggressive Compiler Optimization Opportunities

In the discussion that follows, five types of opportunities are explored. In each case, previous proposals to exploiting these opportunities are presented, including explanations as to why many have not been adopted into the standard optimization vernacular.

3.1.1 Partial Redundancy Elimination

Partial redundancy elimination (PRE) is a mature topic with several well understood classical solutions [79]. Optimizations such as lazy code motion [65] and SSAPRE [63] are able to efficiently identify and exploit all partial redundancies that can be removed safely. In fact, these techniques produce both computationally optimal and lifetime optimal placements (i.e., no further redundancies can be safely removed and they do not introduce any more register pressure than necessary).

However, they only are able to perform safe reorderings, i.e., introduce computation onto paths on which they would have otherwise already been computed [62]. The necessity of this restriction is demonstrated in Figure 3.1. Hoisting the load in block D eliminates the partial redundancy in the loop but may lead to incorrect program behavior. Specifically, it may introduce an invalid address exception onto the path A→B→C→D.

The opportunity demonstrated in Figure 3.1 could be exploited with hardware support.
for speculation. A mechanism that enables suppression of speculatively raised faults would enable a compiler to hoist the potentially excepting load. Chapter 4 provides a thorough discussion of speculation mechanisms.

Note that speculative partial redundancy elimination is not guaranteed to be computationally optimal—additional computations can be introduced into a program’s execution. However, if guided by accurate profile information, speculative partial redundancy elimination can enable a program to execute fewer operations than non-speculative PRE [71]. Figure 3.1 and the remaining figures in this chapter incorporate precise profile information in order to motivate the possibilities for speculative optimization.

### 3.1.2 Partial Dead Code Elimination

Figure 3.2 depicts the logical dual of partial redundancy elimination: partial dead-code elimination (PDE). Shown in Figure 3.2(a) is a control flow graph in which several values are only consumed in a subset of the possible paths, indicating that performance could be
Figure 3.2: **Example of partial dead code elimination opportunities.** Classical removal of all partially dead operations requires more complex algorithms and code duplication.

improved by eliminating these *partially dead* operations.

In order to render these operations *fully dead* (*i.e.*, compute them only on paths on which they are used) a software optimizer must *sink* (*i.e.*, move in the direction of control flow) them onto the cold paths. However, this is insufficient to eliminate all of the partially dead operations as shown in Figure 3.2(b). A problem arises because of a dependence chain through the partially dead operations, which can only be broken by inserting additional copies into the flow graph as shown in Figure 3.2(c).

This example intuitively shows how classical software optimization techniques can be used to eliminate partially dead code, but belies their complexity and negative side effects. Specifically, a classical implementation of partial dead code elimination requires a polynomial-time algorithm and can introduce significant code bloat.

For example, the partial dead code elimination algorithm introduced by Knoop et al. is optimal in the sense that it eliminates all of the partially dead code that could be removed without modifying the control flow or semantics of the program [66]. However, despite its effectiveness the algorithm is essentially a bidirectional iterative dataflow pass (*i.e.*, on each iteration a backwards pass is first run followed by a forward pass). This has a worst-case time complexity of $O(n^4)$ and even using optimistic assumptions Knoop estimates the time complexity to be $O(n^2)$. 


Furthermore, eliminating all partially dead operations requires inserting additional copies, as demonstrated in Figure 3.2(c). The static code growth due to these additional copies is bounded by $O(b)$ where $b$ is the number of basic blocks in the optimization region, which using optimistic assumptions may be bounded by a constant factor [66]. However, these additional copies not only bloat the static code size, but they effectively increase the compilation cost of all subsequent optimizations (i.e., the variable component of many algorithms increases). Therefore, even with optimistic assumptions the effect on overall compilation time is expected to be significant. This additional cost in compilation time is a serious deterrent to runtime optimization systems.

Within the context of a dynamic binary optimization, the compilation cost becomes prohibitive because the effective number of basic blocks in an optimization region is increased by the need to maintain precise exceptions. Therefore, potentially excepting instruction effectively terminates a basic block by inserting an exception edge to the CFG. If interrupts are enabled within an optimization region, each instruction effectively resides in its own basic block.

It may seem that both the time complexity and space expansion problems of Knoop’s partial dead code elimination algorithm can be alleviated by reframing the optimization as a form of global code motion [26]. Framing the problem in this manner trivially eliminates all code bloat effects—code is relocated rather than copied—and enables linear time algorithms (assuming that dominator tree and loop depth information has already been computed for other optimization passes, as both typically are). However, such a formulation is insufficient for removing all partially dead code and is merely capable of the optimization shown in Figure 3.2(b).

Speculative optimization, and more specifically the atomic region abstraction, enables a compiler to exploit dynamically dead opportunities with fewer copies (see Section 4.3). Partially dead code which occurs in practice is simply a type of dynamically dead opportunity. Whether or not such an opportunity is practical to exploit depends on the execution profile.
Figure 3.3: **Example of subroutine inlining opportunities.** Subroutine inlining removes calling convention overheads and also increases the scope of global optimizations. However, it typically increases static code size and overuse can degrade performance.

In other words, if either of the paths \(A \rightarrow C\) or \(B \rightarrow E\) in Figure 3.2 are rarely executed, then all of the opportunities shown could be exploited by a combination of cutting cold paths and global code motion. If both paths are rarely executed then the global code motion pass is also unnecessary.

### 3.1.3 Procedure Inlining

The pervasive use of procedures in modern programs obscures many optimization opportunities. Shown in Figure 3.3(a) is an interprocedural control flow graph annotated with dynamic branch biases. When invoked from the callsite shown, the procedure \(foo\) contains a redundant operation that is obscured by the call boundary. Furthermore, the procedure call incurs overheads such as saving and restoring registers as specified by the calling convention. A classical approach to exploiting these types of optimization opportunities is procedure inlining.

Procedure inlining essentially expands a called procedure by copying it into the flowgraph...
of its caller as shown in Figure 3.3(b). In doing so, it obviates call related overheads and exposes opportunities that would otherwise be obscured by the call boundary to global optimization passes. However, this benefit comes with three costs: increased pressure on the register allocator, increased compilation time and an increase in static code size.

First, the expansion caused by procedure inlining has the effect of increasing the number of statements—and likewise the number of variables—contained in the optimization flowgraph. This can stress the register allocator and result in suboptimal register allocations. Sias, in his analysis of compiler optimization techniques for explicitly parallel instruction computing (EPIC), found that procedure inlining can stress the register stack engine (a hardware mechanism for managing register spills and fills) and thereby hinder performance [93].

Second, increasing the number of statements in the optimization flowgraph also increases compilation time cost. The cost of many compiler optimizations grows super-linearly with the number of statements in the flowgraph. To prevent compilation cost from growing out of control, compilers use heuristics to tightly limit code expansion caused by inlining. In designing their aggressive inliner, Ayers et al. found that limiting code expansion to 20% enabled them to keep the total compilation time to within 200% of the total compilation time with inlining disabled [6].

Although compilation time may be less of an issue for static compilers, it is of primary concern in runtime optimization systems. In most runtime optimizers, extra time spent compiling must be recovered by producing and sufficiently executing higher performance code. In their study of JVM inlining heuristics, Cavazos and O’Boyle show that an example inlining heuristic nearly always produces higher performance code but increases in compilation cost occasionally result in a reduction of Java system performance [24].

Third, the code expansion incurred by procedure inlining also increases the overall static size of the program being optimized. This increase in static program size, known as code bloat, can be significant. Hank et al. show that aggressive inlining increases the static code size by a ratio of of $4.0x$ on average and by as much as $17.4x$ [46].
Increased static code size can cause performance degradation by increasing the number of instruction cache misses. Similar to the heuristics used to control compilation time, compilers may restrict code expansion to control detrimental side-effects in the instruction cache. For example, the inliner in the IMPACT compiler is tuned to restrict code expansion to a ratio of $2.0x$ and in practice expands the static code size of a program by roughly $1.5x$. Despite these restrictions, the increased static code size can still increase instruction cache misses sufficiently to incur a net program slowdown [93].

Furthermore, increases in static code size may pose further problems to dynamic binary optimization systems. These systems place optimized code translations into software-managed memory known as a translation cache. The translation cache is typically maintained in a fixed-size region of memory that must periodically be reclaimed to make room for new translations [7, 32]. If code bloat is not restrained, the increased pressure upon the translation cache could cause it to thrash. If a program’s entire working set of translations does not fit in the translation cache, then performance of the system will suffer dramatically.

Speculative optimization techniques provide viable alternatives to classical procedure inlining. Several of the techniques described in Chapter 4 enable a compiler to inline only the commonly executed portions of a method into a callsite. In doing so, they expose similar opportunities to classical inlining but with far less code expansion. Some of these techniques are already common to JVM implementations, and the atomic region abstraction enables yet further improvements.

### 3.1.4 Control Flow Restructuring

The opportunities depicted thus far are all able to be exploited without changing the logical control flow structure of the program being optimized. Control flow operations may be replicated, but they still occur in the same order as in the original program. However, additional opportunities exist which cannot be exploited without enabling control flow graph changes. Optimizations based on control flow restructuring—reordering and elimination
Figure 3.4: Example of partially-redundant conditional control flow. (a) The control flow operations are redundant along several of the possible paths, (b) but eliminating all redundancies would require significant code duplication.

of partially redundant control flow operations—have been proposed which expose further optimization opportunities [14, 16, 17, 38, 80, 91].

Unfortunately, as this section will show, prior proposals for identifying and exposing these opportunities are both complex and time consuming. Furthermore, control flow restructuring requires code duplication to maintain program correctness, which has already been shown to introduce additional compilation costs and can degrade performance.

A simple opportunity for control flow restructuring is presented in Figure 3.4(a). It depicts two opportunities to remove branches that depend on conditions that are partially redundant with earlier branches. The first opportunity is a branch which, along one path, reuses the exact condition computed for an earlier branch thereby making the reused condition partially redundant. The second opportunity is a branch condition that is not partially redundant but will always be true if a prior branch condition is also true. In other words,
the earlier branch condition subsumes the later branch condition [80].

Prior work has introduced both intraprocedural and interprocedural algorithms which can identify and exploit these conditional branch opportunities [16, 80]. Though effective, they require complex polynomial time algorithms to identify optimization opportunities for control flow restructuring and require code duplication to exploit them. Figure 3.4(b) shows that exploiting even the example opportunities requires significant code duplication. Because of these issues, control flow restructuring is of questionable practicality in a static compiler and is simply untenable in a dynamic compilation framework. One of the authors of the control flow restructuring works more recently viewed “restructuring as too expensive for a dynamic compiler” [15].

Control flow restructuring can also expose additional opportunities for PDE and PRE. Figure 3.5(a) first shows an opportunity for partial dead-code elimination which shares similarities with the opportunities shown in Section 3.1.2. The difference is that the partially dead code in block C cannot be eliminated without changing the control dependence structure of the program. In particular, exploiting this opportunity requires changing the control
dependence structure of the optimization region as shown in Figure 3.5(b). By making the partially dead operation control dependent upon a later branch it has been rendered totally live (i.e., the operation is only executed on the paths on which it is consumed).

Figure 3.6(a) shows an example of a PRE opportunity which is obscured by a control flow obstacle. As already mentioned in Section 3.1.1, safe PRE techniques would normally be unable to exploit this opportunity. By duplicating code and restructuring the control flow, as shown in Figure 3.6(b), the opportunity can be exposed to safe optimization.

Techniques have been proposed that are capable of identifying and eliminating PDE opportunities like the one shown in Figure 3.5 [14,38,91]. However, similar to the techniques used for eliminating partially redundant or subsumed branches, these techniques require complex analysis algorithms and significant code duplication.¹ A similarly complex technique that also relies on code duplication has been proposed for exploiting PRE opportunities such as the one in Figure 3.6 [17].

¹The revival transformation does not require code duplication, but is unable to expose some PDE opportunities obscured by control flow. The authors themselves note that exposing other interesting PDE opportunities requires code duplication, which they call decision node copying [38]
All of these restructuring techniques also introduce additional complexity to compiler development. Verifying the correctness of optimizations which restructure the control flow semantics of a program is a non-trivial undertaking. In particular, maintaining precise exception semantics while still allowing control flow to be reordered and eliminated can be difficult or even impossible without the hardware support described in Section 4.2.

Furthermore, many of these techniques can produce irreducible flowgraphs [17]. Therefore, they interfere with many existing compiler analysis and optimization routines which rely on reducibility. Reconstructing a reducible flowgraph requires additional compilation time and incurs further code duplication [58].

Speculative optimization offers an easier and more practical approach to exploiting these opportunities, one without the complex analysis and code duplication required by control flow restructuring. For example, the atomic region abstraction converts highly biased control flow into assert operations, thereby converting control operations into dataflow (see Chapter 2.2). Because assert operations are simple dataflow operations, they can be redundancy eliminated. Similarly, they remove the control obstacles to the PDE and PRE opportunities described in this section.

### 3.1.5 Partially Redundant Array Bounds Check Elimination

More recently, a specialized variation of control flow restructuring has been proposed to optimize a common idiom in managed languages, array bounds checking [15]. Array bounds checks can often be eliminated because the array indices are known to reside within the limits of an array, as shown in Figure 3.7(a). The ABCD optimization proposed by Bodik et. al provides a cost-effective approach to identifying and eliminating these fully redundant bounds checks.

Eliminating partially redundant bounds checks, shown in Figure 3.7(b), is more complex and requires control flow restructuring. Bodik et. al show that ABCD can be extended to identify partially redundant bounds checks and further propose a transformation to eliminate
them based on inserting stronger compensating checks. However, simply inserting a stronger check is insufficient as it would change the exception behavior of a program.

Instead, the authors propose an extension to ABCD that uses speculation to optimize the partially redundant bounds check. First, a compensating check is speculatively placed early in the flowgraph which enables eliminating the partially redundant bounds checks. Second, an unoptimized version of the region is generated which contains all of the bounds checks in their original program positions. At runtime, if the compensating check does not detect an out of bounds access then the optimized version of the region is executed, otherwise the unoptimized version is executed so that the out of bounds exception is thrown precisely. This transformation is shown in Figure 3.7(c). Note that in the case of managed languages, further speculation can be used to defer generating the unoptimized version until the compensating check detects a potential out of bounds exception (as described in Section 4.2).
3.2 Opportunities in Multithreaded Systems

Modern multiprocessor systems share physical memory and provide well-defined consistency models which dictate the order in which memory operations appear to execute. Stated differently, memory consistency models restrict the set of values that a particular memory load may return [1]. Memory consistency models have been well-defined from a hardware system perspective for a number of years and have recently been incorporated into modern programming language definitions [19, 59, 74].

Memory models can range from strict models, such as sequential consistency, to relaxed models, such as release consistency. From a performance point of view, the flexibility of relaxed models is preferable because they permit several important optimizations. From a programming and debugging perspective, strict models, particularly sequential consistency, are preferable because they are more intuitive and easier to understand. In order to satisfy these competing goals, programming languages tend towards data-race-free memory models. Data-race-free memory models are indistinguishable from sequential consistency in programs which are synchronized sufficiently to eliminate all data races [3]. These models enable implementations which can use optimizations such as hardware write buffers and compiler-driven register allocation of memory variables. Implementations need only be restricted by the ordering rules imposed via synchronization actions.

Nonetheless, three significant optimization opportunities remain. The first two pertain to the synchronization actions themselves: frequent synchronization introduces excessive performance overhead, and synchronization is often used to protect against data races that rarely manifest. The last is specific to optimization at the instruction set architecture (ISA) level: when optimizing a program binary, the memory model of the ISA must be upheld.
3.2.1 Synchronization Elimination

One way to achieve sufficient synchronization is to excessively synchronize a program. For example, the class libraries which are defined as part of the Java runtime environment frequently define methods as synchronized. Synchronized methods acquire a lock on their parent object when they are entered and only release the lock when the method is exited. As a result, invocations of a synchronized method are mutually exclusive with one another and with invocations of other methods in the same object.

Excessively-synchronizing methods reduces the possibility of race conditions and makes it easier to write correct multithreaded Java programs. However, it can also introduce unnecessary synchronization overhead as discovered by Heydon and Najork while designing a high-performance Java-based web crawler [49]. For example, they noticed that using the Java class libraries to write a single line to a log file involves 67 lock acquisitions.

Two strategies to remove excessive synchronization in object-oriented languages have been proposed. The first strategy uses interprocedural escape and pointer analysis techniques to determine if an object can ever be accessed by more than a single thread. If an object can only be accessed by a single thread, then it can be allocated as a thread-local and all synchronization related to the object can be eliminated [12, 20]. This strategy is effective when static whole-program analysis can be performed, but is challenging in the context of managed runtimes, because dynamic class loading obscures interprocedural information, and in dynamic binary optimization systems.

The second strategy combines related lock synchronization by coarsening the granularity of the locks. Lock coarsening was first proposed as a static compilation technique [33] and has more recently been implemented in a dynamic compilation environment [98]. Lock coarsening can alleviate excessive locking overhead, but may reduce program scalability. A coarsened lock is held for longer and may cover previously unprotected regions of a program. Both effects may result in increased lock contention and reduce program scalability.
Coarse-grained locks are sometimes employed explicitly by programmers as another conservative strategy to achieving sufficient synchronization. Rather than locking only localized regions of a program which can incur a data race, coarse-grained locks are used to guarantee mutual exclusion of large regions of the program. It is often simpler to write a correct parallel program when using coarse-grained locks, but it may also be harder to provide good scalability. Coarse-grained locks obscure parallelism which might otherwise be available.

### 3.2.2 Synchronization Optimization

An alternative approach to reducing synchronization overheads is to make locks more efficient without eliminating the underlying locks themselves. There are two sources of inefficiency that these techniques strive to reduce: dynamic synchronization complexity and serialization bottlenecks.

In Java, synchronization operations are specified using monitors which are semantically richer than traditional locks. In addition to mutual exclusion, Java monitors also provide communication mechanisms for inter-thread waiting and notification. However, these communication mechanisms require large and complex monitor implementations which are rarely used. As a result, Bacon et al. introduced thin locks, which are only used to provide mutual exclusion for rarely contended locks, to optimize the performance of Java monitors. Furthermore, as mentioned in Section 3.2.1, many locks in Java are excessive and are never accessed by more than a single thread. In these cases, lock reservations further eliminate the need for expensive atomic acquire and release operations [61].

A hardware speculation proposal to reduce the serialization cost of synchronization has also been proposed. Rajwar and Goodman note that lock-based synchronization uses mutual exclusion to provide an illusion of atomicity and isolation. In many cases, mutual exclusion is overly conservative because concurrently executing threads affect disjoint data sets. Therefore, hardware support for speculative lock elision (SLE) enables multiple threads
to simultaneously execute code protected by the same lock by *eliding* all synchronization operations [88]. Hardware mechanisms are proposed for monitoring memory accesses and speculatively buffering all memory updates. Hardware also detects common acquire and release idioms and replaces them with a simple load to check if the lock is available. If the lock is available and the threads do not conflict, then the SLE instantaneously commits all buffered memory updates at the lock release point. Otherwise, if the lock is taken or hardware detects a memory conflict, all buffered updates will be discarded, and the locked region will be restarted (possibly acquiring the lock if conflicts continue).

The hardware provided in the atomic region abstraction enables a software-controlled implementation of SLE. This optimization enables SLE to complement the thin lock and lock reservation implementations used in modern JVM implementations by removing redundant locks and enabling optimizations across synchronization actions (see Section 6.3).

### 3.2.3 Constraints on Binary Optimization

The data-race-free memory models adopted by Java and C++ enable compiler and hardware optimizations in programs that are sufficiently synchronized. In such programs, constraints on program behavior are largely restricted to the synchronization operations themselves. Nonetheless, once a binary has been generated the memory model of the underlying ISA obscures all flexibility provided at the language level.

For example, the x86 memory model provides strict ordering guarantees with respect to loads. Load operations may not be reordered with one another and stores may not be reordered with other stores or with preceding loads. Essentially, the only reordering allowed by the x86 memory model is the reordering of loads with preceding stores [57].

To alleviate these limitations, hardware techniques are used to enable speculative re-ordering of load operations [44]. For example, modern out-of-order x86 implementations may speculatively execute a load operation before a load that precedes it in program order. The load is allowed to commit once it is the oldest remaining operation and only if no con-
Figure 3.8: **Example of optimizations disallowed by memory model constraints.** Seemingly redundant memory operations may be illegal to eliminate, depending on the memory model of the system. The depicted store removal is illegal under the x86 memory model.

Conflicting coherence requests have been received for the loaded memory address. If a conflicting access does occur, then the load must be re-executed along with all of its consumers.

Despite these techniques, other optimization opportunities remain as shown in Figure 3.8. In the example, the register $r1$ is spilled to the stack but then soon restored. In addition, the stack location used to spill $r1$ is overwritten by a later store. In a single-threaded context, it would be legal to perform the optimizations shown in Figure 3.8(b). However, when optimizing a multithreaded binary for x86 it is illegal to eliminate the seemingly dead store.

Speculative compiler optimization alone is unable to relax these constraints. The atomic execution guarantees provided by hardware enables a compiler using the atomic region abstraction to effectively manage these memory model constraints (see Chapter 8).
Chapter 4
Speculative Compiler Optimization

This chapter provides an overview of speculative compiler optimization techniques and characterizes them according to the usage models they support. Prior approaches to speculative optimization, as well as the atomic region abstraction, can be characterized according to three aspects: region shape, misspeculation recovery, and speculation strategy.

Region shape describes the types of control flow that a speculative optimization technique can encompass, ranging from a trace of basic blocks to arbitrary control flow graphs. Misspeculation recovery describes the mechanisms provided in hardware and software to enable the recovery of correct state in the event of a misspeculation. Speculation strategy describes the types of control and data speculation that are enabled by a technique.

4.1 Region Shape

This section characterizes speculative optimization approaches according to the control flow that they support. In general, the goal of each approach is to support regions which are more general than a single basic block, but are more focused than traditional global and interprocedural techniques. In doing so, profile information can be used to generate regions with sufficient scope to expose opportunity and yet enable optimizations tailored towards common program paths.

Trace scheduling was the first to introduce the notion of a compilation unit that crosses basic block boundaries and can easily incorporate profile information [41]. Assembling sequences of commonly executed basic blocks into a larger trace, provides a simple abstraction
Figure 4.1: **Example of trace scheduling bookkeeping complexity.** (a) Removing the partially redundant operations shown (b) requires code rescheduling (c) and a combination of control flow fixup and compensation code.

for a compiler to implement cross-basic block scheduling optimizations. When scheduling a trace, the compiler temporarily ignores control flow entrances and exits and therefore is unconstrained by them.

However, maintaining correctness during trace scheduling involves bookkeeping complexities that prove difficult to manage. For example, Figure 4.1 demonstrates very simple opportunities that a compiler would attempt to exploit. Though trace scheduling exposes such opportunities, exploiting them is difficult.

The superblock solved this problem by introducing a single-entry multiple-exit compilation unit, free of convergent control flow [54]. By performing tail-duplication, reconvergent control flow can be removed from traces. This trivially eliminates much of the bookkeeping complexities of traces. The hyperblock furthered the superblock by including small control flow “hammocks” and predicating them [73].

A more general shape abstraction for compiler optimization was introduced by Chambers and Ungar. Using deferred compilation, the Self dynamic optimizing compiler could arbitrarily remove program paths from an optimization region [25]. In their system uncommon paths are replaced with stubs that invoke the dynamic compiler. If invoked, a stub directs the compiler to generate code for an uncommon path that both reuses the existing stack frame and reconverges into the original optimization region.

Though effective at reducing compilation costs and enabling optimizations such as reg-
ister allocation, the deferred compilation abstraction limits other optimizations because stubbed control flow paths potentially reconverge. Whaley introduced the partial method compilation abstraction which eliminates this restriction [104]. The partial method compilation technique also replaces uncommon program paths with stubs, but these stubs are assumed to never reconverge. As a result, conventional compilation techniques can exploit some speculative redundancy opportunities. A similar abstraction is provided by the uncommon traps mechanism used in the HotSpot server compiler [85].

Hank et al. introduced an equally general compilation unit for static compilers. Their region-based compilation enables a compiler to select arbitrarily shaped pieces of a program as the unit of compilation [46], and a similar partial inlining approach has been proposed by Muth and Debray [81]. Their partial inlining technique provides much of the optimization opportunities of region-based compilation, while mitigating compilation time and code bloat issues.

Atomic regions support a similarly general region shape, but offers more powerful semantic guarantees. Similar to the partial method compilation techniques developed by Whaley, an atomic region enables a compiler to speculatively remove cold paths from an optimization region. Likewise, even if taken, these cold paths can be assumed to never reconverge.

Furthermore, atomic regions enable a compiler to make more aggressive assumptions because of hardware atomicity. Cold paths in the atomic region can be converted into asserts which roll back the entire region if triggered. This enables a compiler to simply remove all dependences related to the asserted path resulting in simplified control flow and dataflow. These simplifications expose speculative opportunities as conventional optimizations.

### 4.2 Misspeculation Recovery

This section overviews the techniques and mechanisms that have been proposed to recover from a misspeculation in speculatively optimized code. I will initially focus on software-only
techniques but will progressively introduce techniques that rely on hardware mechanisms.

A software only recovery mechanism was pioneered by Holzle et al. in their work on a dynamic optimizing compiler for the Self language. To support source-level debugging, they developed a technique for dynamically deoptimizing methods [50]. Their technique replaces the stack frame generated by an optimized program region with the stack frame expected by a programmer and transparently relocates execution from optimized code into a corresponding unoptimized code location. Holzle further generalized this technique into what is known today as on-stack replacement [51].

Some Java virtual machine implementations use on-stack replacement to provide recovery support for unexpected program behaviors [40, 85]. Doing so requires annotating any optimized code location that could require recovery as a safe point. Each safe point is treated as an opaque call which consumes all live variables and sufficiently constrains optimization to enable on-stack replacement. These constraints become increasingly common when safe points for exceptions and interrupts are considered.

These software-only approaches to recovery, though effective for other uses, overly limit speculative compiler optimization. A major concern of any speculatively optimizing compiler is guaranteeing that speculatively executed operations will not adversely affect correctness by computing incorrect values or trigger spurious exceptions. Satisfying this concern using software-only techniques requires sophisticated analysis and conservative optimization. As a result, a range of hardware proposals have been proposed to aid the compiler.

The simplest and most commonly used hardware feature might be obvious but bears mentioning: extra registers. Extra registers provide an optimizer with hidden locations in which to hold speculatively computed values without the need to spill non-speculative recovery values to memory. Extra registers are essential and an assumed component of each of the following hardware mechanisms.

Several hardware mechanisms have been proposed to support speculatively hoisting operations past branches, in other words executing operations “early”. These mechanisms
not only support improved scheduling for in-order architectures, but also support aggressive partial redundancy elimination (see Section 3.1.1 for an example).

Instruction boosting [95] supports speculatively hoisting operations above branches by enabling the compiler to communicate its decisions to the hardware. The result of a speculatively hoisted operation is labeled with the future branch outcomes upon which it is control dependent. The hardware uses this information—in conjunction with shadowed register files and store buffers—to only commit the result or any raised exception once the operation becomes non-speculative.

Sentinel scheduling [72] and write-back suppression [23] provide hardware support to simplify exception handling for speculatively hoisted instructions. In sentinel scheduling, instructions which may speculatively cause an exception are annotated and are paired with another instruction that non-speculatively checks for exceptions. With write-back suppression, all speculatively executed instructions are annotated so that if one raises an exception all speculative updates can be suppressed.

All three of these schemes specifically focus on handling speculative results and speculative exceptions but ignore other operations. For these schemes, misspeculation recovery remains a complex and non-trivial problem. When a misspeculation occurs, recovery code must be generated which reconstructs non-speculative state. However, a multitude of speculative optimizations are possible and implementing a compiler capable of generating correct recovery code for every possibility is a daunting task.

A more general misspeculation recovery mechanism can be provided by fast checkpoint hardware. A hardware checkpoint takes a logical snapshot of register and memory state that can later be restored or replaced by a subsequent checkpoint. Several proposals use checkpoint hardware for misspeculation recovery. A checkpoint is taken prior to entering

---

1The general speculation model does not require compensation code. Speculatively hoisted operations are labeled as such so that any exception is suppressed and instead the output of the operation is “poisoned” so that non-speculative consumers can raise the exception. However, the general speculation model does not support precise exceptions [93]. Therefore, general speculation violates correctness on the systems considered in this dissertation and is not further considered.
an optimization region, and the checkpoint is restored in the case of a misspeculation or exception in the optimization region. If an optimization region executes successfully, the checkpoint is discarded, typically by taking a new checkpoint.

Melvin and Patt were the first to introduce a checkpoint-based recovery mechanism for use in speculative compiler optimization. They proposed a new block-structured ISA to provide single-entry and single-exit atomic blocks [76,77]. Atomic blocks can be aggressively optimized for a predicted control flow path, and unexpected exits can be made to discard all speculative state by a hardware-supported fault assert operation. When triggered, a fault assert discards all register and memory updates performed in an atomic block and redirects control flow to an alternate implementation of the asserted code.

The rePLay framework [86] refined the block-structured ISA and introduced the frame. The frame is a single-entry and single-exit atomic trace of basic blocks in which all side exits have been converted into assert instructions. A hardware-only system, rePLay relies on a modified branch predictor to identify predictable instruction traces and place them into frames which are processed by a hardware code optimizer. These optimized frames are then stored in a frame cache, and execution is redirected to them by a frame predictor. If an assert detects a misspeculation, the frame is rolled back, and execution resumes using normal instructions.

Transmeta’s CMS was the first software-based dynamic optimization system to rely on hardware atomicity primitives for recovery [32]. Hardware atomicity enables simple recovery because if a misspeculation occurs hardware rolls back all state to a well-defined point. Execution is then simply redirected to a non-speculative implementation of the same code region (see Chapter 2. The atomic region abstraction relies on an identical checkpoint recovery model albeit with a more general shape model because CMS is designed using the superblock abstraction.
4.3 Speculation Strategy

This section focuses on the types of speculation optimizations that are easily implemented by different abstractions. In theory, each speculative optimization abstraction supports a complete range of control speculative optimizations. In practice, the choice of abstraction makes some types easier than others. Primary focus is placed on the types of control speculation that each abstraction supports. Although not a focus of this dissertation, the complementary issue of data speculation is discussed in Section 4.3.1 for completeness.

Control speculative optimizations are those that optimistically violate control dependencies as expressed by a program or binary. There are two general forms of control speculation: hoisting and sinking. Hoisting occurs when an operation is moved above a branch and occurs implicitly as part of some optimizations (e.g., partial redundancy elimination). Sinking occurs when an operation is moved below a branch and also occurs as part of some optimizations (e.g., partial dead code elimination).

The speculation strategy used in most modern compilers is based around traces, superblocks, or hyperblocks. The trace naturally encapsulates frequently executed control flow in a single compilation unit. Because a trace only includes a single control flow path, it trivially exposes speculative opportunity to a compiler. However, exploiting these opportunities requires custom compiler optimizations that maintain correctness, as discussed in Section 4.1. Even so, some hoisting opportunities cannot be exploited because of potentially excepting instructions.

By removing side entrances, the superblock and hyperblock convert some partially redundant operations into fully redundant operations enabling a compiler to trivially exploit these hoisting opportunities. Furthermore, systems designed to support the superblock and hyperblock typically include mechanisms to enable speculative hoisting of potentially excepting instructions, as discussed in Section 4.2

Using predication, the hyperblock also enables a compiler to if-convert short unbiased
paths. Predication enables otherwise control-dependent instructions to be speculatively promoted past their predicate producer by using speculative code motion techniques derived from superblocks.

The region-based [46] and partial-method [104] compilation frameworks provide alternative support for reducing cold-path optimization obstacles. In these frameworks, cold paths in an optimization region are cut and replaced by exit stubs. If a cold-path is reconvergent, then its removal may expose speculative hoisting opportunities to classical optimizations, without needing tail-duplication.

In each of these frameworks, an exit implicitly uses all live variables. As a result, sinking opportunities such as elimination of partially dead code remain obscured. Identifying and eliminating these opportunities is complex and expensive (see Section 3.1.2).

The atomic region speculation model is the first to enable a software optimizer to effectively remove exits. In the atomic region abstraction, a compiler can convert cold exits into assert operations, and, if taken, these asserted exits will trigger a hardware rollback and restart execution in an alternate implementation of the same region. Therefore, an asserted exit is not a consumer of any live variables (other than those used to compute the assert condition itself). Furthermore, an asserted exit does not impose any control dependences, which trivially enables sinking opportunities such partial dead code elimination.

In addition, an atomic region relies on hardware providing an illusion of instantaneous execution. This enables additional optimizations that would not otherwise be possible. For example, memory model constraints can be relaxed, which enables additional memory reordering and redundancy elimination. This enables new optimizations such as software-controlled speculative lock elision (see Chapter 6.3).

On the other hand, the atomic region abstraction incurs a significant performance penalty in the case of a misspeculation. Only highly biased control flow can be practically converted into assert operations. Trace-based and superblock-based speculation strategies have much lower misspeculation penalties and can therefore can exploit moderately biased control flow.
That being said, these strategies can complement one another as shown in Chapter 7.

4.3.1 Data Speculation

Although this dissertation primarily focuses on control speculation, it is important to note its relationship to data speculation. Data speculative optimizations are those that optimistically execute an operation before all of its data dependences are resolved. Of particular interest are optimizations that reorder memory operations that may alias. Note that optimizations such as redundancy or dead code elimination may implicitly reorder loads and stores.

The control speculation abstractions above all assume protection from incorrectly reordering memory aliases. Before applying aggressive optimizations, memory alias analysis must be performed to prove the correctness of optimizations that reorder memory operations. However, it is not always possible to prove that two memory accesses do not alias.

To address this limitation, Gallagher et al. introduced the memory conflict buffer which dynamically disambiguates potential memory aliases [43]. A hardware structure, the memory conflict buffer enables a software optimizer to protect the address of reordered memory operations against unexpected memory alias violations. Practical application of aggressive control speculation typically depends on hardware support for data speculation via structures like the memory conflict buffer. For example, recent processors which support superblock and hyperblock optimizations provide alias protection support in hardware [53, 64].

It should be noted that superblock-based designs are based on in-order, statically-scheduled processors. They rely on the compiler to identify instruction-level parallelism and to reorder and schedule instructions to exploit it. As a result, aggressive software scheduling is critical to achieving good performance and therefore more instruction reorderings may be necessary. These reorderings often require the aid of hardware support for data speculation.

Dynamic binary optimization also benefits from hardware support for data speculation because precise alias analysis is infeasible. Tight compilation budgets prevent the use of sophisticated analysis techniques and the lack of high-level source code information limits the
precision that can be attained. Therefore, even classical optimizations such as redundancy elimination typically require hardware support for data speculation.

Similarly, the alias imprecision inherent to type-unsafe languages such as C obscure optimization opportunities. Hardware support for data speculation can also aid in the optimization of programs written in these languages. For example, Lin et al. introduce an extended static single assignment (SSA) framework that enables control speculation and data speculation [69]. They demonstrate the usefulness of their framework by implementing SSA-based partial-redundancy elimination [63] that removes likely redundancies that are obscured by possible memory aliases. To maintain correctness, their framework presumes hardware support for data speculation. Dai et al. recently introduced a more general compilation framework which enables classical compiler optimizations to employ hardware supported data speculation [30].

However, it is unclear whether hardware support for data speculation is needed in the context of statically-typed, type-safe languages that are compiled for out-of-order processors. The type information and safety provided by these languages enables simple type-based alias analysis with sufficient precision to support classical optimization [34]. The dynamic scheduling offered by the out-of-order processor obviates the need for aggressive software scheduling.

Likewise, aggressive control speculative optimizations on these systems do not require data speculation support in order to be beneficial (as demonstrated in Section 6). Nevertheless, data speculation could expose additional opportunities and further study is warranted.
Chapter 5

Study of Control Bias in Integer Programs†

This chapter explores the feasibility for exploiting control bias as a proxy for aggressive, but safe, compiler analysis and optimizations. As suggested by the examples in Chapter 3, control-speculative optimizations—those which speculatively optimize common paths at the expense of uncommon ones—have the potential to trivially expose otherwise complex optimization opportunities. Nonetheless, the extent to which this is possible depends on the manner and frequency of biased control flow in a program.

To gain insight into these questions, the following sections analyze biased control flow behavior in the SPECint 2000 benchmarks. An abstract model of a software optimization system that speculatively exploits biased control flow behavior is also presented. Studies of this abstract model identify the mechanisms necessary to making such an optimization system practical.

5.1 Cost-benefit Tradeoff for Control Speculation

The potential for aggressive control speculation can be seen by looking at branch bias across complete program runs. In a software optimization system capable of exploiting this potential, a decision to speculate or not is made when the code is generated. In making this decision, the ratio of correct to incorrect speculations must be considered. For a branch, this ratio is the branch’s bias; for example, speculatively removing the branch in Figure 3.8 will benefit execution whenever the common path is followed but incur misspeculation costs

†The content of this chapter derives from work published at the 2005 International Symposium on Code Generation and Optimization [108].
whenever the infrequent path is executed. Speculation will improve performance whenever
the aggregate benefit exceeds the aggregate penalty:

\[
(\text{correct}\_\text{preds} \times \text{benefit}) > (\text{incorrect}\_\text{preds} \times \text{penalty})
\]

Thus speculation should be applied to all branches whose bias (or more precisely the ratio
of correct-to-incorrect speculation) exceeds the ratio of the misspeculation penalty to the
correct speculation benefit:

\[
\frac{\text{correct}\_\text{preds}}{\text{incorrect}\_\text{preds}} > \frac{\text{penalty}}{\text{benefit}}
\]

The opportunity for control speculation can be estimated by studying branch profiles
from complete program runs. For each of the SPECint 2000 benchmarks, Figure 5.1 plots
the cumulative distribution of dynamic branches that can be speculatively removed (e.g.,
assumed to follow a single path) as the overall misspeculation rate increases. Following a
curve away from the origin yields an increasing percentage of dynamic branches that could
be removed (y axis) with a resultant increase in the overall misspeculation rate (x axis). On
each curve, a circle is placed at the point corresponding to speculatively removing branches
with an average bias of 99% or greater; for example, the circle for gcc indicates that over 70
percent of dynamic branches could be eliminated with a net misspeculation rate of less than
0.1%. In general, this 99% threshold sits at or near the knee of the curve in each benchmark,
allowing correct speculation on between 25 and 90 percent (average 46 percent) of dynamic
branches with an average of about one misspeculation every 20,000 instructions. Clearly with
such misspeculation rates, even very aggressive speculation (i.e., where the misspeculation
penalty is two orders of magnitude larger than the benefit of correct speculation) can be
profitable.

While these results demonstrate significant opportunity, they are potentially optimistic.
In selecting the set of branches for speculation, the behavior of the whole program’s run
(representing future knowledge) has been used. The next section explores the effectiveness
5.2 Previous Techniques for Detecting Branch Bias

Despite significant potential, mechanisms are needed for deciding which branches can be speculatively removed. Further analysis demonstrates that this is a non-trivial issue. Specifically, two conventional techniques—using profile data from a training run and using profile data from the beginning of a run—have significant limitations.

**Profiling from a previous run:** Many aspects of program behavior are consistent from one
Table 5.1: Simulation data sets and run length. As our intention was to demonstrate the fragility of offline profiling, we attempted to find reasonable inputs whose behavior differed from the evaluation set. In some cases, we diverged from the standard SPEC training sets for profiling, which in most cases are unrealistically similar to the ref inputs. All benchmarks were compiled for the Alpha architecture using peak compiler optimization. ⋆ Since the optimization level of gcc is hard coded, we had to modify its execution to give the appearance of -O0.

<table>
<thead>
<tr>
<th>Bmark</th>
<th>Profile Input</th>
<th>Evaluation Input</th>
<th>Len</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td>input.compressed</td>
<td>input.source 10</td>
<td>19B</td>
</tr>
<tr>
<td>crafty</td>
<td>ponder=on ver 0</td>
<td>ponder=off ver 5 sd=12</td>
<td>45B</td>
</tr>
<tr>
<td>eon</td>
<td>rushmeier input</td>
<td>kajiya input</td>
<td>9B</td>
</tr>
<tr>
<td>gap</td>
<td>(test input)</td>
<td>(train input)</td>
<td>10B</td>
</tr>
<tr>
<td>gcc</td>
<td>-O0 cp-decl.i ⋆</td>
<td>-O3 integrate.i</td>
<td>13B</td>
</tr>
<tr>
<td>gzip</td>
<td>input.compressed 4</td>
<td>input.source 10</td>
<td>14B</td>
</tr>
<tr>
<td>mcf</td>
<td>(test input)</td>
<td>(train input)</td>
<td>9B</td>
</tr>
<tr>
<td>parser</td>
<td>(test input)</td>
<td>(train input)</td>
<td>13B</td>
</tr>
<tr>
<td>perl</td>
<td>scrabbl.pl</td>
<td>diffmail.pl</td>
<td>35B</td>
</tr>
<tr>
<td>twolf</td>
<td>(train input) fast 3</td>
<td>(ref input) fast 1</td>
<td>36B</td>
</tr>
<tr>
<td>vortex</td>
<td>(train input)</td>
<td>(reduced ref input)</td>
<td>32B</td>
</tr>
<tr>
<td>vpr</td>
<td>-bend_cost 2.0</td>
<td>-bend_cost 1.0</td>
<td>21B</td>
</tr>
</tbody>
</table>

If the training input differs materially from the evaluation input, the difference in program behavior can be substantial. In Figure 5.1, the benefit and misspeculation rates achieved from selecting biased branches (using a 99% threshold) from a differing input are plotted as
triangles; the set of inputs used is described in Table 5.1. For these inputs, the benefit is reduced by a factor of 3 on average and the misspeculation rate increases by a factor of 10. Using a higher threshold does not significantly reduce the misspeculation rate for some of the worst offenders (crafty, perl and vpr) and achieves only approximately 3/4ths of the benefit. The misspeculation rate can be reduced by averaging together a number of profiles; while this does reduce misspeculation rate it also reduces opportunity as input-dependent branches will no longer be considered (data not shown). Overall, this form of speculation control does not do a good job of approximating self-training, an observation also made by Wall [103].

**Profiling from initial behavior:** Another approach is to use initial branch bias behavior to predict the overall behavior of a branch. A recent study shows initial behavior is a generally more effective predictor of branch bias than having a profile from a training data set. In some programs, however, a significant number of executions need to be recorded in order to reliably predict branch behavior [105].

Since most highly-biased branches exhibit identical behavior during their entire lifetime, the bias of an initial segment of execution is an effective predictor of which branches will be highly biased. In fact, 80 percent of the benefit of self-training can be captured by choosing to speculate only on branches whose bias exceeds 99% for their first 1,000 executions. The remaining 20 percent of benefit is derived from branches that are not initially biased, but whose overall behavior is biased.

The difficulty with this approach is the same one observed by Wu et al.: some branches change their behavior—sometimes drastically so—during their execution [105]. For the benchmarks studied in this chapter, 7 percent of the static branches selected as highly biased (99% biased or greater) from their initial 1,000 executions had an average bias for the whole run that fell below 99%; more than a third of these branches had average biases less than 90%. The inclusion of these false positives results in a misspeculation rate of 2.6%; without them, the misspeculation rate is only 0.13%.
It is tempting to think that by observing a longer initial sequence before making a
decision, misspeculations can be eliminated; however this is not particularly effective. The
crosses in Figure 5.1 show the benefit/misspeculation trade-offs for 5 different training period
lengths: 1k, 10k, 100k, 300k, and 1 million executions. While increasing the initial sequence
length does reduce misspeculation rate (points farthest from the y-axis correspond to the
shorter training period), in some cases (bzip2, perl) it takes more than 300k executions
to reach a rate comparable to self-training. In one case, mcf, even 1 million executions are
insufficient, resulting in a 3% misspeculation rate. Furthermore, the cost of a longer training
period is a reduction in the achievable benefit.

The problem with both of these conventional mechanisms is that they lack robustness.
While each works well in certain circumstances, misspeculation rates as high as one per
100 instructions executed remains. Clearly such misspeculation rates are unacceptable for
aggressive control speculation, where misspeculation detection and recovery could take hun-
dreds of cycles.

This lack of robustness derives from the fact that once a decision to speculate is made it
is never reconsidered. Section 5.4 demonstrates that, by adding a small amount of reactivity,
the system can be made quite robust. The next section more closely investigates branches
that change behavior over their lifetimes.

5.3 Characterization of Changing Branches

When classifying branches from their initial bias, there are two challenging possibilities: 1)
branches that start biased but become unbiased or biased in the opposite direction, and
2) branches that are initially unbiased and later become biased. The first category is the
most serious because it represents potential misspeculations; the second category merely
represents lost opportunity and, as shown in Section 5.4, the loss is modest.

Figure 5.2 provides some insight into the difficulty of this problem; five static branches
Figure 5.2: **Five static branches with initially invariant behavior.** Branch bias averaged over blocks of 1000 dynamic instances. In all of these cases, the branch can be considered highly biased for at least the first 20,000 branch instances.

from the benchmark gap are shown that are characterized as biased for at least the initial 20,000 executions (most of which are initially 100% biased) then change their behavior, in some cases completely reversing their bias. By solely looking at a sequence of initial branch outcomes, these branches are indistinguishable from branches that remain biased throughout a program’s execution.

Manual inspection of the source code provides little insight. In some cases, the branch’s behavior is correlated to a path or calling context and the branch’s initial behavior is determined by the control flow of early executions. In other cases, no obvious correlation exists, leaving only the unsatisfactory explanation that the branch’s behavior is data dependent. In one case, the branch condition is purely a function of a loop induction variable so that it is false for the first 32,768 executions and true for the rest. No simple features appear to exist that would enable these branches to be distinguished from truly biased branches.
5.4 Requirements for Robust Control Speculation

Despite the lack of a static heuristic for classifying branches according their branch bias behavior, a robust control-speculation system can be built. This section, describes a simple model for controlling speculation that addresses the aforementioned realities of branch bias behavior. Despite its simplicity, this model is effective enough that its performance is comparable to, and occasionally exceeds, static self training (i.e., using the same input for profiling as evaluation). The model is studied in the abstract in order to explore its fundamental requirements independent of implementation.

5.4.1 A Simple Effective Model

The fundamental requirement of any robust model is an ability to tolerate branches with biases that vary over time. It is necessary for branches to be reclassified when their behavior changes. Figure 5.3(a) depicts an abstract model for the systems described in Section 5.2, which classifies branches according to earlier profile information. This model lacks robustness because there is no recourse when a branch has been incorrectly characterized.

![Finite-state machine model for branch behavior characterization](image)

Figure 5.3: A finite-state machine model for branch behavior characterization.

Figure 5.3(b) shows a model with two additional transitions, both back to the monitor state. From the biased state, the transition should be taken when the branch is resulting in an undesirable rate of misspeculations. From the unbiased state, it is merely necessary to periodically revisit the monitor state. As a following sensitivity analysis will show, the existence of these transitions is fundamental; most every other attribute of this model is of secondary importance.
Table 5.2: Model Parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monitor period</td>
<td>10,000 executions</td>
</tr>
<tr>
<td>Selection threshold</td>
<td>99.5 percent</td>
</tr>
<tr>
<td>Misspeculation threshold</td>
<td>10,000 (+50 on misp., -1 otherwise)</td>
</tr>
<tr>
<td>Wait period</td>
<td>1,000,000 executions</td>
</tr>
<tr>
<td>Oscillation threshold</td>
<td>will not optimize a sixth time</td>
</tr>
<tr>
<td>Optimization latency</td>
<td>1,000,000 instructions</td>
</tr>
</tbody>
</table>

Nevertheless, to evaluate the model, various model parameters must be assigned. Table 5.2 shows the parameters used, which roughly approximate latencies and thresholds practical for implementation.

To a large degree, the model parameters are chosen to reduce the effort required by the optimization system. In particular, every transition into or out of the biased state requires code to be re-optimized. The main drawback of using a model like Figure 5.3(b) is the potential for oscillating in and out of the biased state. Therefore, four strategies are used to dampen this oscillation:

1. First, a moderately long monitoring period (10,000 executions) provides a simple filter for reducing the number of false positives.

2. Second, hysteresis is introduced by using a stricter threshold for entry into the biased state than for eviction. For example, to target branches with average bias of greater than 99%, the bias must be greater than 99.5% to begin speculation, and branches are only evicted when their bias falls below 98% for a non-trivial time period. This is implemented in the model using a saturating counter that counts up 50 on a misspeculation and down by one on a correct speculation; the branch is evicted if the counter reaches 10,000 (requiring at least 200 misspeculations). This hysteresis is necessary to tolerate short bursts of misspeculations by otherwise biased branches.

3. Third, the model spends a relatively long waiting period (1 million executions) in the unbiased state. In addition to reducing the frequency at which a branch’s classification
needs to be reconsidered, increasing this period reduces the likelihood that a branch which is only temporarily biased will be selected for speculation.

4. Fourth, there is an absolute limit to the number of times each branch can oscillate. This is a necessity for the small number (~50 of over 7000) of branches that otherwise oscillate hundreds or thousands of times, even for relatively short program runs. After a threshold number of oscillations, a branch is permanently classified as unbiased. This limit has little impact on the number of branches that can be speculatively optimized but provides a two-thirds (on average) reduction in the number of requested re-optimizerizations.

For transitions into or out of the biased state, which are accompanied by re-optimizerization requests, the latency required to make modifications to the code is modeled. Given the abundance of thread-parallel resources in current and future processors, the model assumes that re-optimizerization is performed in parallel with execution and hence has latency, but no overhead. A latency of 1 million instructions is used (the functional simulations described below have no notion of time). Thus, after a branch has been selected for speculation, the model waits 1 million instructions before counting correct and incorrect speculations. Likewise, when a branch is evicted from the biased state, correct and incorrect speculations continue to be counted for the following 1 million instructions until the repaired code fragment takes effect. The value of this latency represents an estimate of the compilation cost of a dynamic optimizer for median-sized optimization regions (~100 instructions).

5.4.2 Reactive Model Performance

This section, demonstrates two characteristics of the model: 1) its ability to select a set of branches on which to speculate is comparable to what is achievable by self training, and 2) the model is rather forgiving in regard to its parameters—except that all of the transitions must be present.
Figure 5.4: Reactive control performs comparably with self-training. The line still represents the correct/incorrect speculation trade-off achievable through self-training. The other marks are results from the reactive control model. square: baseline, x: no eviction (without biased→monitor transition), +: no revisit (without unbiased→monitor transition), circle: eviction by bias sampling, ellipse: shorter revisit period, diamond: lower (1,000) eviction threshold, triangle: sampling. As all of the points except the x and + are collocated, the behavior of the model is primarily only sensitive to the presence of all of the transitions.

As some of the changes of program behavior are only observed in long runs of the programs, the following experiments were performed in the context of a fast functional simulator that simulates each benchmark to completion. These runs explore the behavior of a speculation control mechanism in an abstract context, independent of implementation. The behavior of each branch is tracked independently, except when modeling optimization latency.

Figure 5.4 plots the results of these simulations in the same format as Figure 5.1. For reference, the self-training line is shown. The performance of the model with the parameters shown in Table 5.2 is shown by a square dot. In all benchmark runs, the performance is
Table 5.3: Model Transition Data. Only a small fraction of branches need to be evicted from the biased state and mispredictions can be very far apart.

<table>
<thead>
<tr>
<th>Bmark</th>
<th>Static branches</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>executed</td>
<td>biased</td>
<td>evicted</td>
<td>total</td>
<td>%</td>
</tr>
<tr>
<td>bzip2</td>
<td>282</td>
<td>109</td>
<td>6</td>
<td>15</td>
<td>44.1%</td>
</tr>
<tr>
<td>crafty</td>
<td>1124</td>
<td>396</td>
<td>138</td>
<td>276</td>
<td>25.1%</td>
</tr>
<tr>
<td>eon</td>
<td>403</td>
<td>95</td>
<td>3</td>
<td>3</td>
<td>38.3%</td>
</tr>
<tr>
<td>gap</td>
<td>3011</td>
<td>1045</td>
<td>167</td>
<td>201</td>
<td>52.5%</td>
</tr>
<tr>
<td>gcc</td>
<td>7943</td>
<td>2068</td>
<td>11</td>
<td>12</td>
<td>66.3%</td>
</tr>
<tr>
<td>gzip</td>
<td>314</td>
<td>66</td>
<td>7</td>
<td>12</td>
<td>35.4%</td>
</tr>
<tr>
<td>mcf</td>
<td>366</td>
<td>210</td>
<td>22</td>
<td>47</td>
<td>33.6%</td>
</tr>
<tr>
<td>parser</td>
<td>1552</td>
<td>284</td>
<td>53</td>
<td>124</td>
<td>26.3%</td>
</tr>
<tr>
<td>perl</td>
<td>1968</td>
<td>1075</td>
<td>58</td>
<td>64</td>
<td>63.4%</td>
</tr>
<tr>
<td>twolf</td>
<td>1542</td>
<td>440</td>
<td>19</td>
<td>22</td>
<td>32.1%</td>
</tr>
<tr>
<td>vortex</td>
<td>3484</td>
<td>1671</td>
<td>67</td>
<td>104</td>
<td>88.5%</td>
</tr>
<tr>
<td>vpr</td>
<td>758</td>
<td>340</td>
<td>16</td>
<td>38</td>
<td>31.6%</td>
</tr>
<tr>
<td>ave</td>
<td>34%</td>
<td>2%</td>
<td>76</td>
<td>44.8%</td>
<td>65,000</td>
</tr>
</tbody>
</table>

competitive with self training. In gzip and mcf, the model outperforms static self training, because it can adapt to the low frequency time-varying behavior of branches; for example, the average bias of the middle branch in Figure 5.2 is about 60% so it should not be selected for speculation by a static mechanism, but a reactive model can discern that its behavior consists of two highly-biased regions, each of which can be exploited.

Table 5.3 presents results regarding how often branches transition into and out of the bias state. Of the static (conditional) branches executed, 34 percent enter the biased state some time during the benchmark run. Of these, about 7 percent—2 percent (37% × 7%) of executed static branches—are later evicted from the biased state. Some of these branches are evicted more than once; the average evicted branch transitions back to the monitor state 1.6 times (total evictions/static branches evicted). Almost half of all dynamic branches can be speculatively removed, incurring only one misspeculation every 65,000 dynamic instructions, on average.
5.4.3 Sensitivity Analysis

These above results are surprisingly insensitive to exactly how the model is implemented. Exploration of a number of configurations showed that most changes merely shift the model’s performance up or down along the self-training curve. Some of these sensitivity results are included in Figure 5.4; in many cases the points in the figure overlap, emphasizing the insensitivity:

1. **Lower Misspeculation Threshold:** Lowering this threshold from 10,000 to 1,000 makes the system less tolerant of branches with varying biases, leading to a more conservative system.

2. **Misspeculation Sampling:** Rather than tracking each branch’s misspeculation rate continuously, this experiment periodically re-samples the branch’s bias to make the eviction decision. Computing the bias of 1,000 samples every 10,000 executions (a 10% duty cycle) ends up evicting more branches resulting in a slight reduction of both correct and incorrect speculations.

3. **Sampling in “monitor” State:** Using a 1-in-8 sampling rate adds a little additional uncertainty causing a few unbiased branches to be declared biased. Larger sampling rates can be tolerated as well by lengthening the monitor period to keep the number of samples constant.

4. **More Frequent Revisit:** Lowering the revisit wait period by an order of magnitude to 100,000 executions introduces two competing factors: 1) a reduction of time spent by biased branches in the unbiased state (increased opportunity), and 2) branches that are only momentarily biased are more likely to be selected and later evicted (increased re-optimization cost).

5. **Optimization Latency:** All of the results discussed include a latency for optimizing and deploying new code. If this latency is set to zero (data not shown), the percentage
of correct speculations increases only an additional 0.1% and the number of misspeculations is reduced by a factor of 1.1. This latency tolerance arises from two factors: 1) the branch in question may not be executed again for many instructions, and 2) although the branch may not be considered highly biased, it still may be biased in the same direction; as a result only a fraction of future executions will cause misspeculations. Figure 5.5 shows the misprediction rate (fraction of branches not in the original bias direction) in the vicinity (up to 64 branches) of a transition out of a highly biased state. Over 50 percent of the static branches have a 30% or lower misspeculation rate during the transition period. It is really only the 20 percent of branches that become perfectly biased in the other direction that require quick action.

However, if the transitions back to the monitor state are removed, behavior changes significantly. If the revisit transition (unbiased→monitor) transition is eliminated, the model achieves only a little more than 80% of the correct speculations of the baseline. Removing the eviction transition (biased→monitor) increases misspeculation rate by almost two orders of magnitude. Table 5.4 relates the average benefit and misspeculation rates for each experiment.
Table 5.4: **Model Sensitivity.** Only the **no revisit** and **no eviction** configurations truly differ from the baseline.

The fact that the model is so insensitive to its parameters relaxes the demands placed on a real implementation. It means that the model can be implemented in a simplistic manner without significant impact on performance.

<table>
<thead>
<tr>
<th>configuration</th>
<th>correct</th>
<th>incorrect</th>
</tr>
</thead>
<tbody>
<tr>
<td>no revisit</td>
<td>35.8%</td>
<td>0.007%</td>
</tr>
<tr>
<td>lower eviction threshold</td>
<td>42.9%</td>
<td>0.015%</td>
</tr>
<tr>
<td>eviction by sampling</td>
<td>43.6%</td>
<td>0.021%</td>
</tr>
<tr>
<td>baseline</td>
<td>44.8%</td>
<td>0.023%</td>
</tr>
<tr>
<td>sampling in monitor</td>
<td>44.8%</td>
<td>0.025%</td>
</tr>
<tr>
<td>more frequent revisit (100k)</td>
<td>46.1%</td>
<td>0.033%</td>
</tr>
<tr>
<td>no eviction</td>
<td>53.9%</td>
<td>1.979%</td>
</tr>
</tbody>
</table>
Chapter 6

Atomic Regions for Managed Languages

This chapter focuses on common optimization opportunities in managed languages and shows that the atomic region abstraction both trivially exposes them and is easy to implement. Using two examples from the DaCapo Java benchmark suite, I will first depict some of these speculative optimization opportunities and the complexity of exploiting them without atomic regions.

I will then describe a proposed implementation of hardware atomicity for modern out-of-order processors and how it can be used to provide the necessary primitives for the atomic region abstraction. Likewise, I describe how the atomic region abstraction can easily be incorporated into a modern Java Virtual Machine (JVM) to enable aggressive speculative optimization. Finally, an evaluation of the proposed system, running Java programs from the DaCapo benchmark suite, demonstrates potential for a 10-15% average performance improvement.

6.1 Opportunities in Managed Languages

Even high quality managed language code has significant inefficiencies resulting from two sources: good software engineering practice and the safety mechanisms provided by modern languages. Good software engineering practices and an emphasis on programmer productivity demand that source code be readable, debuggable, maintainable, and reusable, which often translates to frequent control flow and many invocations of small virtual methods.

†The content of this chapter derives from work published at the 34th International Symposium on Computer Architecture [83].
Modern language safety features include performing NULL checks on dereferenced pointers, array bounds checks to catch array overruns, and checked dynamic casts to ensure type safety. While these checks rarely fail, their frequency significantly impacts the average basic block size, as observed by the compiler.

In principle, compilers can be quite effective at mitigating these inefficiencies. Much of the inefficiency results from having to recompute values and perform checks that are redundant or are subsumed by other checks. Classical optimizations such as value numbering and partial redundancy elimination address these inefficiencies when they are within an optimization region. However, because of the frequency of branches, only a fraction of the redundancy is within a single basic block, necessitating a larger optimization scope.

In general, the optimization opportunity exposed to a compiler increases as the scope of optimization is grown. In other words, optimizing across multiple basic blocks (i.e., global optimization) tends to expose more opportunities than simply optimizing within a single basic block (i.e., local optimization). Likewise, inter-procedural optimizations expose much more opportunity than intra-procedural optimizations.

Unfortunately, the software complexity necessary to exploit opportunities tends to grow along with the scope of optimization and the difficulty of identifying more complex opportunities. Global optimizations depend on and must correctly interpret information from a slew of supporting passes such as def-use, interval and alias analysis. Extending these optimizations towards more complex opportunities is often impractical, as Chapter 3 shows.

Speculative optimization compounds the problem, as it fundamentally involves identifying operations necessary only on cold paths and then moving them away from hot paths without violating correct program execution. This delicate balancing of performance and correctness concerns can quickly become untenable as optimization scope is widened and the number of paths considered increases. In contrast, hardware atomicity primitives enable atomic-region based optimization that trivially scales to large optimization scopes containing numerous paths. The reason is simple, the atomic region trivially exposes speculative
optimization opportunities to conventional compiler passes.

To help motivate this point, the remainder of this section will describe a representative optimization opportunity from the DaCapo [11] benchmark Xalan and a more sophisticated optimization opportunity taken from the DaCapo benchmark Jython. Each example illustrates the issues which confound traditional speculative optimization approaches to exploiting these opportunities.

### 6.1.1 DaCapo Xalan example

Figure 6.1(a) shows a simplified control flow graph for the most frequently invoked method in Xalan, `addElement`. The method inserts an integer into a `SuballocatedIntVector` object, which provides an efficient implementation of an extensible vector of integers. To avoid having to reallocate and copy the whole vector whenever the vector extends beyond its current allocation, the object maintains an array of integer sub-arrays so that the vector can be extended simply by allocating a new integer sub-array.

Similar to much of the managed code I analyzed, the function `addElement` has a fast hot path and a slower cold path. The fast path is invoked whenever an element is inserted...
into the same sub-array as was previously accessed (the software caches the most recent sub-array); since insertions are generally to sequential elements and the sub-arrays are large, this fast path ends up handling 99.8% of the calls. The slow path handles the rare cases when an access is performed to a segment other than the cached one, including when new segments are allocated. At the hottest call site, the function is called twice sequentially on the same object, as shown below:

```cpp
m_data.addElement(m_textPendingStart);
```
```cpp
m_data.addElement(length);
```

Inlining this method at both call sites (as shown in Figure 6.1(b)) can expose some redundancy to the compiler. Figure 6.2(a) shows the code for the hot path \( a_1 \rightarrow b_1 \rightarrow a_2 \rightarrow b_2 \). By performing superblock formation [54], which involves code replication, the compiler can remove the incoming edge \( c_1 \rightarrow a_2 \) (shown in Figure 6.1(c)), so that it can guarantee that execution of block \( b_2 \) only occurs if block \( b_1 \) was executed (i.e., \( b_1 \) dominates \( b_2 \)). This restructuring enables the compiler to trivially remove those operations from \( b_2 \) that are redundant with those in \( b_1 \) (shown in Figure 6.2(b)). One of the optimizations applied (constant propagation of the first \( ++i \)) effectively removes an instruction from the path \( a_1 \rightarrow b_1 \rightarrow a_2 \rightarrow c_2 \); to correct for this, however, the compiler must insert a compensation block, \( C \), into the control flow graph as shown in Figure 6.1(d). The block \( C \) holds the removed code as shown in Figure 6.2(c).

While these optimizations can be relatively effective, their implementation introduces a certain amount of compiler complexity. The compiler must guarantee that any exit from the hot path, however unlikely, will generate correct results. It must provide two key assurances to fulfill this guarantee. First, the compiler must ensure that sufficient program state is kept live in the hot path such that at each exit the precise program state required by the cold path can be reconstructed. Second, it must maintain mappings from the optimized hot path’s state to that of the cold path so that compensation code can be generated, for every
Figure 6.2: **Compiler-based redundancy removal.** (a) Unoptimized code after inlining. (b) Through superblock formation, the second copy of blocks a and b can be optimized knowing that the first copies will already have been executed, enabling constant folding of the first increment of i and removal of the redundant NULL check and load of the vector’s length field. (c) The constant folding of the increment to i effectively involves downward code motion of i += 1 past the branch in block a2, requiring compensation code to be inserted in block C.

exit from the hot path, to undo any hot-path specific optimizations (e.g., constant folding the increment of i in Figure 6.2).

In contrast, with a hardware atomicity primitive, the same hot path code can be generated without encountering such complexities. Figure 6.3 shows the atomic region optimization process for the same method. The compiler merely replicates the hot code for execution in an *atomic region*; the entry to the atomic region is delimited by an instruction (aregion_begin) that communicates the beginning of speculative execution to the hardware, and exits from the atomic region are delimited by an instruction (aregion_end) that instructs the hardware to commit the region’s results atomically. The compiler converts branches to cold paths into conditional abort instructions (aregion_abort); if an abort condition evaluates such that control should transfer to a cold path, the hardware rolls back to the state prior to the aregion_begin, and transfers control to the original (non-speculative) version of the code, *as if speculative execution of the hot path had never occurred.*

This example is rather simple due to the relatively small scope of optimization. The next example shows that as the scope of the optimization grows, so does the complexity disparity between traditional speculative optimization approaches and atomic region optimization.
Figure 6.3: **Atomic region optimization.** (a) Unoptimized code after inlining. (b) Through atomic region formation the hot region is wrapped with hardware atomicity primitives. After converting cold exits into assert operations, unmodified classical constant folding and redundancy eliminated optimizations are able to constant fold the first increment of $i$ and remove the dynamically redundant NULL check and load of $\text{length}$.

### 6.1.2 DaCapo Jython example

Consider the inter-procedural control flow graph of the fully-optimized code generated by a leading commercial JVM\(^1\) for the most frequently executed loop from Jython, shown schematically in Figure 6.4(a). Despite showing only commonly taken paths, significant complexity remains in the depicted loop. Few paths through this loop are ever executed yet the hottest of those paths executes 109 conditional branches and over 600 instructions. In addition, the hottest path makes twenty calls to eight different methods.

Note that the loop shown already benefits from aggressive optimization by the JVM. The JVM has correctly identified the hot paths and has aggressively inlined away method calls—unoptimized the hottest path would include eight more calls to two additional methods—but the remaining method calls have not been inlined for two reasons. First, some of the method calls are to virtual methods that are truly polymorphic despite being monomorphic along the hot path. Second, a method which is called four times during each loop iteration is not inlined because doing so would cause the static code footprint of the loop to grow to tens of

\[^1\]It was politely suggested that I refrain from naming the JVM analyzed to avoid potential legal implications. My analysis might otherwise constitute reverse engineering.
Figure 6.4: Complexity of Compiler Optimizations. Abstract inter-procedural control flow graph from Jython (only executed paths shown). (a) As optimized by a commercial JVM. (b) The hot path if optimized in isolation. (c) The control flow/call graph resulting from partial-inlining and superblock formation to optimize the hot paths. (d) The control flow/call graph using the proposed hardware support for atomic regions. Atomic regions enable the compiler to isolate the hot path from the cold paths for the purpose of optimization; if one of the compiler’s speculations should fail, state is rolled back to the beginning of the atomic region and control is transferred to a non-speculative version of the code.

thousands of instructions—the static footprint of the JVM optimized loop is already in the thousands of instructions. Similarly, the JVM has identified and eliminated many redundant operations along the hot path. Nevertheless, many dynamically redundant type and NULL checks (demanded by Java language semantics) remain.

Manual analysis of the hot path shows that aggressive speculative optimizations can eliminate more than two-thirds of the dynamic instructions (Figure 6.4(b)), which translates into both improved performance and reduced power consumption. Traditional approaches to implementing these speculative optimizations, however, come at a significant cost in complexity, because correct execution along all of the potential paths has to be preserved by the compiler. For example, achieving these benefits requires a correct implementation of partial inlining—in which only hot paths are inlined to alleviate static code bloat issues. When discussing the correctness of their partial inliner implementation (which did not use
atomic regions), Muth and Debray remarked [81], “The flow of control in the program resulting from partial inlining is sufficiently complex that it is no longer obvious that the resulting program is semantically equivalent to the original.”

Figure 6.4(c) shows the simplest possible control flow graph—having aggressively eliminated 67 branches with redundant conditions—that achieves the desired optimization of the hot path. Because of the difficulty of verifying the correctness of these radical program transformations, many commercial systems do not perform speculative optimizations to this extent.

The atomic region abstraction enables a compiler to achieve the desired optimization with trivial complexity, as shown in Figure 6.4(d). The reason is simple, hardware atomicity enables a software compiler to enter an intuitive optimization contract with the hardware. The compiler identifies an optimization region and the hardware in turn guarantees to either execute the region in its entirety or not at all. This, in turn, enables the compiler to optimize the hot program in isolation after simply replacing the cold paths with conditional abort instructions that verify a cold path would not have been taken. In the infrequent event that a cold path is traversed, a conditional abort will be triggered and hardware will roll back execution to the state prior to entering the atomic region. Hardware also transfers control to a compiler-specified alternate version of the code that includes the eliminated cold paths.

Replacing cold paths with conditional asserts also eliminates all the dataflow dependences from the cold path. This trivially enables classical compiler optimizations to exploit speculative optimization opportunity without being rewritten. In addition, the atomic region abstraction simplifies the implementation of new optimizations. For example I was able to produce a working implementation of partial inlining in six hours, a feat I would dare not attempt without hardware atomicity primitives.
6.2 Providing Hardware Atomicity

This section proposes an implementation for hardware atomicity which is synergistic with other recent microarchitecture proposals. It satisfies the requirements described in Chapter 2.1 while also being compatible with the out-of-order microarchitecture used as a baseline. The implementation of hardware atomicity is similar to prior hardware checkpointing proposals for the implementation of resource-efficient high-performance processors.

6.2.1 Checkpoints and Hardware Atomicity

Modern processors employ speculative execution and typically record information at a fine granularity for when speculation fails and execution state needs to be restored. However, speculation mostly succeeds, and the recorded information is not frequently needed. Checkpoint processors use this observation to optimize recovery information management [4, 29, 55, 75]. They record recovery state at coarse intervals (100s of instructions) instead of at every instruction. When a misspeculation does occur, the processor restores the checkpoint and restarts execution, adaptively tracking information at a finer granularity after a misspeculation. This checkpoint abstraction obviates much of the fine-grain bookkeeping, since execution can always restore to a safe point.

An extension of the checkpoint abstraction provides hardware atomicity, namely ensuring that memory updates also appear to occur atomically. Hardware provides atomicity for a sequence of instructions by ensuring that either all the instructions appear to be committed at the same time or that none are. Specifically, the memory operations performed by an atomic region appear to occur instantaneously, with all other memory operations in the system appearing to occur either before or after.

Checkpoint processors do not otherwise provide the appearance of instanteity required by hardware atomicity. Previous proposals generally provide an execution that satisfies the underlying memory model, the requirements of which may be weaker than atomicity.
One possible implementation that provides hardware atomicity involves the following steps: 1) creating a register checkpoint at the recovery point, 2) tracking all memory addresses accessed by the instructions, 3) buffering all updates performed by the instructions, 4) using an ownership-based cache coherence protocol to detect conflicting accesses from other agents, 5) discarding updates on a conflict, and 6) committing the updates in the cache atomically. These requirements are similar to those for speculative lock elision (SLE) [88].

A practical implementation of hardware atomicity should expose a simple set of primitives to software. The proposed implementation uses exactly the instruction set primitives proposed in Chapter 2.1.

### 6.2.2 Microarchitectural Implications

A critical aspect of hardware atomicity is its synergy with recent microarchitectural proposals. This is important since any proposal for exposing hardware mechanisms to software must also be amenable to high-performance implementation. A simple abstraction also provides significant flexibility to hardware designers. The atomic region abstraction allows hardware to execute the code region in whatever way seems fit, as long as when an abort condition occurs, the execution restores to the beginning of the region with appropriate information in the appropriate registers.

The proposed implementation of hardware atomicity leverages similarities with a checkpoint processor. The common and fast path execution for hardware atomicity must be fast and have low overhead, and these requirements can be satisfied by a checkpoint processor with atomicity support.

Various implementation strategies based on checkpoint architectures exist for providing the memory requirements of hardware atomicity. In this implementation, the data cache retains the data footprint of the atomic region and a register rename table checkpoint is used for recovering register state. Each cache line is extended with two bits for tracking which addresses have been read and written in the atomic region. These addresses are
exposed to the coherency mechanism to observe invalidations. Flash clear operations are used to commit and/or abort speculative state.

While support for hardware atomicity may appear similar to hardware support for transactional memory [67], significant differences in requirements and usage exist, resulting in different hardware implementation requirements. Transactional memory is proposed primarily for scalability and can potentially tolerate some loss of single-thread performance to achieve this scalability. In contrast, the atomic region abstraction uses hardware atomicity to improve single-thread performance, and any hardware execution overheads will reduce the benefit of these optimizations. Section 5 discusses the implications of simplified implementations on the usage model. Use of a checkpoint execution substrate for implementing hardware atomicity enables a nearly no-overhead common case execution and permits multiple atomic regions to be in-flight simultaneously.

Apart from the performance goal of fast common case execution, the atomic region abstraction simplifies the functionality required from the hardware implementation. Since the hardware is used opportunistically to improve the performance of a single thread, a best effort implementation is sufficient.

6.3 Forming and optimizing regions

This section focuses on how the compiler uses the atomic region abstraction to generate better code. Specifically, it demonstrates how support for atomic regions can be introduced into a compiler without significant changes, an algorithm for selecting appropriate atomic regions while achieving good program coverage, why assertions constrain optimization significantly less than branches, and optimizations enabled by atomic regions.

Atomic regions and abort as try/catch: Modern languages like Java generally provide support for structured exception handling, which in Java takes the form of try and catch blocks. These primitives enable the programmer to specify one block of code that should be
executed assuming that no exceptions occur and another one to be executed on an exception. To support these language features, a compiler must be able to represent them in its intermediate representation (IR).

One of the most important observations made in this work is that the IR support for try and catch is similar to what is required to represent both atomic regions and the abort path to non-speculative recovery code. This observation reduces the problem of supporting software speculation within the compiler to that of simply transforming the program’s control flow graph so that atomic regions look like try blocks and non-speculative recovery code looks like a catch block. As a result, unmodified optimizations can exploit the speculative optimization opportunities exposed by the atomic region. The entire implementation of the atomic region abstraction (including all transformations and optimizations) required approximately 3,000 lines of code (LOC) (~3% of the optimizing compiler), roughly two-thirds of which is the region selection algorithm. While the complexity of atomic region formation corresponds closely that of reported by Hwu et al. for superblock formation (2,000 LOC), their superblock optimizations incurred an additional 12,000 LOC [54].

Region formation: In selecting regions for optimization, three properties are maintained: 1) overly large regions must be avoided, 2) atomic regions must not be nested, and 3) atomic regions will be single-entry, multiple-exit subgraphs, containing arbitrary intraprocedural control flow. The first property permits a best-effort implementation of atomicity (i.e., atomic regions that overflow the cache or receive an interrupt will abort) and bounds the lost effort when a region aborts. Nesting is avoided, in part, to demonstrate that its support is not a hardware requirement. In addition, nesting only occurs as a result of encapsulating a non-inlined call within an atomic region, and I have yet to observe a case where this will significantly improve optimization. The last property simplifies region formation by building upon other well understood single-entry techniques, without the control flow limitations imposed by building regions from traces [37, 54, 86].

The process of region formation is fundamentally a profile-driven one. The goal is to select
regions for optimization that exclude infrequently executed (or “cold”) code paths. As is
typically done in JVMs, the first-pass compiler inserts instrumentation to profile program
behaviors (e.g., branches, virtual calls). For this implementation cold paths are those with
a branch bias of less than 1%; these paths will be removed from atomic regions. All other
paths are non-cold.

The region formation algorithm has five steps:

Step 1. Aggressively inline methods
Step 2. Select region boundaries (See Algorithm 6.1)
Step 3. Replicate flowgraphs for selected regions
Step 4. Convert cold edges into asserts
Step 5. Remove all inlined methods from non-speculative paths

The first step enlarges the optimization scope by aggressively inlining methods. This
can be done without fear of the “code bloat” typically associated with inlining for two
reasons. First an inlined method along a speculative path will only be retained if it is con-
tained entirely within an atomic region (inlined methods that do not satisfy this criteria are
pruned away). Second, the remaining inlined methods will have their infrequently executed
paths speculatively removed, enabling the retained paths to be further reduced in size by
optimization.

The next step, selection of region boundaries is the crux of region formation and is
detailed in Algorithm 6.1; here I overview its operation. The goal of boundary selection is
to identify a set of blocks that will become the entry and exit points for atomic regions. To
be more precise, it focuses on identifying blocks that should become atomic region entries.
The placement of atomic region exits is largely born of necessity (i.e., atomic regions may
be terminated to maintain the three previously mentioned invariants).

Placement of atomic region boundaries starts by considering loops to decide whether
individual loop iterations should be executed in atomic regions or whether the whole loop
Algorithm 6.1 Selection of atomic region boundaries

procedure SELECTBOUNDARIES(method)
    \textit{selectedBoundaries} $\leftarrow \emptyset$ \hfill $\triangleright$ Set of blocks

    // Place region boundaries at the headers of large loops (i.e. those with long iterations
    // or high trip counts) and loops containing calls reachable along non-cold paths
    \textit{L} $\leftarrow$ \textsc{LoopsInPostOrder(method)} \hfill $\triangleright$ Process loops from innermost to outermost
    \textbf{foreach} \textit{loop} in \textit{L} \textbf{do}
        \textit{loopBlocks} $\leftarrow$ \textsc{GetBlocks}(\textit{loop})
        \textit{loopHeader} $\leftarrow$ \textsc{GetLoopHeaderBlock}(\textit{loop})
        \textit{hasWarmCall} $\leftarrow$ \textsc{HasCallOnWarmPath}(\textit{loopHeader}, \textit{loopBlocks})
        \textit{loopPreHeader} $\leftarrow$ \textsc{GetLoopPreHeaderBlock}(\textit{loop})
        \hfill // \textsc{LoopWeight} defined in Algorithm 6.2
        \textit{loopPathLength} $\leftarrow$ \textsc{LoopWeight}(\textit{loop}) / \textsc{GetExecCount}(\textit{loopPreHeader})
        \textbf{if} (\textit{loopPathLength} $\geq$ \textit{LoopPathThreshold}) \textbf{or} \textit{hasWarmCall} \textbf{then}
            \textit{selectedBoundaries} $\leftarrow$ \textit{selectedBoundaries} $\cup$ \{\textit{loopHeader}\}

    // Prune inlined methods that contain selected loops or calls reachable along non-cold paths.
    // This limits unnecessary code bloat and is part of my partial inlining implementation
    \textbf{foreach} \textit{inlinedMethod} in \textsc{INLINEDMETHODS(method)} \textbf{do}
        \textit{inlinedBlocks} $\leftarrow$ \textsc{GetBlocks}(\textit{inlinedMethod})
        \textit{inlinedEntry} $\leftarrow$ \textsc{GetEntryBlock}(\textit{inlinedMethod})
        \textit{inlinedContinuation} $\leftarrow$ \textsc{GetContinuationBlock}(\textit{inlinedMethod})
        \textit{hasWarmCall} $\leftarrow$ \textsc{HasCallOnWarmPath}(\textit{inlinedEntry}, \textit{inlinedBlocks})
        \textit{hasSelectedLoop} $\leftarrow$ (\textit{selectedBoundaries} \cap \textit{inlinedBlocks}) $\neq$ \emptyset
        \textbf{if} \textit{hasWarmCall} \textbf{or} \textit{hasSelectedLoop} \textbf{then}
            \textsc{UnInlineMethod}(\textit{inlinedMethod})

    // Place region boundaries along acyclic paths
    \textit{visited} $\leftarrow$ \emptyset
    \textit{traceBoundaries} $\leftarrow$ \textsc{GetEntryBlock}(\textit{method}) $\cup$ \textsc{GetExitBlock}(\textit{method})
    \textit{traceBoundaries} $\leftarrow$ \textit{traceBoundaries} $\cup$ \textsc{GetCallBlocks}(\textit{method})
    \textit{maxBlockExecCount} $\leftarrow$ \textsc{GetMaxBlockExecCount}(\textit{method})
    \textit{B} $\leftarrow$ \textsc{BlocksSortedByExecCount}(\textit{method}) \hfill $\triangleright$ Process by block execution frequency
    \textbf{foreach} \textit{block} in \textit{B} \textbf{do}
        \textbf{if} \textit{block} $\notin$ \textit{visited} \textbf{and} \textsc{GetExecCount}(\textit{block}) $\geq$ (\textit{maxBlockExecCount}/100) \textbf{then}
            \textit{traceBoundaries} $\leftarrow$ \textit{traceBoundaries} $\cup$ \textit{selectedBoundaries}
            \hfill // \textsc{TraceDominantPath} defined in Algorithm 6.2
            \textit{dominantPath} $\leftarrow$ \textsc{TraceDominantPath}(\textit{block}, \textit{traceBoundaries})
            \hfill // Selects boundaries that minimize Equation 6.1
            \textit{acyclicBoundaries} $\leftarrow$ \textsc{SelectAcyclicBoundaries}(\textit{dominantPath})
            \textit{selectedBoundaries} $\leftarrow$ \textit{selectedBoundaries} $\cup$ \textit{acyclicBoundaries}
            \textit{visited} $\leftarrow$ \textit{visited} $\cup$ \textit{dominantPath}

    \textbf{return} \textit{selectedBoundaries}
Algorithm 6.2 Used during selection of atomic region boundaries

// Generate an ordered list containing the most frequently executed // path through the specified block. Stop tracing at selected // boundaries and trace boundaries

procedure TraceDominantPath(seedBlock, traceBoundaries)
    dominantPath ← [seedBlock]
    traceBlock ← seedBlock; done ← false
    while ¬done do
        traceBlock ← GetDominantOutEdge(traceBlock)
        dominantPath ← dominantPath + [traceBlock]
        if traceBlock ∈ traceBoundaries then
            done ← true
    
    traceBlock ← seedBlock; done ← false
    while ¬done do
        traceBlock ← GetDominantInEdge(traceBlock)
        dominantPath ← [traceBlock] + dominantPath
        if traceBlock ∈ traceBoundaries then
            done ← true
    
    return dominantPath

procedure HasCallOnWarmPath(entryBlock, searchScope)
    worklist ← {entryBlock}
    while worklist ≠ ∅ do
        currBlock ← GetElement(worklist)
        worklist ← worklist − {currBlock}
        if HasCall(currBlock) then
            return true
        foreach edge in GetOutEdges(currBlock) do
            if GetProb(edge) ≥ coldThreshold then
                successorBlock ← GetTarget(edge)
                if successorBlock ∈ searchScope then
                    worklist ← worklist ∪ {successorBlock}
    
    return false

procedure LoopWeight(loop)
    weight ← 0
    foreach block in GetBlocks(loop) do
        blockExecCount ← GetExecCount(block)
        numBlockOps ← GetNumOperations(block)
        weight ← weight + (blockExecCount * numBlockOps)
    
    return weight
should be encapsulated within a single atomic region. There are two factors which influence the decision: the dynamic path length through the loop and whether the loop contains a call statement which will not be inlined or speculatively removed. The algorithm chooses per-iteration atomic regions when loop iterations are large or if the average number of iterations executed is high enough that the region might overflow the cache. Because atomic regions are terminated at non-inline calls, to avoid nesting, and a new atomic region often begins immediately after a call return, an atomic region boundary is also inserted in the header of any loop containing a call, to prevent the creation of irreducible flowgraphs [47]. Note that calls which are inlined or which will be speculatively removed are ignored.

Next, the region selection un-inlines any aggressively inlined methods that will not be completely encapsulated in an atomic region; this step prevents code bloat resulting from the method needing to be fully duplicated on an atomic region’s non-speculative path. If an aggressively inlined method includes an atomic region boundary (from the previous step) or a call statement which will not be speculatively removed, it is un-inlined.

The last part of the boundary selection algorithm places boundaries along acyclic paths. The algorithm iteratively selects the hottest block that has not already been visited and traces the dominant path through the block, terminating the trace at already selected region boundaries or at the CFG entry and exit or any call continuation. All loop pre-headers and loop exits contained on the dominant path, as well as the start and end of the path, become candidates for boundary selection. The algorithm selects the subset of the candidate boundaries that minimizes $\Pi$ in Equation 6.1 where $R$ is the desired region size and $r_n$ is the size of the $n^{th}$ candidate region. Equation 6.1 increases in value for regions further from the desired size but is also biased towards selecting oversized regions by the $r_n$ term in the denominator. This equation was originally proposed for the task selection algorithm of master/slave speculative parallelization [107].
Once region boundaries have been selected, Step 3 creates atomic regions by performing a depth first search, ignoring cold paths, starting from each selected region boundary and stopping at other selected region boundaries, the CFG exit, and any call statements. The visited blocks are copied, an aregion_begin is placed at the entry of the duplicated region, and an aregion_end is placed at each region exit. All edges into the original region entry block are redirected to the aregion_begin, and an exception edge is added from the aregion_begin to the original region entry block. Figure 6.5(b) shows the result of this step (partial loop unrolling has also been applied to the outer loop).

The remaining steps of atomic region formation convert cold branches into asserts (Step 4) and replace aggressively inlined methods on non-speculative paths with call statements (Step 5).
As initially stated, region formation should avoid generating large atomic regions. I found that setting `LOOPPATHThreshold` and $R$ to a value of 200 high-level intermediate representation operations\(^2\) satisfies this property without sacrificing much opportunity.

**Why asserts constrain optimization less than branches do:** One of the final steps of region formation converts branches from the hot path to the cold paths into assertions in the compiler’s intermediate representation. These assertions constrain optimizations less than branches, because the assertion operations are implemented in the high-level IR as simple operations that have only source operands and no side effects. Like an ALU operation that produces no value and unlike branches, an assert can be completely ignored when optimizing other data-independent instructions. Furthermore, asserts can be optimized by existing passes: they can be freely scheduled across branches, limited only by their data dependences and the boundaries of the atomic region, and redundant asserts are eliminated by existing redundancy elimination passes such as global value numbering. Only dead code elimination needs a slight modification to consider asserts essential so that they will not be removed despite having no dataflow consumers.

**Atomic regions enable optimizations:** The guarantees provided by atomic regions enabled us to implement several additional optimizations: partial inlining, partial loop unrolling and speculative lock elision\(^3\) [88]. The implementations of partial inlining and partial loop unrolling were enabled by the design simplicity offered by atomic execution, and speculative lock elision was enabled by the atomicity and isolation guarantees provided by hardware. The relatively small amount of code required to implement these optimizations (∼200 LOC each for partial inlining and partial loop unrolling, and ∼400 LOC for speculative lock elision) demonstrates the simplicity offered by atomic regions.

Partial inlining exposes additional opportunity by enlarging the optimization scope, but

---

\(^2\)There is a loose correspondence between IR operations and the number of hardware instructions actually generated.

\(^3\)SLE is used to reduce monitor overhead, but this optimization would also reduce monitor-induced serialization in multithreaded workloads.
limits static code expansion by obviating the need to inline infrequently executed paths in
the method. Partial loop unrolling has similar benefits. However, implementing either opti-
mization without atomic regions overly burdens the compiler writer with the responsibility
of guaranteeing that correct program state can be recovered and forward progress made if
an infrequent path is executed. With atomic region support, the implementation of both
partial inlining and loop unrolling is greatly simplified. The hot paths of inlined methods
and loops are simply wrapped in atomic regions and the infrequent paths are converted into
assertions. If an infrequent path is executed, an assert will fire and hardware will redirect
execution to the corresponding non-speculative code, which has not been inlined or unrolled.

Speculative lock elision (SLE) exploits opportunity exposed by my atomic region forma-
tion. Atomic regions often contain balanced pairs of Java monitor enter and exit operations,
and these monitors are typically uncontended. The JVM used already provides fast-path im-
plementations for common lock behaviors using reservation locks [61], but even the fastest
path must still check the status of the lock and update it with a store (both at monitor
entry and monitor exit) to track lock nesting depth. This monitor overhead can be elimi-
nated with atomic regions; when a balanced pair of monitor operations is contained within
an atomic region, my implementation of SLE must only load the value of the lock upon
monitor entry and verify—a compare and branch—that it is not held by another thread.
In the common case, no action is needed at the monitor exit. This improvement to single-
thread performance is in addition to any concurrency benefits from optimistically executing
a synchronized method/block.

6.4 Experimental method

Evaluating the performance impact of run-time compiler enhancements using new hardware
features presents a number of challenges. First, in the absence of real hardware, a full-system
simulator is a necessity, as a JVM and some of the workloads are multithreaded and use
many system features. Second, because the compilation is performed during the program run, the benchmark runs have to be sufficiently long for the staged optimizer to produce the fully optimized code. Third, in order to compare the performance of two different compilers it is necessary to select equivalent regions of the program’s execution to make an “apples-to-apples” comparison. Figure 6.6 depicts the evaluation infrastructure, which was developed to overcome these obstacles.

The benchmark being evaluated is executed using a modified version of the Apache Harmony Dynamic Runtime Layer Virtual Machine (DRLVM) for Java [5] on the SoftSDV full-system simulator [101], which has been extended to support the ISA extensions for atomic regions⁴ discussed in Section 6.2. I use the DRLVM server execution manager configuration to maximize code quality; the whole process is completely automatic and profile driven. A functional simulation is run for a sufficiently long duration to allow for all initial compilation to be performed and for the staged optimizer to generate fully optimized code for commonly executed methods.

Once a representative portion of the execution is reached, the state of the functional simulation is recorded for use in a timing simulation. The format of the state recorded, known as a long instruction trace (LIT) [94], contains a snapshot of the initial processor

⁴For debugging the compiler, I also developed a means to test on real machines by registering a signal handler for invalid opcode exceptions (triggered by the unrecognized aregion_begin instructions) that inspects the faulting instruction and branches immediately to the (non-speculative) recovery path.
<table>
<thead>
<tr>
<th>Processor frequency</th>
<th>4.0 GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rename/issue/retire width</td>
<td>4/4/4</td>
</tr>
<tr>
<td>Branch mispred. penalty</td>
<td>20 cycles</td>
</tr>
<tr>
<td>Instruction window size</td>
<td>128</td>
</tr>
<tr>
<td>Scheduling window size</td>
<td>64</td>
</tr>
<tr>
<td>Load/store buffer sizes</td>
<td>60/40</td>
</tr>
<tr>
<td>Functional units</td>
<td>Pentium® 4 equivalent,</td>
</tr>
<tr>
<td>Branch predictor</td>
<td>combine: 64K gshare/16K bimod</td>
</tr>
<tr>
<td>Hardware data prefetcher</td>
<td>Stream-based (16 streams)</td>
</tr>
<tr>
<td>Trace Cache</td>
<td>64 K-uops, 8-way</td>
</tr>
<tr>
<td>I-TLB</td>
<td>128 entries</td>
</tr>
<tr>
<td>D-TLB</td>
<td>64 entries, 4-way</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>32 KB, 4-way, 4 cycle hit, 64B line</td>
</tr>
<tr>
<td>L2 Unified Cache</td>
<td>4 MB, 8-way, 20 cycle hit, 64B line</td>
</tr>
<tr>
<td>L1/L2 Line size</td>
<td>64-bytes</td>
</tr>
<tr>
<td>Memory latency</td>
<td>100 ns</td>
</tr>
</tbody>
</table>

Table 6.1: **Baseline processor parameters**

architectural state and memory as well as a trace of all system interrupts necessary to simulate system events such as DMA traffic. The LIT is consumed by a detailed execution-driven simulator coupled to a micro-operation (uop) level x86 architecture model\(^5\). This simulator accurately models a detailed memory subsystem, wrong path execution, interrupts, system interactions, and DMA events. The baseline **4-wide OOO** (out-of-order) processor parameters are shown in Table 6.1. A checkpoint execution substrate, similar to that of checkpoint processors, provides atomic execution.

Because different compilation approaches are being compared, equivalent regions of the benchmark must be selected for evaluation. To accomplish this, the compiler has been modified to insert special markers that are understood by the full-system simulator. These markers bound equal work at the Java bytecode level, thus allowing for a fair comparison. Selecting good marker locations first requires collecting a complete trace of method invocations from the benchmark’s execution. This trace can then be divided into groups of 10,000 methods and run through the SimPoint 3.0 phase classification tool [45] to identify phases.

\(^5\)This work was performed while on an internship at Intel and used a proprietary simulator.
Table 6.2: **DaCapo benchmarks used in evaluation.** # = number of samples used in evaluation.

For up to four phases per benchmark, a *marker method* is also identified that can be used to bound a simulation sample and that is infrequently invoked (so that it minimally perturbs the execution). Three dynamic invocations of this marker method are used to identify the sample: i) the beginning of the warm-up period, ii) the end of warm-up/the beginning of timing simulation, and iii) the end of timing simulation. This method shares similarities with concurrent work [87].

When the JVM is invoked it is also passed a marker method identifier—the class name, method name, and call signature—which compares it to each method compiled and then inserts the marker into the matching method’s prologue. While the exact number varies, warm-up and simulation intervals are selected to contain on the order of millions to tens of millions of instructions. For benchmarks with multiple phases, results are weighted for each sample by its phase’s contribution to the overall execution. Note that non-deterministic benchmarks are not supported because the number of marker method invocations can change for different compiler configurations.

The DaCapo benchmark suite [11] is used for the evaluation (version dacapo-2006-10). The suite is intended for evaluation of JVMs by the programming languages, memory management, and computer architecture communities and consists of a set of open source, real world applications with non-trivial memory working sets. Table 6.2 lists the benchmarks used and their descriptions. The remaining benchmarks are not included due to experimen-
tal method limitations: chart and eclipse were too long running, luindex's samples could not be validated in time, and lusearch is non-deterministic. To avoid non-determinism in xalan, the single-threaded version from the beta-2006-08 release of the benchmarks is used. To work around a bug in DRLVM, the version of jython is from the same beta release.

6.5 Results

The following analysis focuses on two metrics: performance and dynamic micro-operation (uop) count reduction. The performance metric simply compares execution time of the sampled regions. Reduction in dynamic uop counts is also measured because, in general, it will translate into improved energy efficiency. Fewer uops flowing down the pipeline results in less switching activity, which in turn results in a reduction in the amount of energy consumed to perform a given unit of program work.

First, two compiler configurations are compared: no-atomic, a baseline set of optimizations that corresponds closely to Harmony’s default server configuration, and atomic, which exploits hardware-supported atomic regions. The optimization passes enabled are the same for both, except atomic performs atomic region formation, partial inlining, partial loop unrolling, and speculative lock elision.

As shown in Figure 6.7, these optimizations enable a significant (12% average) speedup across the evaluated benchmarks. Furthermore, these speedups are accompanied by a nearly comparable reduction in the number of uops retired, as shown in Figure 6.8. By providing a simple recovery abstraction (atomic regions) the hardware has facilitated the compiler’s generation of higher performance and more efficient code.

Hand inspection of the generated atomic regions uncovered clear evidence of speculative optimization, even though the compiler does not explicitly implement any. For example, in one atomic region elimination of cold paths enabled the compiler to simplify an indirect branch to a conditional branch (because 7 of the 9 possible cases were cold and converted into
asserts), eliminate branches via constant propagation previously inhibited by cold control flow, eliminate partially redundant loads, and eliminate partially redundant checks.

Some of the benefits in other regions, however, were merely the result of increasing the scope of optimization through partial inlining and partial loop unrolling beyond what the baseline inliner and loop unroller were exposing. In order to demonstrate that this scope enlargement is not responsible for all of the benefits, two additional configurations are compared: no-atomic and atomic further configured with unrealistically large inlining thresholds (a factor of five larger than the baseline), which should achieve the optimization potential resulting simply from scope enlargement. The results for these configurations are also shown in Figures 6.7 and 6.8. Note that both partial inlining and partial loop unrolling are disabled in the atomic+aggressive inlining configuration.

From this data it can be seen that the atomic region-based optimizations are achieving more than just scope enlargement, as the performance from no-atomic+aggressive inlining is less than half of the atomic+aggressive inlining case. Actually, increasing the optimization scope appears to disproportionately benefit the atomic+aggressive inlining case, as its speedup (25.3%) is more than the sum of those of atomic and no-atomic+aggressive inlining (12.2% and 7.5%, respectively).
Figure 6.8: **Micro-operation (uop) reduction.** All runs use the same hardware configuration, and differences result from increased optimization effectiveness. The second set of bars for *jython* demonstrate a benefit of more optimistic region formation (Section 6.5.1).

### 6.5.1 Understanding the Variation

Clearly, atomic regions do not uniformly benefit all of the benchmarks; the speedup achieved by the **atomic+aggressive inlining** configuration ranges from 56% (*hsqldb*) to 2% (*pmd*). This section explores the sources of this variation.

Across the benchmarks, a strong correlation exists between uop reduction and speedup, which is not surprising as both generally occur when code is optimized more effectively. It should be noted that the observed uop reduction steps not only from the removal of instructions (as is done in SlipStream [99]); in many cases uops have been replaced with other, simpler uops (*e.g.*, SLE replaces compare-and-swap primitives and Java monitor updates with a load and a branch) and the critical path through the code has been shortened. As a result many of the benchmarks exhibit superlinear speedups relative to their uop reduction.

The biggest factor affecting the degree of optimization seems to be coverage. Table 6.3 shows that four of the benchmarks with high speedups—*bloat, hsqldb, jython*, and *xalan*—execute most (upwards of 69%) of their uops in atomic regions. As I am reporting coverage **after** optimization and most of the reduction in dynamic uop count occurs in the atomic regions, an even larger fraction of the program is actually encapsulated in atomic regions.
Table 6.3: **Atomic region statistics.** coverage: fraction of executed uops in atomic regions, unique: average number of unique atomic regions in execution sample(s), size: average size of atomic regions (in dynamic instructions), abort %: percentage of regions aborting, aborts/1k uop: number of aborts per 1,000 uops. Data shown for the atomic+aggressive inlining configuration.

<table>
<thead>
<tr>
<th>Bench.</th>
<th>Atomic Regions</th>
<th>Region Abort Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>coverage</td>
<td>unique</td>
</tr>
<tr>
<td>antlr</td>
<td>9%</td>
<td>96</td>
</tr>
<tr>
<td>bloat</td>
<td>69%</td>
<td>93</td>
</tr>
<tr>
<td>fop</td>
<td>20%</td>
<td>73</td>
</tr>
<tr>
<td>hsqldb</td>
<td>76%</td>
<td>75</td>
</tr>
<tr>
<td>jython</td>
<td>87%</td>
<td>14</td>
</tr>
<tr>
<td>pmd</td>
<td>32%</td>
<td>32</td>
</tr>
<tr>
<td>xalan</td>
<td>78%</td>
<td>37</td>
</tr>
</tbody>
</table>

than these coverage numbers suggest; this effect also explains how antlr can achieve a 17% uop reduction with only 9% coverage. These four benchmarks also have the largest atomic regions. With average region sizes ranging from 75 to 225 instructions after optimization there is significant scope for optimization.

The outlier from this trend is antlr, which manages to achieve significant speedups despite low coverage. This occurs because a large fraction of the instructions are eliminated from the atomic regions it does form. On average, two-thirds of the instructions in antlr’s atomic regions get optimized away. Like the other benchmarks that get significant speedups antlr gets most of its benefits from two main sources: partial redundancy elimination and elimination of monitor overhead in the Java class library.

The pmd benchmark actually slows down in the atomic configuration because it has relatively low coverage and yet incurs a 2.2% abort rate for its atomic regions. This relatively high abort rate is the result of a behavioral change in four atomic regions that occurs between when profiling occurs and when the optimized code is deployed and an execution sample is taken. Frequent misspeculations result because a path that initially appears cold is removed from its atomic region but is later frequently executed. As described in Chapter 5, reacting to these behavior changes through adaptive recompilation can eliminate their negative
performance impacts. In Chapter 7, I describe an implementation of a reactive mechanism which is sufficient to address these behavior changes.

Two other benchmarks—hsqldb and bloat—also have non-trivial abort rates, but achieve significant speedups despite them. In hsqldb, the aborts occur very early in the atomic region so they have little negative impact beyond a pipeline flush. In bloat, they do have a large impact; almost all of bloat’s aborts occur in one of its four execution samples—the one representing the least dominant phase—and that sample incurs a 33% slowdown. Discounting that phase, bloat’s speedup would be 40% (up from 32%) for the atomic+aggressive inlining configuration.

Despite performing well in the atomic+aggressive inlining configuration, the jython benchmark incurs a slowdown in the atomic configuration. The source of this discrepancy is an important method getitem (called four times in a hot loop) that is not being inlined by the partial inliner in the atomic configuration. This results in a large number of small atomic regions being formed that incur more overhead than they provide optimization opportunity. The getitem method is not being inlined because it contains what appears to be a polymorphic call site and the region formation algorithm will not partially inline methods containing polymorphic calls. If getitem were inlined, however, this call site is perfectly monomorphic. By forcing the implementation to recognize this fact, getitem is inlined and the atomic configuration’s 9% slowdown becomes a 10% speedup, as shown by the gray bars in Figure 6.7. These performance benefits may also be achieved through implementing an adaptive recompilation strategy that performs aggressive speculation (e.g., assumes polymorphic call sites are monomorphic) and recompiles methods containing frequently misspeculating asserts.

### 6.5.2 Architectural Analysis of Atomic Regions

This section studies the atomic regions generated by the compiler from the hardware’s perspective. In terms of implementing atomicity in hardware, it is important to understand the size of atomic regions in terms of dynamic instruction count and data footprint.
If the compiler-generated atomic regions were consistently small, they could be buffered completely within the pipeline; this is not the case. A 128-entry reorder buffer would be unable to support nearly 25% of the atomic regions executed, resulting in frequent aborts and significant performance degradation. A small fraction of atomic regions even contain over 1,000 uops. By using register checkpoints for recovery (similar to branch checkpoints except that they live past speculative retirement), hardware enables the compiler to construct such regions.

The evaluated hardware implementation uses the data cache to buffer the reads and writes in the atomic regions, similar to prior work [31,75]. Modern L1 data caches are easily sufficient for holding the read and write set of an atomic region. The majority of dynamically executed atomic regions access less than 10 cache blocks and 50 cache blocks is sufficient for 99% of the atomic regions (for reference a 32KB cache with 64B blocks holds 512 blocks). Only 110 out of the 1.7 million dynamically executed atomic regions touched more than 100 cache blocks and only one overflowed the cache. Clearly, the region selection algorithm is effective at tolerating the constraints of a bounded atomic primitive.

Once again, the read and write sets of the atomic regions fit easily within the cache, but the number of loads and stores, which tend to be proportional to the number uops in the atomic region, are generally too large to fit in the load and store buffers. Even if the region formation algorithm described in Section 6.3 were tuned to select smaller regions, a compiler would have a difficult time guaranteeing that a region would satisfy more limited hardware. Smaller region sizes would also limit optimization opportunity.

6.5.3 Microarchitectural sensitivity

Because atomic regions are intended to improve single-thread performance, they must be implemented with minimal overhead in order to preserve the benefits achieved by compiler optimizations. All the experiments thus far have assumed a checkpoint execution substrate (similar to checkpoint processors) to implement hardware atomicity with high performance.
Figure 6.9: **Sensitivity to hardware atomicity implementation.** All runs use the same code (atomic+aggressive inlining) on different hardware configurations: **chkpt:** the base high-performance non-stalling checkpoint execution substrate, **+ 20-cycle:** stalls the pipeline for 20 cycles at every `aregion_begin`, and **single-inflight:** stalls an `aregion_begin` at decode if another uncommitted atomic region is already in the pipeline.

However, it is worth investing the performance of alternate or simplified implementations in the absence of a high-performance checkpoint substrate to provide atomicity. Alternate schemes can incur two additional sources of overhead: overhead in the form of serializing operations that may occur as part of the `aregion_begin` to record recovery state and serializing overheads that may occur due to simplified implementations of the `aregion_begin` and `aregion_end` instructions in the absence of a checkpoint substrate. It is possible to explore the performance sensitivity to these effects simply by modeling two ways such overheads may be exposed. First, the performance of the atomic+aggressive inlining compiler configuration was measured with a simulator configured to stall the pipeline for 20 cycles when processing an `aregion_begin`. Second, to study implementations that only permit a single atomic region to be in flight at a time; an `aregion_begin` is stalled at decode until all preceding atomic regions commit.

As shown in Figure 6.9, each of these configurations eliminate or drastically reduce the benefit of atomic regions in most benchmarks. The sole exception is antlr, which shows limited sensitivity because of its sparing use of atomic regions.
In addition to the baseline processor configuration, performance was also measured on two more-modest microarchitectures, as might be incorporated into future multiprocessors: a 2-wide OOO version of the baseline machine (pipeline widths reduced to 2/2/2) and a 2-wide half OOO configuration that halves the superscalar width and all other processor structures (including caches and TLBs). The relative speedups achieved by the atomic region-based optimizations closely tracked the 4-wide OOO results shown in Figure 6.7, generally within a percent or two.

6.5.4 Limitations of the existing compiler

In the process of implementing my atomic region-based optimizations, I found that sometimes the benefits of optimization were mitigated by limitations in the compiler’s other optimizations and code generation. One particularly spectacular example of this effect occurred when I tried to remove the garbage collection safe point from loops completely encapsulated in atomic regions, replacing it with a single load of thread’s local yield flag in the loop’s pre-header. As it turns out, the JVM’s register allocator implicitly relied on the call to the yield() function to prevent the registers within the loop from being assigned to variables only used outside the loop. If this call was removed, performance degrades because many of the frequently accessed variables within loops are spilled to the stack.

As such, these performance results should not be considered a definitive characterization of the potential of atomic regions. I believe that significant further optimization potential exists. Nevertheless, it is important to recognize that atomic regions primarily facilitate the optimization phase of the compiler, and must be complemented by high quality code generation and run-time services to achieve high performance.
6.6 Conclusion

This chapter demonstrates that the introduction of architectural support for atomic regions greatly facilitates the implementation of speculative optimizations in a JIT compiler. The atomic region abstraction permits the compiler to isolate the hot paths for the purpose of optimization by replacing infrequently-executed paths with asserts. As a result, speculative optimizations can be performed without compensation code, enabling a great reduction in compiler complexity to achieve a given code quality. The prototype implementation achieves average speedups of 12% across a suite of DaCapo benchmarks, with commensurate reductions in the number of micro-operation flowing through the pipeline.

However, these experiments used short simulations using a modified JVM and therefore suffered from several shortcomings. For example, the extra runtime compilation costs were not measured and these might overwhelm the benefits achieved. Likewise, an atomic region optimization can become unprofitable as a program changes phases and complete program runs are necessary to fully observe and properly react to such effects. Finally, the prototype focuses on managed languages and ignores pre-compiled native binaries.

The next chapter addresses these shortcomings and describes the integration of atomic regions into a real dynamic binary translator system.
Chapter 7

Atomic Regions for Dynamic Translation†

Even though managed languages have gained popularity, statically compiled workloads will likely continue to be commonplace. Further, many of these programs are shipped as binaries and are unlikely to be recompiled for future systems. Therefore, this chapter explores the atomic region abstraction in dynamic binary translation systems. Such systems enable transparent compiler-based optimization of a program while it runs and can transparently make use of hardware atomicity primitives to support atomic region optimization. In addition, the dynamic nature of these systems provides access to accurate profile information necessary for speculative optimizations similar to those already described for managed languages.

To illustrate this point, I will show two examples from the SPECint 2000 benchmark suite [96], one from 255.vortex and one from 176.gcc. In each of these examples I will first describe how they are optimized by a commercial dynamic translation system for x86, specifically Code Morphing Software (CMS) [32] for the Transmeta Efficeon processor, and then describe how the atomic region paradigm enables their further optimization.

I will then provide background on CMS for the Transmeta Efficeon processor and describe how the atomic region abstraction was incorporated into a real system. With moderate effort, the atomic region abstraction is able to provide up to a 9% performance improvement (3% on average) on full runs of the SPECint 2000 benchmarks. In order to achieve these performance gains, it is necessary to identify and respond to frequent misspeculation, but I will show that even a simple control mechanism is sufficient to rein in all detrimental side-effects.

†The content of this chapter derives from work published at the 15th International Conference on Architectural Support for Programming Languages and Operating Systems [82].
Figure 7.1: Potential for atomic region optimizations. Optimization region for method ObjectGet from vortex. (a) Control flow graph for optimization region with the hot path highlighted, (b) dataflow graph and schedule for region as optimized by CMS using superblocks, (c) atomic region representation of the same control flow graph with cold paths converted into asserts, (d) dataflow graph and schedule for region optimized with atomic regions. Atomic regions enable CMS to trivially exploit additional optimization opportunity beyond that provided by superblocks alone.

7.1 SPECint 2000 Vortex example

Shown in Figure 7.1(a) is the control flow graph (CFG) for a portion of the method ObjectGet. The portion shown is the optimization region selected by the Transmeta CMS translator. This is one of the hottest regions in vortex and accounts for approximately 10% of the overall execution time. A single hot path exists through this optimization region, including 56 x86 instruction and nine cold exits.

CMS uses superblock scheduling in conjunction with a suite of classical optimizations to generate the Efficeon code and schedule shown in Figure 7.1(b). This code has been aggressively scheduled and optimized including the speculative hoisting of loads, reordering of memory operations and removal of redundant operations. As described in previous
work [32], this requires no compensation code because CMS makes use of the atomic execution hardware provided by Efficeon.

Ignoring cache misses, this optimized code emulates the original 56 x86 instructions by executing 72 Efficeon operations in 26 cycles or an average of 2.15 x86 instructions per cycle (IPC) and 2.77 IPC in Efficeon operations. The code generated by CMS is of high quality: the static schedule produced has a similar height to the dynamic schedule achieved by a modern out-of-order processor given the same number of functional units.

![Figure 7.2: CMS baseline optimization.](image)

CMS uses temporaries to eliminate three redundant loads and eliminate a fourth load by forwarding a previously stored value. Original x86 code is shown on the left. The code on the right illustrates the effect of optimized CMS code.

Figure 7.2 shows a few of the optimizations that CMS applies to the region from Figure 7.1(a). In particular, CMS is able to identify three redundant load operations and eliminate them by buffering previously loaded values in temporary registers. Likewise, CMS forwards a value stored to the stack to a later consumer, which obviates a fourth load. In each of these cases, CMS uses liveness analysis to prevent unnecessary register copying (for example, the temporaries introduced in block G are never copied to register edi because of a subsequent kill).

Although the code generated by CMS is already of high quality, hand inspection shows
Figure 7.3: Optimizations enabled by atomic regions. Asserts trivially expose speculative opportunities to classical optimizations. For example, a partially dead store gains the appearance of a fully dead store and a partially redundant exit computation appears fully redundant.

that significant optimization opportunity remains. Put simply, superblock scheduling and optimization has enabled CMS to eliminate some redundancies, hoist critical operations past exits and generate a nearly optimal schedule, but it has not enabled CMS to remove operations that are only needed along cold exit paths.

Because the code generated by CMS already uses hardware primitives to execute the region atomically, these additional optimization opportunities can be trivially exposed by converting the cold exits into asserts, as advocated by the atomic region abstraction (shown in Figure 7.1(c)).

An assert operation simply performs a conditional check to verify that a cold exit has not been followed. If the cold exit is followed, the assert triggers an abort which causes the entire atomic region to be rolled back and redirects execution to code which includes the cold exit. An assert therefore enables the compiler to speculatively isolate frequently occurring paths in the CFG from rarely taken exits.

Figure 7.3 demonstrates a few of the additional opportunities the atomic region abstrac-
tion exposes in the same region from Figure 7.1(a). By converting cold exits into simple dataflow operations, an assert provides speculative opportunities with a non-speculative appearance. For example, after converting the cold exits in blocks F, G, and H the partially dead store in block F appears fully redundant to classical optimizations in CMS and is thereby eliminated. In addition, the partially redundant branch exit computation in block D becomes fully redundant after assert conversion (enabling the removal of the redundancies in block I).

Figure 7.1(d) shows the optimized code and schedule that results after converting cold exits into asserts. By removing the cold exits, CMS is able to remove 15 additional Efficeon operations from the region. Given fewer operations and fewer control flow constraints, the superblock scheduler generates a 27% shorter schedule. This more aggressively optimized code now executes in 19 cycles at an average x86 IPC of 2.95.

In contrast, implementing the same optimization using a software-only approach would require complex and difficult to implement techniques. First, operations only consumed along cold exits would need to be identified and eliminated. Next, compensation code for these eliminated operations would need to be pushed onto the cold exit paths to maintain correctness. Finally, the implementation of these analysis and code transformations must be fast enough to be practical for CMS.

This is not only a complex undertaking but it can result in significant code duplication (see Chapter 3). For example, one of the speculative optimization opportunities available in Figure 7.1(a) is a store in basic block J that is made dynamically dead by a store in block O. Eliminating the store in basic block J requires placing compensation copies along each of the exit paths of blocks J, K and M. Without atomic regions, correctly identifying all the available opportunities and properly placing the necessary compensation code—all in a fast and efficient implementation—is a non-trivial proposition.

This vortex example is a rather simple one. The next example introduces further complexity to demonstrate additional opportunity exploitable by atomic regions as well as the
Figure 7.4: **Unbiased control flow in an atomic region.** Optimization region for method `combine_movables` from `gcc`. (a) Control flow graph for optimization region with the hot path highlighted and control flow paths annotated with their frequency. (b) atomic region representation of the same control flow graph with highly biased paths converted into asserts.

need for a fast but intelligent misspeculation recovery and control mechanism.

### 7.2 SPECint 2000 GCC example

The following example, taken from `gcc`, demonstrates an additional optimization opportunity enabled by atomic execution. It also shows that some cold paths are occasionally taken and therefore fast misspeculation recovery is needed to profitably speculate on these paths as well as an intelligent control mechanism that is capable of tolerating infrequent misspeculations.

Shown in Figure 7.4(a) is the control flow graph (CFG) for a portion of the method `combine_movables` as selected for optimization by the Transmeta CMS translator. This is a commonly executed region in `gcc` and accounts for approximately 1% of the overall execution time. The region includes 32 x86 instructions and eight exit branches. Of the nine possible paths through the region three are common (*i.e.*, executed greater than 1% of the time), one is relatively uncommon (*i.e.*, executed less than 1% but still more than 0.01% of the time),
three are rarely executed (i.e., executed 0.01% of the time or less) and two more are never executed.

Using superblock formation and scheduling, CMS generates aggressively scheduled code, including the speculative hoisting of loads and reordering of memory operations. In the absence of cache misses, the generated code will execute 57 Efficeon operation to emulate the original 32 x86 instructions in 19 cycles, for an average x86 IPC of 1.68 and Efficeon IPC of 3. Similar to the vortex example, CMS has been able to generate high quality and well scheduled code but optimization opportunity still remains. A number of operations are only needed along rarely or never taken paths and two of the cold conditional exit computations can be merged.

Eliminating operations only consumed along cold paths is trivial using atomic region asserts. By converting the cold exits into asserts (shown in Figure 7.4(b)), the results of these operations are no longer consumed along any path and therefore can be removed by classical dead-code elimination.

Figure 7.5: **Assert merge optimization.** Asserts convert biased control flow into dataflow operations which can be further optimized. For example, a pair of similar conditional checks can be combined into a single assert operation which subsumes the originals.

Merging exit condition computations depends on the all-or-nothing property of atomic execution. As shown in Figure 7.5, the two of the condition checks in this region are computed using the same logical operation (i.e., bitwise AND) and register operand but have differing immediates. Once these exits have been converted into asserts it becomes apparent that the atomic region will only commit if both assert checks pass. Therefore, a simple extension to classical constant folding enables CMS to identify and exploit this opportunity by merging
similar assert operations. Also shown in Figure 7.5, the first assert is strengthened so that it subsumes the second assert and enables it to be eliminated. Recall that an assert is a purely dataflow operation in the IR and therefore classical redundancy elimination techniques are sufficient to identify and eliminate the second assert.

After converting cold exits into asserts and after merging similar assert operations, CMS is able to remove 9 additional Efficeon operations from the region (not shown). In addition, converting six cold exits into asserts leaves fewer control flow constraints and the superblock scheduler is thereby able to generate a 32% shorter schedule. This more aggressively optimized code now executes in 13 cycles at an average x86 IPC of 2.46.

However, converting the uncommon but taken exits from this region into asserts poses a challenge to the atomic region paradigm. If taken these exits will first cause their equivalent assert to trigger an abort, causing hardware to rollback and restart execution at nonspeculative recovery code. Misspeculating on these asserts incurs a penalty of the time spent partially executing the atomic region, rolling back and restarting at the recovery code. In contrast, the performance benefit of successfully speculating on each assert operation is likely to be small. For example, converting six cold branches in `combine_movables` into asserts only saves six cycles.

As already described in Chapter 5, in order for an assert to be profitable the inequality in Equation 7.1 must be satisfied. Specifically, it is only profitable to speculatively remove branches whose bias exceeds the ratio of the misspeculation penalty to speculation benefit.

\[
\frac{\text{correct_preds}}{\text{incorrect_preds}} > \frac{\text{penalty}}{\text{benefit}} \tag{7.1}
\]

It is therefore important to minimize the cost of misspeculation so that a larger set of cold paths are profitable to convert into asserts. For example, assuming a misspeculation cost of 20,000 cycles and a speculation benefit of 1 cycle it would not be profitable to generate two of the asserts shown in Figure 7.4(c). On the other hand, reducing the misspeculation cost
to 200 cycles would make all of the asserts shown profitable.

Likewise, a control mechanism is necessary to identify and disable unprofitable asserts. Even with an accurate profile used to select branches to convert into asserts, poor selections will be made simply due to changes in program behavior. A branch that was extremely biased when a profile was collected may exhibit unbiased behavior later on. Profile inaccuracies only worsen the situation.

In addition, the control mechanism must be intelligent. A naive threshold-based control mechanism would be unable to differentiate between profitable and unprofitable asserts. An assert which misspeculates a large number of times is tolerable (and even desirable) so long as it is correctly speculated often enough to satisfy the inequality in Equation 7.1

Both of the previous examples imply that a compiler must choose the same optimization scope for atomic regions as it would for superblocks. For this work, I indeed engineer the compiler in this way. However, atomic regions are more general than superblocks because they can contain arbitrary control flow and therefore can encapsulate larger optimization scopes. While there could be a benefit to taking advantage of this difference, such an exploration is beyond the scope of this dissertation.

7.3 Background

The Transmeta Efficeon, first released in 2003, utilizes a low complexity design to provide high-performance and low-power x86 compatibility. The Efficeon hardware is a very long instruction word (VLIW) processor that executes an instruction set dissimilar x86. The instruction set is designed to enable the Code Morphing Software (CMS) software system to faithfully execute x86 code through interpretation and by dynamically translating x86 code into high-performance native code.

In this section, I first describe the architecture of the Efficeon processor. A brief overview of CMS and how it is used to provide high-performance execution of x86 binaries on the
Efficeon executes molecules composed of a variable number of 32-bit packets. Each packet includes a \textit{stop} bit as well as a \textit{type} field that specifies which functional unit it uses.

Efficeon then follows.

### 7.3.1 Efficeon Processor Architecture

The Efficeon is an in-order VLIW processor, which is designed to provide high-frequency execution of software-scheduled code. To further simplify the design and reduce power, it does not provide hardware interlocks or register scoreboarding and therefore relies completely upon a compiler to correctly schedule dependent and independent operations. To simplify the compiler’s task, Efficeon provides hardware support for taking fast register and memory checkpoints and for reordering memory operations.

Depicted in Figure 7.6, an Efficeon VLIW instruction, or \textit{molecule}, is variable length and composed of 32-bit \textit{packets}. Each packet includes a \textit{stop} bit, which denotes the end of a molecule. A molecule may contain up to eight packets.

A packet typically encodes a functional operation, or \textit{atom}, but may also encode auxiliary information such as longer immediates or memory alias protection. An Efficeon atom has a three-address format and is analogous to an instruction from a load-store architecture. An
atom is statically assigned to one of seven functional units: two memory, two integer, two floating point and one branch.

The Efficeon processor provides hardware support for fast register and memory checkpoints. The Efficeon has two copies of each register: a *shadowed* and a *working* copy. Likewise, the Efficeon includes a *speculative* bit for each line in its data cache [90]. Between checkpoints, all updates are speculatively written to either working registers or to the data cache. If a cache line is speculatively written, its speculative bit is set and it is transitioned to the dirty state (after first evicting any non-speculative dirty data on the line into a victim cache). The hardware can *commit* speculative work in a single cycle by copying the working registers onto their shadowed counterparts and flash clearing all speculative bits in the data cache. Alternatively, the hardware can *rollback* all speculative work by restoring the working registers from their shadowed counterparts and flash invalidating all speculative lines in the data cache. Section 7.4.1 describes the primitives used by software to control this commit and rollback hardware.

The Efficeon also provides memory alias detection hardware, which a compiler can use to guarantee the correctness of reordered memory operations. Often memory operations do not alias, but the compiler can not statically prove their independence. In these situations, the compiler can generate code that includes *alias* packets. If used to initiate protection of a coupled load or store atom, an alias packet captures the memory address used by the atom into an alias register. If used to detect an alias with a coupled load or store atom, the alias packet compares the memory address against the contents of one or more alias registers, and, if a match is made, an alias fault is triggered. In this way, software can check if speculatively-reordered loads alias with the stores they were hoisted above. The alias hardware also enables the compiler to eliminate redundant loads and stores in more situations [64].

The Efficeon shares many of the above architectural traits with the Transmeta Crusoe that preceded it, but several key differences exist [64]. Both processors have statically
scheduled VLIW architectures but the Efficeon can issue seven atoms per cycle versus four atoms in the Crusoe. To provide better code density, an Efficeon molecule is variable in length whereas a Crusoe molecule may only be two or four packets long. Most relevant to this paper, the Efficeon has more relaxed speculation support because it buffers all speculative memory updates in a 64-KB first level data cache whereas the Crusoe buffers all speculative memory updates in a gated store buffer.

7.3.2 CMS Overview

The Transmeta Code Morphing Software [32] is a software system designed to provide high-performance execution of x86 binaries on Efficeon hardware. To accomplish this goal, it includes a robust and high-performance dynamic binary translator, supported by a software x86 interpreter. The translator is a large and well-tuned software system which includes components to identify commonly-executed regions of x86 code, convert the corresponding x86 instructions into a three-address intermediate representation (IR), and then optimize, schedule, and deploy each translated region.

The first several times a group of x86 instructions is encountered by CMS they will not be translated. Rather, they will be emulated by the CMS interpreter. In doing so, CMS is able to collect a dynamic execution profile of the x86 instructions as well as provide a low-latency “cold start” response. If a group of x86 instructions is executed enough times, CMS will generate a translation for them.

A CMS translation is a multiple-entry, multiple-exit region of x86 instructions that can contain arbitrary control flow such as indirect branches, divergences, and loops. As with other dynamic translation systems, exits from a CMS translation are directly linked, or chained, to other translations [7,27]. In CMS, chaining is lazily performed the first time an exit is executed.

The CMS translator uses a staged optimization strategy in managing its translations. Translations are first lightly-optimized, but later promoted to an aggressively-optimized
translation if executed frequently enough. This staged optimization strategy enables CMS to focus its compilation efforts on the small subset of an x86 program’s instructions where the majority of execution time is spent.

In this chapter, I focus my efforts on improving the quality of these aggressive translations, and the remainder of this section focuses on the design of the aggressive optimizer.

The aggressive optimizer is designed to enable compiler optimizations to exploit as much available opportunity as possible, while keeping the total compilation time to a minimum. It primarily accomplishes these goals through a careful organization of compilation steps as described below:

1. **Region preparation:** Decode the selected x86 region into a three-address code intermediate representation (IR).

2. **Flow analysis:** Generate a topological ordering of basic blocks in the region, compute dominators and post-dominators, and rename operands into a static single-assignment (SSA) form.

3. **Control flow manipulation:** Unroll loops, if-convert short branch-overs and control flow divergences. Also create single-entry multiple-exit sub-regions (hyperblocks) that are wrapped with checkpoint commit points to provide atomicity. Incrementally update the already computed flow analysis as necessary.

4. **Forward dataflow pass:** In a single forward pass, apply a suite of optimizations such as constant folding and propagation, common subexpression elimination, and several peephole optimizations. Also perform a simple alias analysis to guide redundant load elimination and later memory optimization and scheduling passes.

5. **Backward dataflow pass:** In a single backward pass, perform a liveness analysis to guide dead-code elimination and dead-store elimination.
6. **Schedule and lower**: Perform loop-invariant code motion, hoist critical operations, allocate registers, perform code lowering, and schedule each hyperblock.

7. **Emit**: Assemble all instructions and update branch targets.

There are two key differences between this organization and that typically employed by a static compiler. First, the CMS translator is broken into distinct phases which constrain the types of changes that can be made to the IR at any given point. For example, modifications to the control flow graph are only performed in Step 3, meaning that later passes can rely on an immutable control flow structure. Likewise, flow analysis is performed early in Step 2 and is properly updated by the control flow manipulation passes so that later phases can rely on accurate information about loop structure, dominators, and post-dominators.

Second, dataflow optimizations that are typically implemented as separate passes in a static compiler are instead folded into a single forward dataflow pass and a single backward dataflow pass. For example, the forward dataflow pass processes the region in topological order and applies a suite of global analysis and optimizations as it visits each statement in a basic block. In doing so, the benefits of several (in the case of CMS, seven) forward dataflow passes can be achieved in roughly the same amount of time as a single forward pass.

These design differences are key to the efficiency of the translator and, thereby, the performance of CMS as a whole. In adding additional optimizations to CMS, it is important to respect these efficiency considerations. In the context of a dynamic optimizer, a powerful but computationally complex optimization is untenable. As the next section will show, incorporating the atomic region abstraction is not only easy to do, but can be done without adding significant overheads.

### 7.4 Atomic Regions in CMS

This section describes the modifications made to CMS in order to incorporate the atomic region abstraction. I first discuss the hardware atomicity primitives that Efficeon provides
Table 7.1: **Efficeon atomicity primitives.** Software uses these operations to control the Efficeon commit and rollback hardware

| Commit | Copy *working* registers into *shadowed* registers.  
| Mark *speculative* lines in the data cache as dirty. |
| Rollback | Copy *shadowed* registers into *working* registers.  
| Invalidate *speculative* lines in the data cache. |

and how they can be used by a software compiler to implement the atomic region abstraction. I then describe how atomic regions were integrated into CMS. Lastly, I introduce a simple mechanism to rein in frequent misspeculations and an optimization for removing redundant asserts.

### 7.4.1 Hardware Atomicity

The Efficeon processor exposes its support for fast hardware checkpoints through the two operations shown in Table 7.1. Software can use these operations to provide the illusion of *atomic execution*—the execution of a region of code completely or not at all.

The **commit** operation is used to denote both the beginning and the end of an atomic region. It is used at the beginning of an atomic region to take a register checkpoint and to treat all future register and memory updates as speculative. It is used at the end of an atomic region to commit all speculative updates and discard the last checkpoint. The **rollback** operation is used to unconditionally abort an atomic region by restoring the last checkpoint. A rollback does not affect the program counter, so an instruction following the rollback can be used to redirect control flow as necessary.

Figure 7.7 illustrates how the CMS translator could use these operations to speculatively optimize a region of code. The optimizer first wraps an optimization region with commit points, and then speculatively removes cold paths from the region. To guarantee correctness, the optimizer inserts a check, *i.e.*, assert, to verify that the cold path is not taken. If the assert determines that the cold path is needed, a rollback is executed that instructs the hardware to discard all speculative state. Control is then redirected to a non-speculative
Figure 7.7: **Atomic region example using the Efficeon atomicity primitives.** If speculation succeeds, the assert path will not be taken and execution will reach the commit at the end of the region. If speculation fails, the abort path executes a rollback before invoking recovery code to restart execution at a non-speculative version of the same code (*e.g.*, via the CMS interpreter).

version of the same region. In this example, execution resumes in the CMS interpreter.

It should be noted that the Efficeon hardware is designed to provide atomicity in a uniprocessor environment. In a multiprocessor environment, additional support is necessary to provide an illusion of atomicity to other threads. Essentially, loads must also be handled speculatively and coherence traffic must be monitored to detect atomicity violations. The necessary support has previously been proposed [13, 83, 89].

### 7.4.2 Incorporating Atomic Regions into CMS

The CMS translator already uses the hardware atomicity primitives to obviate the need for recovery code in superblocks. The CMS optimizer wraps each superblock with commit points to simplify the recovery of precise state in the case of a misspeculation or excep-
tion. For example, if a speculatively hoisted load incurs a memory fault, CMS relies on the
hardware to discard all speculative state and afterward redirect execution to a more con-
servative implementation of the same code (by dispatching to the interpreter in the current
implementation).

However, the CMS translator does not use hardware atomicity to expose speculative
optimization opportunities resulting from biased control flow. As shown in Sections 7.1
and 7.2, the translator can be made to better optimize code by simply generating an atomic
region with rarely executed paths removed. Extending the CMS translator to use the atomic
region abstraction required three additions: representing an assert operation in the IR, a
mechanism for converting biased branches into asserts, and a mechanism for recovering from
misspeculations.

**Assert operations:** The assert operation is represented in the compiler IR as a pseudo
operation. The assert is used to speculatively convert highly-biased conditional branches
into straight-line code. The assert consumes the same condition as a biased branch and—
like the branch it replaces—has no dataflow consumers. Unlike a branch, no operations are
control dependent on an assert, which means that it is not an optimization obstacle for later
passes. The assert is treated as a potentially-excepting operation in the IR to prevent it
from being hoisted out of its atomic region, and an assert is annotated with a numerical
identifier to distinguish it from other asserts in the same region.

**Converting biased branches into asserts:** An accurate execution profile is necessary to
identify which conditional branches are good candidates to convert into asserts. Misspecula-
tions can be very costly, so only highly-biased branches should be converted. However, CMS
does not collect a profile that is sufficient to properly distinguish good candidate branches.
The execution profile collected by the CMS interpreter simply does not include enough
samples to be useful for the purpose of identifying assert candidates.

Rather than forcing the interpreter to collect more samples or adding instrumentation
code to lightly-optimized translations, both of which could incur costly performance overheads, I instead turned my attention to the translation chaining mechanism.

As described in Section 7.3.2, translation exits are lazily chained to other translations. Therefore when a lightly-optimized translation is promoted and retranslated, rarely taken translation exits are unlikely to have been chained. Similarly, the conditional branches corresponding to these unchained exits are likely to be biased. I have implemented a heuristic based on this observation that strikes a reasonable balance between being able to identify good assert candidates and minimizing profiling overheads.

The CMS translator was modified so that it consults chaining information when promoting a lightly-optimized translation. All unchained exits are considered assert candidates, and this information is provided to a new flow manipulation pass added to Step 3 of the optimizer.

**Misspeculation recovery:** Throughout most of the optimizer, the assert is represented as a single dataflow pseudo-operation. In the final code emit step this changes, and the assert is emitted as a conditional branch which targets rollback code. The exact rollback routine that the assert targets depends on the numerical identifier of the assert. Each identifier is associated with a separate rollback routine to simplify misspeculation monitoring (described in Section 7.4.3).

There are a maximum of 31 chainable exits in a translation, and the numerical identifier assigned to an assert is the same as the exit number of the cold branch it replaces. I therefore added 31 rollback routines to CMS which are shared by all asserts. As shown in Figure 7.7, a conditional branch is emitted for each assert that targets the rollback routine with the corresponding numerical identifier.

When an assert fires, control is directed to its rollback routine, which first executes a rollback instruction to discard all speculative register and memory state. It then loads the identifier of the triggered assert into a register and jumps to the misspeculation recovery routine. This misspeculation recovery routine is responsible for recovering the x86 instruction.
pointer and dispatching to the interpreter (to execute the same code non-speculatively). The misspeculation recovery routine is also responsible for monitoring each assert, which I describe next.

7.4.3 Monitoring Speculations

Even though the heuristic for identifying biased branches is reasonably accurate, it is still fallible. It occasionally leads CMS to convert branches into asserts that fire frequently. Often the cause is a change in program behavior: a path that was rarely executed early in the program becomes a common path later in the program. If these problematic asserts are left untended, they will adversely affect performance because of the relatively high cost associated with misspeculation. Therefore, a mechanism is necessary to identify and disable problematic asserts [108].

I developed a simple solution by augmenting the misspeculation recovery routine. The routine updates a misspeculation counter corresponding to the assert that fired. If this counter exceeds a threshold, then the assert is designated misbehaving, and the translation will be reoptimized with the corresponding assert disabled.

Furthermore, it is desirable to tolerate asserts that fire infrequently relative to the total number of times they execute. For these asserts, the performance improvements provided by each successful execution of the assert outweighs the infrequent misspeculation costs. To distinguish between asserts which are problematic and asserts that are tolerable, it is ideal to know the local assert rate, or the number of times an assert fires relative to the number of times it executes.

Discovering the precise execution frequency of an assert is difficult, as it would require intrusive profiling. In the interest of minimizing overheads, I use an alternative approach.

---

1The counter does not increase the size of the metadata associated with a translation because an assert replaces what would have otherwise been a translation exit. Each translation already includes eight bytes of metadata for each translation exit and I simply reappropriate the same storage for each assert.
based on hardware sampling.

By default, CMS takes a program counter sample every 200,000 cycles so that it can identify and promote frequently executing translations. If a sample is taken while a translation is executing, a counter in the translation metadata is incremented. I can therefore use this counter as an approximation for translation execution frequency.

My assert monitoring mechanism is shown in Algorithm 7.1. Essentially, whenever an assert misspeculation counter is updated I also capture the value of the translation sample counter. When the next misspeculation occurs, the code checks whether a sample has been received since the last time the assert fired—by comparing the captured sample value to the current translation sample counter value. A changed sample counter value implies that the translation is commonly executed, and by proxy so is the assert being monitored. To reflect that misspeculations from commonly executed asserts should be tolerated, the misspeculation counter is reset if the sample values do not match. Otherwise, when the assert count exceeds a threshold it is disabled through retranslation.

However, workloads with a large number of commonly executed translations will have an increased latency to detect misbehaving asserts. To prevent this increased detection latency from adversely affecting performance, I also incorporated a mechanism to monitor the global assert rate, or the total number of asserts firing per cycle.

Algorithm 7.1 also shows this global monitoring mechanism. Every 100 million cycles the global monitor is invoked to check if the global assert count exceeds a global assert threshold. If the threshold is exceeded, the parameters of the local assert monitoring mechanism are tightened: either by increasing the assert sample shift parameter (to require sample counter values to differ in more significant bits before considering a translation commonly executed) or by reducing the local assert threshold.
Algorithm 7.1 Assert misspeculation monitoring

// Monitors the behavior of a misspeculating assert.
// Returns true if the assert should be disabled.
procedure MonitorAssert(assertID, transID)
    currSample ← GetTransSampleValue(transID)
    lastSample ← GetCapturedSampleValue(assertID)
    GlobalAssertCount ← GlobalAssertCount + 1
    if AssertSampleMatches(currSample, lastSample) then
        assertCount ← GetAssertCount(assertID) + 1
        if assertCount > AssertThresh then
            return true
        else
            SetAssertCount(assertID, assertCount)
            return false
    else
        SetAssertCount(assertID, 1)
        SetCapturedSampleValue(assertID, currSample)
        return false

// Compares two sample values. Returns true if they are equivalent
// after shifting off some of their least significant bits.
procedure AssertSampleMatches(currSample, lastSample)
    mismatchBits ← currSample ⊕ lastSample
    mismatchBits ← mismatchBits ≫ AssertSampleShift
    return mismatchBits ≡ 0

// Global assert monitoring. If the global assert rate is too high,
// tighten the local sample shift or threshold. Otherwise, loosen them.
procedure GlobalMonitor
    if GlobalAssertCount > GlobalAssertThresh then
        if AssertSampleShift < MaxSampleShift then
            AssertSampleShift ← AssertSampleShift + 1
        else if AssertThresh > 0 then
            AssertThresh ← AssertThresh − 1
    else
        if AssertThresh < LocalAssertThresh then
            AssertThresh ← AssertThresh + 1
        else if AssertSampleShift > 0 then
            AssertSampleShift ← AssertSampleShift − 1
    GlobalAssertCount ← 0
7.4.4 Eliminating Redundant Asserts

After biased branches have been converted into asserts, opportunities exist to remove some of these asserts from the CFG. These opportunities arise either because an assert is redundant—another assert in the same atomic region implements the same (or subsuming) check—or because an assert can be proven to never fire.

The existing common subexpression elimination and constant evaluation optimizations were easily modified to recognize assert operations. I also added an optimization that can eliminate an assert if it is rendered unnecessary by a stronger assert. One assert subsumes than another if it would fire in at least every situation that the other would fire. For example, an assert that fires whenever $r1 < 5$ subsumes an assert which fires whenever $r1 < 4$.

I also extended common subexpression elimination to allow an assert to be removed if it is post-dominated by an equivalent or subsuming assert. Typically an operation must be dominated by an equivalent operation to be removed, but atomicity makes post-dominance a sufficient proxy for dominance.

7.5 Evaluation

In this section, I present the result of incorporating the atomic region abstraction into CMS, evaluated on an Efficeon hardware platform. I first describe the configuration of the evaluation system and provide compilation details for the benchmarks used. I then present and interpret the experimental results. Overall, I find that incorporating atomic regions into CMS provides a 3% average performance improvement, and that a simple assert monitoring mechanism is sufficient enough to prevent slowdowns in any individual benchmark.
### Table 7.2: Evaluation system configuration

<table>
<thead>
<tr>
<th>Processor</th>
<th>Transmeta Efficeon 2 (TM8800)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor frequency</td>
<td>1.2 GHz</td>
</tr>
<tr>
<td>Dynamic Translator</td>
<td>CMS 7.0 (pre-release)</td>
</tr>
<tr>
<td>Registers</td>
<td>64 integer, 64 FP</td>
</tr>
<tr>
<td>Translation Cache</td>
<td>32 MB (of physical memory)</td>
</tr>
<tr>
<td>L1 Instruction Cache</td>
<td>128 KB, 4-way, 64B line</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>64 KB, 8-way, 32B line</td>
</tr>
<tr>
<td>Victim Cache</td>
<td>1 KB, fully-associative, 32B line</td>
</tr>
<tr>
<td>L2 Unified Cache</td>
<td>1024 KB, 4-way, 128B line</td>
</tr>
<tr>
<td>Physical Memory</td>
<td>1 GB DDR-400</td>
</tr>
<tr>
<td>Operating System</td>
<td>Linux 2.6.19</td>
</tr>
<tr>
<td>Compiler (for SPEC)</td>
<td>Intel C++ Compiler 11.0</td>
</tr>
<tr>
<td>Compilation options</td>
<td>-O3 -ipo -no-prec-div -prof_use</td>
</tr>
<tr>
<td>Local Assert Threshold</td>
<td>8 per translation sample</td>
</tr>
<tr>
<td>Global Assert Threshold</td>
<td>16 per 100 million cycles</td>
</tr>
</tbody>
</table>

#### 7.5.1 System Configuration

All of the experiments are run using a Transmeta development system. Shown in Table 7.2, the system configuration is intended to closely represent retail Efficeon hardware\(^2\). Of particular note is the pre-release version of CMS used for this evaluation; this CMS version includes significant enhancements over the last retail version of CMS and is in many ways superior. In terms of raw performance, the pre-release version is marginally faster than the retail version on the benchmarks studied.

For this evaluation, all of the SPEC CPU2000 integer benchmarks were run to completion using the reference inputs. SPEC CPU2006 was not used because the evaluation system—representative of a computer circa 2004—does not satisfy the system requirements. The benchmarks are compiled with the Intel C++ Compiler using the highest performing SPEC “base” compiler options, including profile guided optimizations. All efforts have been made to use the best possible baseline.

---

\(^2\)Transmeta stopped selling microprocessors in 2005.
Figure 7.8: **SPEC CPU2000 integer results.** Results for three atomic region configurations. (d)isabled misspeculation monitoring, (m)onitoring enabled but assert optimizations disabled, (a)tomic region optimizations and monitoring fully enabled. All results have been normalized to the baseline CMS configuration, which has no atomic region support.

### 7.5.2 Experimental Results

The following experiments focus on understanding both the dynamic and static impacts of atomic regions. Broadly, atomic regions are able to improve performance over a baseline CMS by an average of 3% and by up to 9%. This performance improvement is achievable because frequently misspeculating asserts are identified and disabled by the simple assert monitoring mechanism described earlier. Likewise, the compilation overheads introduced by atomic regions are minimal. Finally, atomic regions do not suffer from static code bloat problems but rather they reduce static code size.

Figure 7.8 shows the performance of three configurations of the atomic region implementation in CMS, normalized to the runtime of the baseline configuration. The first configuration shows the performance of a system without the assert monitoring mechanism enabled. The second configuration enables the assert monitoring mechanism but does not use assert operations when speculating on biased exits (*i.e.*, biased-exit branches are simply re-targeted at rollback and recovery code). The third configuration is a complete implemen-
Table 7.3: **Atomic region statistics.** Lists the percentage of dynamic branches that are 99.999% biased or greater, the estimated cost of a misspeculating assert, and the overall misspeculation rates both with and without assert monitoring.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Dynamic Biased Branches (%)</th>
<th>Misspeculation Cost (cycles)</th>
<th>Misspec. / 1M cyc. (no monitoring)</th>
<th>Misspec. / 1M cyc. (w/ monitoring)</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>4.3</td>
<td>1790</td>
<td>1337</td>
<td>0.06</td>
</tr>
<tr>
<td>vpr</td>
<td>9.8</td>
<td>6188</td>
<td>87</td>
<td>0.08</td>
</tr>
<tr>
<td>gcc</td>
<td>5.8</td>
<td>1816</td>
<td>575</td>
<td>0.1</td>
</tr>
<tr>
<td>mcf</td>
<td>8.1</td>
<td>1167</td>
<td>968</td>
<td>0.07</td>
</tr>
<tr>
<td>crafty</td>
<td>14.9</td>
<td>2985</td>
<td>1529</td>
<td>0.06</td>
</tr>
<tr>
<td>parser</td>
<td>14.3</td>
<td>2135</td>
<td>604</td>
<td>0.1</td>
</tr>
<tr>
<td>eon</td>
<td>13.0</td>
<td>961</td>
<td>1127</td>
<td>0.04</td>
</tr>
<tr>
<td>perlbnk</td>
<td>30.7</td>
<td>1878</td>
<td>942</td>
<td>0.08</td>
</tr>
<tr>
<td>gap</td>
<td>26.9</td>
<td>1738</td>
<td>1879</td>
<td>0.08</td>
</tr>
<tr>
<td>vortex</td>
<td>38.7</td>
<td>1863</td>
<td>1236</td>
<td>0.1</td>
</tr>
<tr>
<td>bzip2</td>
<td>19.0</td>
<td>1179</td>
<td>1497</td>
<td>0.09</td>
</tr>
<tr>
<td>twolf</td>
<td>13.6</td>
<td>2598</td>
<td>252</td>
<td>0.07</td>
</tr>
<tr>
<td>average</td>
<td>16.6</td>
<td>2195</td>
<td>1003</td>
<td>0.08</td>
</tr>
</tbody>
</table>

The complete atomic regions implementation provides an overall performance improvement in nearly every benchmark and none of the benchmarks exhibited a slowdown. Atomic regions provides a 3% average improvement over the CMS baseline that uses superblocks. Seven of the benchmarks exhibit greater than a 2% performance improvement and three of these exhibit a greater than 5% performance improvement. The benchmark with the largest performance improvement is *vortex* at 9.3%.

The performance improvements exhibited roughly correlate with the percentage of dynamic branches which are considered highly biased. Shown in the first column of Table 7.3 are the percentage of branches executed in each benchmark which are 99.999% biased or greater (*i.e.*, branches for which fewer than 1 in 100,000 dynamic instances oppose the bias). The eight benchmarks with greater than 10% of their executed branches being biased exhibit a performance improvement of 2% or more (with the exception of *eon* which improves by
1.7%). Similarly, the three benchmarks which exhibit a 5% or greater performance improvement have greater than 25% of their branches being biased.

Targeting a 99.999% observed bias rather than a 100% observed bias is important to broadening the set of branches which will be considered profitable speculation candidates. Not doing so can have a significant impact on the performance achievable by eliminating profitable opportunities. For example, when only 100% biased branches are considered profitable the percentage of viable dynamic branches in gap drops to 11.2% and the performance improvement achieved reduces to 2.3% (from 5.9%).

The current implementation of atomic regions, is unable to profitably speculate on branches less than 99.999% biased, due to a high cost for misspeculation. Shown in the second column of Table 7.3 is the estimated cost of an assert misspeculation in each benchmark, which was measured using an instrumented version of CMS. Because the current implementation uses the CMS interpreter for recovery, each assert misspeculation costs thousands of cycles. Therefore, it is only worthwhile to speculate on branches that go against their bias significantly less than one out of every thousand executions. The selection of assert thresholds, shown in Table 7.2, satisfies this goal although the thresholds have not been highly tuned.

The high cost of an assert misspeculation can also cause severe performance degradation in a naive implementation of atomic regions. The first configuration in Figure 7.8 shows the performance lost if frequently misspeculating asserts are left untended. If problematic asserts are not disabled, such a configuration will incur a misspeculation once every thousand cycles on average. Combined with the high cost for misspeculations this results in a greater than factor of two slowdown for most benchmarks.

The simple assert monitoring mechanism introduced in Section 7.4.3 is sufficient to identify problematic asserts so that they can be disabled after retranslation. As shown in Table 7.3, this simple mechanism is able to reduce the misspeculation rate to fewer than one misspeculation every ten million cycles. In doing so, approximately one in five asserts are
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Static Asserts Disabled (%)</th>
<th>Static Code Reduction (%)</th>
<th>Static Asserts Eliminated (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>33.0</td>
<td>-0.1</td>
<td>0.9</td>
</tr>
<tr>
<td>vpr</td>
<td>1.1</td>
<td>0.6</td>
<td>1.2</td>
</tr>
<tr>
<td>gcc</td>
<td>4.5</td>
<td>0.9</td>
<td>1.2</td>
</tr>
<tr>
<td>mcf</td>
<td>24.0</td>
<td>0.4</td>
<td>1.2</td>
</tr>
<tr>
<td>crafty</td>
<td>28.7</td>
<td>0.4</td>
<td>0.7</td>
</tr>
<tr>
<td>parser</td>
<td>30.1</td>
<td>0.2</td>
<td>2.2</td>
</tr>
<tr>
<td>eon</td>
<td>21.4</td>
<td>0.6</td>
<td>1.1</td>
</tr>
<tr>
<td>perlbmk</td>
<td>17.9</td>
<td>1.9</td>
<td>2.1</td>
</tr>
<tr>
<td>gap</td>
<td>16.1</td>
<td>1.4</td>
<td>1.0</td>
</tr>
<tr>
<td>vortex</td>
<td>3.6</td>
<td>3.5</td>
<td>10.1</td>
</tr>
<tr>
<td>bzip2</td>
<td>34.2</td>
<td>0.4</td>
<td>0.9</td>
</tr>
<tr>
<td>twolf</td>
<td>9.4</td>
<td>0.6</td>
<td>1.0</td>
</tr>
<tr>
<td>average</td>
<td>18.7</td>
<td>0.9</td>
<td>2.0</td>
</tr>
</tbody>
</table>

Table 7.4: **Static code statistics.** Lists the percentage of static asserts disabled because they are misbehaving, the static reduction in translation code size enabled by atomic regions, and the percentage of static asserts that have been redundancy eliminated.

disabled through retranslation (shown in Table 7.4).

However, the additional retranslation costs are minor. The second configuration in Figure 7.8 measures the performance costs associated with atomic regions by enabling assert monitoring and retranslation but disabling all the optimization benefits of assert operations. The performance costs never exceed 1.5% and generally amount to less than 1% of overhead. Overall, this simple assert monitoring mechanism is sufficient and perhaps conservative.

In general, atomic regions also improve static code characteristics. Whereas superblocks can incur significant code bloat due to transformations such as tail duplication, atomic regions do not require duplication to expose additional opportunities. As Table 7.4 shows, static translation size generally decreases by a small amount. The reduction in static code is mostly the result of extra classical optimization opportunities exposed by atomic regions. The redundant assert elimination optimizations described in Section 7.4.4 are also beneficial as they are able to eliminate 2% of asserts on average (up to 10% in vortex).

These results demonstrate that atomic regions are able to offer significant performance
improvements on a real machine and that these performance improvements can be achieved without detrimental side-effects. In addition, it serves as a motivation for future work. As already mentioned, the high misspeculation cost prevents the current implementation from considering branches which are less than 99.999% biased. However, if the misspeculation cost could be reduced significantly, it should be possible to target a lower bias threshold and thereby broaden the set of branches that are profitable to convert into asserts.

To reduce the misspeculation cost, it is possible to implement a misspeculation recovery mechanism that redirects execution to a non-speculative translation rather than the CMS interpreter. Doing so will incur some code duplication, but so long as the duplication is incurred judiciously it could make for a worthwhile trade-off.

7.6 Conclusions

In this chapter, I have demonstrated that the previously proposed atomic region abstraction truly is simple and intuitive to integrate into a mature compilation system. I have also shown that atomic regions expose real performance opportunities even in a well-engineered commercial system.

My experience in this work has also resulted in several opinions on the relative merit and utility of atomic regions, especially in comparison to superblocks. Any view of atomic regions and superblocks as purely competitive abstractions is overly simplistic. Instead, in my experience atomic regions and superblocks are complementary and synergistic with one another.

Specifically, the key advantage of the atomic region abstraction lies in exposing opportunities to remove operations that are partially redundant or partially dead along hot paths (i.e., operations that are fully redundant or fully dead once cold paths are removed). The strength of the approach is the relative simplicity in which it can expose these opportunities. Although superblocks could be used to expose such optimization opportunities, doing so
requires significantly more effort. Despite being a mature and highly-engineering implementation, CMS does not exploit superblocks for partial redundancy or dead code elimination.

These observations have led me to believe that the real benefits of the superblock abstraction are scheduling optimizations that reduce the critical path height and increase instruction-level parallelism (ILP). On a wide in-order superscalar such as Efficeon, generating good static schedules is key to performance and therefore the superblock plays a critical role on these types of machines.

Looking forward, physical constraints portend a re-emergence of simple, in-order processor designs. To provide good single-thread performance, these designs necessitate sophisticated compiler infrastructures, and I believe that hardware support for speculative optimizations is an energy and complexity effective approach to achieving that performance. Specifically, I believe support for the atomic region, along with the superblock, is a strong candidate for incorporation into these future designs.
Chapter 8
Atomic Region Memory Model

One of the salient features of the atomic region abstraction is its simple and intuitive execution model, namely atomic execution. An atomic region will appear either to execute completely and instantaneously or will appear as if it never executed. As claimed in Chapter 2.1, this illusion simplifies speculative compiler optimization even in a multiprocessor system.

However, the atomic region is not a panacea. Modern programming languages specify memory consistency models that define observability rules for shared memory in a multiprocessor system. In simple terms, these memory consistency models restrict the values that can be observed by memory read operations [1]. From the point of view of a compiler, the memory consistency model constrains the statement reorderings which are available when optimizing a program. To maintain correctness, any use of the atomic region abstraction must also satisfy these constraints.

As this chapter will prove, atomic region formation trivially satisfies memory model constraints. A program region can be converted into an atomic region without changing the semantics of the program. Furthermore, within an atomic region, memory model constraints can be relaxed, which may expose additional optimization opportunity.

At the same time, the atomic region abstraction should not obscure other optimization opportunities. The boundaries of an atomic region, in particular, should not unnecessarily obstruct optimization. Therefore, this chapter will also prove that the atomic region abstraction continues to permit optimizations across atomic region boundaries.

The next section introduces the formal specification used throughout the chapter. The
remainder of the chapter contains proofs demonstrating the compatibility of atomic region formation with memory model constraints and the optimizations permitted by atomic regions.

8.1 Formal Specification of Multithreaded Programs

The terms, definitions and formal notation used in the remainder of this chapter borrow from Shasha and Snir [92], Manson et. al [74] and Effinger-Dean et. al [36].

The following specification considers programs as expressed by a programming language. A program is composed of a set of statements which express the behavior of the program and adhere to the semantics of the programming language. For the purposes of this chapter, both high-level languages such as Java and low-level machine languages such as the x86 ISA are considered programming languages. A program $P$ is formally specified by the tuple $P = \langle S, CFG, DFG \rangle$, where:

- $S$ is a set of statements.
- $CFG$ is the control flow graph and is a directed graph over the set of vertices defined by $S$. Each edge in $CFG$ expresses a control flow ordering between statements in $S$.
- $DFG$ is the dataflow graph and is a directed graph over the set of vertices defined by $S$. Each edge in $DFG$ expresses a dataflow ordering between statements in $S$.

Definition Suppose $s_x, s_y \in S$ such that a path exists from $s_x$ to $s_y$ in $CFG$, then $s_x$ is a predecessor of $s_y$. Likewise, $s_y$ is a successor of $s_x$.

Definition Suppose $s_x, s_y \in S$ such that an edge exists from $s_x$ to $s_y$ in $CFG$, then $s_x$ is an immediate predecessor of $s_y$. In particular, a path of length one exists from $s_x$ to $s_y$ in $CFG$. Likewise, $s_y$ is an immediate successor of $s_x$. 

125
**Definition** Suppose $s_x, s_y \in S$ such that a path exists from $s_x$ to $s_y$ in $DFG$, then $s_y$ is *data dependent* on $s_x$.

The definition for *control dependence* is determined by language semantics. Typically the classical definition of control dependence is used, which is determined by the post-dominance relation [39]. As mentioned by Manson et al. [74], the Java definition is *loop control dependence*, which additionally specifies that operations following any potentially infinite loop are control dependent on the loop [10]. For the purposes of this chapter, it is sufficient to note that control dependence can be derived from the control flow graph and language semantics of a program.

When a statement is executed, it becomes an *action* which may include reading or writing a memory location. For simplicity, statements are assumed to correspond with a single action. Two actions are said to *conflict* if at least one is a write and they both reference the same memory location. A multithreaded program may have different possible *executions*, each with a different interleaving of conflicting actions. An execution $E$ of a program $P$ is formally specified by the tuple $E = \langle A, \rightarrow^{po}, \rightarrow^{so}, \rightarrow^{sw}, \rightarrow^{hb}\rangle$, where:

- $A$ is a set of *actions*. Each action is a tuple $a = \langle t, k, u \rangle$, where:
  - $t$ is the thread identifier.
  - $k$ specifies the *kind* of action, which in particular includes reads and writes to shared memory. A read of shared memory location $x$ is denoted by $\text{read}(x)$ and a write to a shared memory location $x$ is denoted by $\text{write}(x)$. Also pertinent to this chapter are the *synchronization* actions $\text{lock}(x)$ and $\text{unlock}(x)$ that, respectively, are used to enter and exit a critical section coordinated through shared memory location $x$.
  - $u$ is a unique identifier of the action.
• $\text{po}$ is the program order relation over the set of unique identifiers and specifies a total order over actions performed by a single thread. Program order is consistent with the control flow and dataflow semantics of program $P$. For each $\langle t_i, k_i, u_i \rangle, \langle t_j, k_j, u_j \rangle \in A$ such that $u_i \neq u_j$:
  
  - If $t_i = t_j$, then either $u_i \text{po} \rightarrow u_j$ or $u_j \text{po} \rightarrow u_i$.
  
  - If $u_i \text{po} \rightarrow u_j$ or $u_j \text{po} \rightarrow u_i$, then $t_i = t_j$.
  
  - If $t_i \neq t_j$, then $u_i \text{po} \rightarrow u_j$ and $u_j \text{po} \rightarrow u_i$.

• $\text{so}$ is the synchronization order relation over the set of unique identifiers and specifies a total order over synchronization actions performed by any thread. Furthermore, for $\langle t_i, k_i, u_i \rangle, \langle t_j, k_j, u_j \rangle \in A$ such that $u_i \neq u_j$:
  
  - If $k_i$ and $k_j$ are synchronization actions, then either $u_i \text{so} \rightarrow u_j$ or $u_j \text{so} \rightarrow u_i$.
  
  - If either $k_i$ or $k_j$ is not a synchronization action, then $u_i \text{so} \rightarrow u_j$ and $u_j \text{so} \rightarrow u_i$.

• $\text{sw}$ is the synchronizes-with relation over the set of unique identifiers that specifies a partial order over synchronization actions performed by any thread and is consistent with synchronization-order. The details of this relation are dependent on the memory consistency model. For example, the Java memory model [59, 74] defines synchronizes-with relations for $\text{lock}(x)$ and $\text{unlock}(x)$ actions. For each $\langle t_i, \text{unlock}(x), u_i \rangle, \langle t_j, \text{lock}(x), u_j \rangle \in A$:
  
  - If $u_i \text{so} \rightarrow u_j$, then $u_i \text{sw} \rightarrow u_j$.

• $\text{hb}$ is the happens-before relation over the set of unique identifiers and is the transitive closure of the program order and synchronizes-with relations.

An execution is said to contain a data race if and only if it contains a conflict which is not ordered by the happens-before relation, i.e., there exists $\langle t_i, k_i, u_i \rangle, \langle t_j, k_j, u_j \rangle \in A$ such
that $u_i \rightarrow^{hb} u_j$, $u_j \rightarrow^{hb} u_i$ and at least one of $k_i$ and $k_j$ is a write. A program $P$ is said to be *correctly synchronized* if and only if every execution $E$ of $P$ is free of data races.

The remainder of this chapter focuses on correctly-synchronized programs. Formal reasoning about programs containing data races is both complex and beyond the scope of this dissertation. Furthermore, modern languages such as Java and C++ emphasize correct synchronization [19, 74]. In fact, the behavior of programs containing data races is undefined by C++ and Posix threads [18]. Only the Java memory model defines the behavior of programs containing data races, and even its current definition inadvertently prohibits some optimizations that the model was intended to permit [102].

By restricting the discussion to correctly-synchronized programs, a valid execution $E$ of program $P$ must be equivalent to some sequential execution of $P$ [2]. Therefore, the happens-before relation of $E$ defines the order of all memory conflicts and, thereby, the value that each memory read returns as well as the final value of each memory location.

The behavior of an execution can be characterized by the value that each memory read returns and the final value of each memory location. The execution of a correctly-synchronized program can therefore be characterized by the happens-before relation between memory conflicts of an execution. Intuitively, two executions have the same behavior if each conflicting read observes the same write and the last conflicting write to each memory location is the same. This yields the following definition of equivalent executions.

**Definition** Let $A$ be the set of actions for a valid execution $E$ of correctly-synchronized program $P$, and let $A'$ be the set of actions for a valid execution $E'$ of $P$. Executions $E$ and $E'$ are *equivalent* if and only if $A = A'$ and for all $(t_i, k_i, u_i), (t_j, k_j, u_j) \in A$ such that $k_i$ and $k_j$ conflict, there exists $(t_i, k_i, u_i), (t_j, k_j, u_j) \in A'$ such that:

- If $u_i \rightarrow^{hb} u_j$ in $E$, then $u_i \rightarrow^{hb} u_j$ in $E'$.

- If $u_j \rightarrow^{hb} u_i$ in $E$, then $u_j \rightarrow^{hb} u_i$ in $E'$.
Similarly, two programs have the same behaviors if each possible execution in one program has an equivalent execution in the other program. This concept is useful when considering the correctness of transforming one program representation into another.

**Definition** Program $P_i$ is said to be equivalent to $P_j$ if and only if:

- For any execution $E_i$ of $P_i$ an equivalent execution $E_j$ exists of $P_j$
- For any execution $E_j$ of $P_j$ an equivalent execution $E_i$ exists of $P_i$.

In some cases, transforming one program into another may require introducing new statements. To simplify the comparison of these programs the following notion of statement equivalence is provided.

**Definition** Statement $s_x$ is said to be equivalent to statement $s_y$ if and only if $s_x$ and $s_y$ produce the same kind of action when executed.

### 8.2 Formal Specification of Atomic Regions

Next, the formal notation is extended to include the atomic region abstraction by defining the statements `aregion_begin`, `aregion_end`, and `aregion_abort`. These statements (and actions by the same name) correspond to the hardware atomicity primitives introduced in Chapter 2.1. Together, they provide atomic execution semantics and define an atomic region. The statements `aregion_begin`, `aregion_end` and `aregion_abort` are collectively called atomic statements and the related actions are called atomic actions. Note that hardware may implicitly abort an atomic region, and therefore an `aregion_abort` action may not always correspond to a statement in the program.

Informally, any path through an atomic region must include an `aregion_begin` action followed by either an `aregion_end` or `aregion_abort` action, and atomic regions are never nested. Paths through an atomic region that include an `aregion_end` are said to commit.
Paths through an atomic region that include an \textit{aregion\_abort} are said to \textit{abort} and do not include any action other than the preceding \textit{aregion\_begin} (hardware discards all intervening speculative actions). In addition, all actions outside of the atomic region appear to occur either before or after the atomic region.

The following constraints are placed on the program order relation of any valid execution of a program $P$, which contains atomic regions. Let $E$ be a valid execution of $P$, with a set of actions $A$:

- For each $\langle t_b, \text{aregion\_begin}, u_b \rangle \in A$, there exists some $\langle t_b, k_c, u_c \rangle \in A$ such that $u_b^\text{po} \rightarrow u_c$ and either $k_c = \text{aregion\_end}$ or $k_c = \text{aregion\_abort}$. Furthermore, for any $\langle t_b, k_i, u_i \rangle \in A$:
  
  - If $k_c = \text{aregion\_end}$ and $u_b^\text{po} \rightarrow u_i \rightarrow u_c$, then $k_i$ is not an atomic action.
  - If $k_c = \text{aregion\_abort}$, then $u_b^\text{po} \rightarrow u_i$ or $u_i^\text{po} \rightarrow u_c$.

- For each $\langle t_c, k_c, u_c \rangle \in A$ such that $k_c = \text{aregion\_end}$ or $k_c = \text{aregion\_abort}$, there exists some $\langle t_c, \text{aregion\_begin}, u_b \rangle \in A$ such that $u_b^\text{po} \rightarrow u_c$. Furthermore, for any $\langle t_c, k_i, u_i \rangle \in A$, if $u_b^\text{po} \rightarrow u_i \rightarrow u_c$, then $k_i$ is not an atomic action.

For convenience, the notation $[u_b, u_c]$ is used to denote an atomic region as specified by actions $u_b$ and $u_c$, which satisfy the constraints above. For any action $\langle t_b, k_i, u_i \rangle \in A$ where $u_b^\text{po} \rightarrow u_i \rightarrow u_c$, $u_i$ is further said to be \textit{contained} in the atomic region $[u_b, u_c]$. Note that an atomic region which aborts does not contain any actions.

The following constraints are placed on the synchronizes-with relation of any valid execution of a program $P$, which contains atomic regions. For any valid execution $E$ of $P$, with a set of actions $A$:

- Let $\langle t_i, k_i, u_i \rangle, \langle t_j, k_j, u_j \rangle \in A$ such that $k_i$ conflicts with $k_j$ and $t_i \neq t_j$. Also let $u_i$ be contained in the atomic region specified by $[u_b, u_c]$, where $\langle t_b, \text{aregion\_begin}, u_b \rangle, \langle t_b, \text{aregion\_end}, u_c \rangle \in A$. Then, atomic execution orients the conflict by imposing either $u_j^\text{sw} \rightarrow u_b$ or $u_c^\text{sw} \rightarrow u_j$ on the synchronizes-with relation.
• Let \( \langle t_i, k_i, u_i \rangle, \langle t_j, k_j, u_j \rangle \in A \) such that \( k_i \) and \( k_j \) are synchronization actions. Also let \( u_i \) be contained in the atomic region specified by \([u_b, u_e]\), where \( \langle t_b, \text{aregion}_\text{begin}, u_b \rangle \), \( \langle t_b, \text{aregion}_\text{end}, u_e \rangle \in A \). Then the following synchronizes-with relations are imposed by atomic execution semantics:

- If \( u_i \xrightarrow{\text{sw}} u_j \), then \( u_e \xrightarrow{\text{sw}} u_j \).
- If \( u_j \xrightarrow{\text{sw}} u_i \), then \( u_j \xrightarrow{\text{sw}} u_b \).

### 8.3 Constraints on Atomic Region Formation

Assume a program \( P \), which does not contain atomic regions, is transformed into program \( P' \) by inserting \texttt{aregion}_\texttt{begin} and \texttt{aregion}_\texttt{end} statements to form atomic regions. For any behavior that is speculatively removed from an atomic region in \( P' \) an \texttt{aregion}_\texttt{abort} statement is also inserted. This transformation is called \textit{atomic region formation} (see Chapter 2.2).

Informally, atomic region formation is only valid if the transformed program is equivalent to the original program. First, the transformed program must contain atomic regions which satisfy the constraints of Section 8.2. Second, any execution of the transformed program must have an equivalent execution in the original program. Third, any execution of the original program must have an equivalent execution in the transformed program.

To form an atomic region, a compiler must first select a \textit{region} of a program to transform. A region \( R \) is a subset of a program and is similarly defined by the tuple \( R = \langle S, CFG, DFG \rangle \). Before specifying constraints on region selection, definitions for an entry and exit of a region are provided.

**Definition** Let \( s_x \) be an immediate predecessor of \( s_y \) in program \( P \). If \( s_x \notin R \) and \( s_y \in R \), then \( s_y \) is an \textit{entry} of region \( R \).

**Definition** Let \( s_y \) be an immediate successor of \( s_x \) in \( P \). If \( s_x \in R \) and \( s_y \notin R \), then \( s_x \) is an \textit{exit} of region \( R \).
To satisfy the program order constraints of Section 8.2, a compiler must first select a region that does not already contain any atomic statements (atomic regions are never nested). The region should also specify a connected set of statements because a disjoint set of statements can trivially be split into separate regions. Furthermore, the region must have well defined entry and exit points such that any path through the region will encounter exactly one entry and one exit statement.

In order to be considered for atomic region formation, a region $R$ selected from program $P$ must satisfy the following constraints. Let $CFG_P$ be the control flow graph of the statements in $P$ and $CFG_R$ be the control flow graph of the statements in $R$, then:

- For any $s_x \in R$, $s_x$ is not an atomic statement.

- $CFG_R$ is a weakly connected subgraph of $CFG_P$, i.e., if each edge of $CFG_R$ were undirected, then $CFG_R$ would be a connected graph.

- Let $s_x, s_y \in R$ such that $s_x$ and $s_y$ are both entries of $R$. If $s_x$ is a predecessor of $s_y$ in $P$, then for each path $\tau$ from $s_x$ to $s_y$ in $CFG_P$, there exists some $s_n \in R$ such that $s_n$ is an exit of $R$ and $s_n$ is a vertex in $\tau$.

- Let $s_x, s_y \in R$ such that $s_x$ and $s_y$ are both exits of $R$. If $s_x$ is a predecessor of $s_y$ in $P$, then for each path $\tau$ from $s_x$ to $s_y$ in $CFG_P$, there exists some $s_n \in R$ such that $s_n$ is an entry of $R$ and $s_n$ is a vertex in $\tau$.

Once a compiler has selected an appropriate region, it can then transform it into an atomic region. To do so, an `aregion_begin` must be placed at each entry of the region, an `aregion_end` must be placed at each exit, and `aregion_abort` statements must be inserted on any control flow path that has been speculatively removed. Also, if the atomic region aborts then execution must be resumed on an alternate control flow path that is equivalent to executing the original region prior to atomic region formation.
Atomic region formation is formally specified as follows. Given a region \( R \) in program \( P \), which satisfies the constraints on region selection, a compiler may transform \( P \) into \( P' \) by generating an atomic region \( R_a \) that satisfies the following constraints:

- For each \( s_x \in P \), \( s_x \in R_a \) if and only if \( s_x \in R \).

- For each \( s_y \in R \) such that \( s_y \) is an entry of \( R \), there exists a statement \( s_b = \text{aregion\_begin} \) where:
  - \( s_b \in R_a \) and \( s_b \) is the only immediate predecessor of \( s_y \).
  - For each \( s_x \in P \) such that \( s_x \) is an immediate predecessor of \( s_y \) in \( P \), \( s_x \) is an immediate predecessor of \( s_b \) in \( P' \). Thus, \( s_b \) replaces \( s_y \) as an entry of \( R_a \).
  - An alternate control flow path is provided as the parameter to \( s_b \) (see Section 2.1). This alternate control flow path contains statements that are equivalent to a control flow path starting at entry \( s_y \) of \( R \) and continuing through some exit of \( R \). It will be executed by hardware if the atomic region starting at \( s_b \) aborts.

- For each \( s_x \in R \) such that \( s_x \) is an exit of \( R \), there exists a statement \( s_e = \text{aregion\_end} \) where:
  - \( s_e \in R_a \) and \( s_e \) is the only immediate successor of \( s_x \).
  - For each \( s_y \in P \) such that \( s_y \) is an immediate successor of \( s_x \) in \( P \), \( s_y \) is an immediate successor of \( s_e \) in \( P' \). Thus, \( s_e \) replaces \( s_x \) as an exit of \( R_a \).

- If any behaviors are speculatively removed from a path through \( R_a \), a statement \( s_a = \text{aregion\_abort} \) is used to terminate the path such that \( s_a \in R_a \) is used to terminate the path such that \( s_a = \text{aregion\_abort} \). Furthermore, \( s_a \) has no successors in the control flow graph of \( P' \).

It is now possible to prove that atomic region formation satisfies the constraints on program order from Section 8.2. In particular, each control flow path through the atomic
region must include an `aregion_begin` followed by either an `aregion_end` or `aregion_abort`, and atomic regions are never nested.

**Lemma 8.1.** Suppose atomic region formation is used to transform region $R$, which satisfies the constraints on region selection, into an atomic region $R_a$. Then, any control flow path through $R_a$ satisfies the atomic region constraints on the program order relation (see Section 8.2).

*Proof.* By the constraints on region selection, any control flow path through $R$ will pass through a single entry and a single exit. By the constraints on atomic region formation, every entry of $R_a$ is an `aregion_begin` and every exit of $R_a$ is an `aregion_end`. Likewise, any control flow path through $R_a$ will pass through a single entry and at most a single exit (a path may terminate at an `aregion_end`).

Therefore, every control flow path through $R_a$ starts with an `aregion_begin` and ends with either an `aregion_end` or `aregion_abort`. Likewise, every control flow path includes exactly one `aregion_begin` and exactly one of either `aregion_end` or `aregion_abort`.

Thus, any control flow path through $R_a$ satisfies the program order constraints on atomic region execution. □

Provided that atomic region formation satisfies the program order constraints of Section 2.2, the next two proofs show that atomic regions do not introduce new executions to a transformed program and that forming an atomic region containing a single statement does not eliminate any executions from a transformed program.

**Lemma 8.2.** Suppose atomic region formation is used to transform a correctly-synchronized program $P$ into program $P'$. Then, for any valid execution $E'$ of $P'$, an equivalent execution $E$ exists of $P$.

*Proof.* Suppose $E'$ is a valid execution of $P'$, then the happens-before relation of each conflict in $E'$ can be shown to exist in some execution $E$ of program $P$.  

134
It is necessary to demonstrate that transforming \( P \) into \( P' \) does not introduce new happens-before relations to any conflicts. The proof first considers atomic regions that abort and then considers atomic regions that commit. In both cases, let \( A \) be the set of actions for execution \( E \) and let \( A' \) be the set of actions for execution \( E' \). For the set of actions \( A' \) of \( E' \), \( A' \supseteq A \) and the actions in \( A' - A \) are the atomic actions produced by the atomic regions in \( P' \).

Let \( \langle t_b, \text{region}_\text{begin}, u_b \rangle, \langle t_b, \text{region}_\text{abort}, u_a \rangle \in A' \) such that \([u_b, u_a]\) specifies an atomic region that aborts. By the constraints on atomic region execution, \([u_b, u_a]\) does not contain any actions and, therefore, does introduce any new happens-before relations. By the constraints on atomic region formation, the actions following \( u_a \) in program order must be equivalent to an execution of the original region in \( P \).

Suppose there exists a pair of conflicting actions \( \langle t_i, k_i, u_i \rangle, \langle t_j, k_j, u_j \rangle \in A' \) such that \( t_i \neq t_j \), \( k_i \) conflicts with \( k_j \) and \( u_i \xrightarrow{\text{hb}} u_j \). Let \( \langle t_b, \text{region}_\text{begin}, u_b \rangle, \langle t_b, \text{region}_\text{end}, u_e \rangle \in A' \) such that \([u_b, u_e]\) specifies an atomic region that commits. Because \( P \) is correctly synchronized, there must also exist \( \langle t_i, k_{i+1}, u_{i+1} \rangle, \langle t_j, k_{j-1}, u_{j-1} \rangle \in A' \) such that \( u_i \xrightarrow{\text{po}} u_{i+1} \xrightarrow{\text{sw}} u_{j-1} \xrightarrow{\text{po}} u_j \).

There are two cases to consider:

1. A synchronizes-with relation introduced by \( u_e \) such that:
   - If \( u_i \) is contained in \([u_b, u_e]\), then \( u_i \xrightarrow{\text{po}} u_e \xrightarrow{\text{sw}} u_j \)
   - If \( u_{i+1} \) is contained in \([u_b, u_e]\), then \( u_{i+1} \xrightarrow{\text{po}} u_e \xrightarrow{\text{sw}} u_{j-1} \)

2. A synchronizes-with relation introduced by \( u_b \) such that:
   - If \( u_j \) is contained in \([u_b, u_e]\), then \( u_i \xrightarrow{\text{sw}} u_b \xrightarrow{\text{po}} u_j \)
   - If \( u_{j-1} \) is contained in \([u_b, u_e]\), then \( u_{i+1} \xrightarrow{\text{sw}} u_b \xrightarrow{\text{po}} u_{j-1} \)

Each of these relations enforce \( u_i \xrightarrow{\text{hb}} u_j \) but, clearly, this relation was already enforced by \( u_{i+1} \xrightarrow{\text{sw}} u_{j-1} \). Simply removing all atomic actions from \( E' \) yields a valid execution \( E \) of \( P \).
The same argument holds for any pair of conflicts and therefore for any execution of \( P' \) an equivalent execution exists of \( P \).

**Lemma 8.3.** If atomic region formation is used to transform a region \( R \) containing a single statement \( s_i \) of correctly-synchronized program \( P \) into program \( P' \), then program \( P' \) is equivalent to program \( P \).

*Proof.* By Lemma 8.2, for any valid execution of \( P' \) an equivalent execution exists of \( P \). Therefore, it is only necessary to prove that for any valid execution \( E \) of \( P \) an equivalent execution \( E' \) exists of \( P' \).

Suppose \( E \) is a valid execution of \( P \), then the happens-before relation of each conflict in \( E \) can be shown to exist in some execution \( E' \) of program \( P' \). Let \( A \) be the set of actions for execution \( E \) and \( A' \) be the set of actions for execution \( E' \). For the set of actions \( A' \) of \( E' \), \( A' \supseteq A \) and the actions in \( A' - A \) are the atomic actions produced by the atomic regions in \( P' \).

Let \( \langle t_i, s_i, u_i \rangle \in A \) be an action produced by the single statement \( s_i \) contained in \( R \). Suppose there exists some \( \langle t_{i-1}, k_{i-1}, u_{i-1} \rangle, \langle t_{i+1}, k_{i+1}, u_{i+1} \rangle \in A' \) such that \( u_{i-1} \xrightarrow{hb} u_i \xrightarrow{hb} u_{i+1} \).

Suppose \( s_i \) is executed in an atomic region which commits. Then, there exists \( \langle t_i, \text{aregion}_\text{begin}, u_b \rangle, \langle t_i, \text{aregion}_\text{end}, u_e \rangle \in A' \) such that \([u_b, u_e]\) specifies an atomic region and \( u_b \xrightarrow{po} u_i \xrightarrow{po} u_e \). This implies \( u_i \) is contained in \([u_b, u_e]\). Let \( \langle t_{i-1}, k_{i-1}, u_{i-1} \rangle, \langle t_{i+1}, k_{i+1}, u_{i+1} \rangle \in A' \) such that \( k_i \) conflicts with \( k_{i-1} \) and \( k_{i+1} \), and \( u_{i-1} \xrightarrow{hb} u_i \xrightarrow{hb} u_{i+1} \). Atomic region constraints may impose \( u_{i-1} \xrightarrow{sw} u_b \) or \( u_e \xrightarrow{sw} u_{i+1} \), both of which are trivially consistent with \( u_{i-1} \xrightarrow{hb} u_i \xrightarrow{hb} u_{i+1} \) in \( E' \).

Suppose \( s_i \) is executed in an atomic region which aborts. Then, there exists \( \langle t_i, \text{aregion}_\text{begin}, u_b \rangle, \langle t_i, \text{aregion}_\text{abort}, u_a \rangle \in A' \) such that \([u_b, u_a]\) specifies an atomic region and \( u_b \xrightarrow{po} u_a \xrightarrow{po} u_i \) (*i.e.*, an equivalent action for \( s_i \) occurs in the alternate code executed after the atomic region abort). Because \( u_i \) is not contained in \([u_b, u_a]\), \( u_{i-1} \xrightarrow{hb} u_i \xrightarrow{hb} u_{i+1} \) is trivially consistent with \( E' \).
Therefore, for any execution $E$ of $P$ an equivalent execution exists for $E'$ of $P'$. Thus, program $P'$ is equivalent to program $P$.

As a result, atomic region formation can be used to form atomic regions containing a single statement without changing the behavior of a program. The next section discusses reorderings allowed for these atomic regions, and the first two proofs show that an atomic region can be incrementally grown from a single statement, albeit with constraints, without changing the behavior of a program.

### 8.4 Reorderings Permitted with Atomic Regions

This section will prove that statements can be reordered both across atomic region boundaries and within atomic regions. Statement reordering is important for two reasons. First, the previous section merely proved that it is safe to form single-statement atomic regions. By enabling a compiler to reorder statements into an atomic region, larger atomic regions are supported. Second, compiler optimizations (such as partial redundancy elimination and dead code elimination) can generally be thought of as reordering redundant statements to the same location and then deleting a redundant statement. The remaining proofs assume the following definition for statement reordering.

**Definition** Two adjacent statements $s_x$ and $s_y$ in program $P$ are said to be reordered in program $P'$ if their relative control flow order is reversed in $P'$. These statements are further said to be safely reordered if and only if:

- $s_x$ and $s_y$ are reordered
- $s_x$ and $s_y$ belong to the same thread in program $P$
- Reordering $s_x$ and $s_y$ does not violate any control or data dependence in program $P$
• Reordering $s_x$ and $s_y$ does not violate the atomic region formation constraints of Section 8.3.

Stated informally, two statements can be safely reordered if doing so would not violate the single-thread semantics of a program. However, a safe reordering might violate the memory ordering constraints of a program and could still be incorrect.

Therefore, the following proofs also consider statements which may have inter-thread dependences (control or data). The next two definitions introduce the terms in-synchronize and out-synchronize to simplify discussion of these statements. Informally, a statement in-synchronizes with an atomic region if the statement must execute before some statement in the atomic region because of an inter-thread dependence. Likewise, a statement out-synchronizes with an atomic region if the statement must execute after some statement in the atomic region because of an inter-thread dependence.

**Definition** Suppose $R_a$ is an atomic region in program $P$, statement $s_i$ is a predecessor of $R_a$, $s_i \notin R_a$, and statement $s_j \in R_a$. Further suppose $E$ is a valid execution of $P$ with a set of actions $A$, where $\langle t_i, s_i, u_i \rangle, \langle t_j, s_j, u_j \rangle, \langle t_j, \text{aregion}\_\text{begin}, u_b \rangle \in A$ such that $u_i$ identifies the action produced by $s_i$, $u_j$ identifies the action produced by statement $s_j$, and $u_b$ identifies the entry of the atomic region containing $u_j$. Consider a (potentially invalid) execution $E'$ which is derived by removing $u_i \xrightarrow{\text{po}} u_b$ from the program order relation of $E$ (if it exists). Then, statement $s_i$ is said to in-synchronize with $R_a$ if and only if $u_i \xrightarrow{\text{hb}} u_j$ in some $E'$ of $P$.

**Definition** Suppose $R_a$ is an atomic region in program $P$, statement $s_j$ is a successor of $R_a$, $s_j \notin R_a$, and statement $s_i \in R_a$. Further suppose $E$ is a valid execution of $P$ with a set of actions $A$, where $\langle t_i, s_i, u_i \rangle, \langle t_i, \text{aregion}\_\text{end}, u_e \rangle, \langle t_j, s_j, u_j \rangle \in A$ such that $u_i$ identifies the action produced by $s_i$, $u_e$ identifies the exit of the atomic region containing $u_i$, and $u_j$ identifies the action produced by statement $s_j$. Consider a (potentially invalid) execution $E'$ which is derived by removing $u_e \xrightarrow{\text{po}} u_j$ from the program order relation of $E$ (if it exists).
Then, statement $s_j$ is said to out-synchronize with $R_a$ if and only if $u_i \xrightarrow{\text{hb}} u_j$ in some $E'$ of $P$.

Provided with these definitions, it is possible to prove that statements can be reordered into an atomic region (either for optimization purposes or to grow the region) by considering statements that in-synchronize or out-synchronize with an atomic region. In particular, statements that are predecessors of an atomic region and do not in-synchronize with the atomic region can be reordered into the atomic region. The corollary is also true, statements that are successors of an atomic region and do not out-synchronize with the atomic region can be reordered into the atomic region.

**Lemma 8.4.** Suppose $P$ is a correctly synchronized program which contains an atomic region $R_a$. Reordering statement $s_i$, which is an immediate predecessor of $R_a$, with an $\text{aregion}_\text{begin}$ of $R_a$ results in a transformed program $P'$. Program $P'$ is equivalent to program $P'$ if $s_i$ does not in-synchronize with $R_a$.

**Proof.** By Lemma 8.2, adding statements to an atomic region does not introduce any new executions to $P'$ that were not possible in $P$. Therefore, it is only necessary to prove that $P'$ does not prohibit executions which were possible in $P$.

Suppose there exists a valid execution $E$ of $P$ which is prohibited in $P'$. Let $\langle t_i, s_i, u_i \rangle, \langle t_i, \text{aregion}_\text{begin}, u_b \rangle \in A$ such that $u_i$ identifies the action produced by executing $s_i$, and $u_b$ identifies the action produced by an entry of region $R_a$. In $P'$, statement $s_i$ is reordered after the $\text{aregion}_\text{begin}$ and therefore $u_i \xrightarrow{\text{po}} u_b$ in any execution $E'$ of program $P'$ with a set of actions $A' = A$.

Therefore, if execution $E$ is not equivalent to any valid execution of $E'$ of program $P'$ there must exist some $\langle t_{i+1}, k_{i+1}, u_{i+1} \rangle \in A$ such that $u_i \xrightarrow{\text{hb}} u_{i+1}$ in $E$ but $u_i \xrightarrow{\text{hb}} u_{i+1}$ in any $E'$. This implies that statement $s_i$ in-synchronizes with $R_a$, which is a contradiction.

**Lemma 8.5.** Suppose $P$ is a correctly synchronized program which contains an atomic region $R_a$. Reordering statement $s_i$, which is an immediate successor of $R_a$, with an
aregion\_end of $R_a$ results in a transformed program $P'$. Program $P'$ is equivalent to program $P'$ if $s_i$ does not out-synchronize with $R_a$.

**Proof.** As with Lemma 8.4, it is only necessary to prove that $P'$ does not prohibit executions which were possible in $P$.

Suppose there exists a valid execution $E$ of $P$ which is prohibited in $P'$. Let $\langle t_i, s_i, u_i \rangle$, $\langle t_i, \text{aregion\_end}, u_b \rangle \in A$ such that $u_i$ identifies the action produced by executing $s_i$ and $u_e$ identifies the action produced by an exit of region $R_a$. In $P'$, statement $s_i$ is reordered before the \text{aregion\_end} and therefore $u_e \xrightarrow{\text{po}} u_i$ in any execution $E'$ of program $P'$, with a set of actions $A' = A$.

Therefore, if execution $E$ is not equivalent to any valid execution of $E'$ of program $P'$ there must exist some $\langle t_{i-1}, k_{i-1}, u_{i-1} \rangle \in A$ such that $u_{i-1} \xrightarrow{\text{hb}} u_i$ in $E$ but $u_{i-1} \xrightarrow{\text{hb}} u_i$ in any $E'$. This implies that statement $s_i$ out-synchronizes with $R_a$, which is a contradiction.

Although Lemma 8.4 and Lemma 8.5 enable some statements to be reordered into an atomic region, a compiler is still constrained. Limiting reorderings to statements that do not in-synchronize with or out-synchronize with an atomic region is overly prohibitive for languages with strict memory models, such as x86.

Hardware can enable a relaxation of these constraints, specifically by permitting any safe reordering of a statement into an atomic region. However, this requires hardware to abort an atomic region if a speculative memory conflict is detected during atomic region execution. For example, the hardware assumed by the implementations of Chapter 6 and Chapter 7 monitors the cache coherence protocol and eagerly detects violations of atomic region execution. If a violation is detected, hardware implicitly aborts the atomic region and resumes execution in a non-atomic version of the region. However, proving this claim requires formal reasoning about speculative actions and is left as future work.

Nonetheless, for languages with weak memory models (or assuming that the previous claim holds), the next proof demonstrates that statements can be freely reordered within
an atomic region. Specifically, safely reordering two statements which are contained in an atomic region is guaranteed to be correct.

**Lemma 8.6.** If program $P$, which contains atomic regions, is transformed into program $P'$ by safely reordering two adjacent statements inside an atomic region of $P$, then $P'$ is equivalent to $P$.

*Proof.* Let $A$ be the set of actions for some valid execution $E$ of program $P$. Suppose that $\langle t_i, s_i, u_i \rangle, \langle t_i, s_j, u_j \rangle \in A$ correspond to adjacent statements $s_i$ and $s_j$ in $P$ such that $s_i$ and $s_j$ can be safely reordered. Further suppose $u_i$ and $u_j$ are contained in an atomic region specified by $[u_b, u_e]$, where $\langle t_i, \text{aregion.begin}, u_b \rangle, \langle t_i, \text{aregion.end}, u_e \rangle \in A$, and $u_i \xrightarrow{\text{po}} u_j$.

For any $\langle t_n, k_n, u_n \rangle \in A$ such that $k_n$ conflicts with $s_j$, the atomic region constraints require that:

- Either $u_n \xrightarrow{\text{sw}} u_b$ which implies $u_n \xrightarrow{\text{hb}} u_j$
- Or $u_e \xrightarrow{\text{sw}} u_n$ which implies $u_j \xrightarrow{\text{hb}} u_n$

An execution $E'$ of program $P'$ exists, which differs from $E$ only that $s_i$ and $s_j$ are reordered and $u_j \xrightarrow{\text{po}} u_i$ in $E'$. Atomic region constraints require that:

- Either $u_n \xrightarrow{\text{sw}} u_b$ which implies $u_n \xrightarrow{\text{hb}} u_j$
- Or $u_e \xrightarrow{\text{sw}} u_n$ which implies $u_j \xrightarrow{\text{hb}} u_n$

Thus for any conflict contained in an atomic region in $E'$, a conflict with an equivalent happens-before relation can be found in some execution $E$ of $P$. The converse argument holds and $P$ is therefore equivalent to $P'$.

As a result, a compiler can ignore memory model constraints when reordering statements within an atomic region. This may provide additional reordering freedom and thereby enable additional optimization (see Chapter 3.2). This is a key benefit of the atomic region abstraction.
However, in regions of a program which do not contain synchronization statements a
compiler is already free of memory model constraints. If atomic regions are inserted into such
a program region, new constraints should not be introduced. Lemma 8.4 and Lemma 8.5
already proved this for optimizations which logically reorder statements into an atomic
region. The following proofs will show that similar freedom exists for optimizations which
logically reorder statements out of an atomic region.

Note that the definition of safe reordering must be refined to include control specula-
tion inside of an atomic region. The following proofs momentarily assume that no control
speculation occurs within an atomic region, but this restriction will be lifted by the refined
definition of safe reordering provided in Section 8.4.1.

Lemma 8.7. Let $P$ be a program with atomic regions. Also let $P$ contain an atomic
region $R_a$ such that no statement in-synchronizes with $R_a$. If program $P$ is transformed
into program $P'$ by safely reordering the aregion_begin statement of $R_a$ with the immediate
successor of the aregion_begin, then program $P'$ is equivalent to program $P$.

Proof. Let $A$ be the set of actions of an execution $E$ of program $P$. Assume that $s_b =$
aregion_begin and $s_e = $ aregion_end are an entry and exit of atomic region $R_a$, respectively,
and that statement $s_i \in R_a$ is an immediate successor of $s_b$. Suppose $s_i$ can be safely
reordered with $s_b$ and no statement in-synchronizes with $R_a$.

Further suppose that $\langle t_i, s_i, u_i \rangle, \langle t_i, s_b, u_b \rangle, \langle t_i, s_e, u_e \rangle \in A$ are actions which correspond
to statements $s_i$, $s_b$ and $s_e$, respectively, and $u_i$ is contained in the atomic region $[u_b, u_e]$.

Let $\langle t_j, k_j, u_j \rangle \in A$ such that $t_j \neq t_i$, and $u_j \xrightarrow{hb} u_i$. Because no statement in-synchronizes
with $R_a$, there must be some $\langle t_i, k_{i-1}, u_{i-1} \rangle \in A$ such that $u_j \xrightarrow{hb} u_{i-1} \xrightarrow{po} u_b \xrightarrow{po} u_i$.

Suppose that the statements $s_b$ and $s_i$ have been reordered in program $P'$. Then in any
execution $E'$ of $P'$, $u_j \xrightarrow{hb} u_{i-1} \xrightarrow{po} u_i \xrightarrow{po} u_b \xrightarrow{po} u_i$ and $u_j \xrightarrow{hb} u_i$. Thus, execution $E'$ is equivalent to $E$.

The converse is also true by the same argument and program $P'$ is equivalent to program
$P$. \qed
Lemma 8.8. Let $P$ be a program with atomic regions. Also let $P$ contain an atomic region $R_a$ such that no statement out-synchronizes with $R_a$. If program $P$ is transformed into program $P'$ by safely reordering the \texttt{aregion\_end} statement of $R_a$ with the immediate predecessor of the \texttt{aregion\_end}, then program $P'$ is equivalent to program $P$.

Proof. Let $A$ be the set of actions of an execution $E$ of program $P$. Assume that $s_b = \texttt{aregion\_begin}$ and $s_e = \texttt{aregion\_end}$ are an entry and exit of atomic region $R_a$, respectively, and that statement $s_i \in R_a$ is an immediate predecessor of $s_e$. Suppose $s_i$ can be safely reordered with $s_b$ and no statement out-synchronizes with $R_a$.

Further suppose that $\langle t_i, s_i, u_i \rangle, \langle t_i, s_b, u_b \rangle, \langle t_i, s_e, u_e \rangle \in A$ are actions which correspond to statements $s_i$, $s_b$ and $s_e$, respectively, and $u_i$ is contained in the atomic region $[u_b, u_e]$.

Let $\langle t_j, k_j, u_j \rangle \in A$ such that $t_j \neq t_i$, and $u_i \xrightarrow{\text{hb}} u_j$. Because no statement out-synchronizes with $R_a$, there must be some $\langle t_i, k_{i+1}, u_{i+1} \rangle \in A$ such that $u_i \xrightarrow{\text{po}} u_e \xrightarrow{\text{po}} u_{i+1} \xrightarrow{\text{hb}} u_j$.

Suppose that the statements $s_i$ and $s_e$ have been reordered in program $P'$. Then in any execution $E'$ of $P'$, $u_e \xrightarrow{\text{po}} u_i \xrightarrow{\text{po}} u_{i+1} \xrightarrow{\text{hb}} u_j$ and $u_i \xrightarrow{\text{hb}} u_j$. Thus, execution $E'$ is equivalent to $E$. The converse is also true by the same argument and program $P'$ is equivalent to program $P$. \qed

The following two examples depict optimization opportunities which are enabled by the reorderings permitted by atomic region semantics. Figure 8.1 depicts an atomic region which contains an \texttt{unlock} statement. In the Java memory model, an \texttt{unlock} action synchronizes-with all subsequently executed \texttt{lock} actions, and, therefore, no statement in-synchronizes with the depicted atomic region but there could be a statement that out-synchronizes with the atomic region.

Lemma 8.6 enables reordering the statements within the atomic region. Because no statement in-synchronizes with the atomic region, Lemma 8.4 permits reordering a statement immediately preceding an atomic region with the \texttt{aregion\_begin}, which enables the elimination of the dead store in block A. Likewise, Lemma 8.7 permits a statement inside
Figure 8.1: Example reorderings in an atomic region that contains an unlock. (a) The atomic region contains a Java unlock statement. Therefore, there could be a statement in the program that out-synchronizes with the region. (b) This prevents reordering and, thereby, optimization of the dead store in block B but otherwise exposes optimization opportunity by permitting other reorderings.

The atomic region to be reordered with the aregion_begin. This further enables the elimination of the redundant load in block B. By Lemma 8.5, a statement that immediately follows and does not out-synchronize with an atomic region may be reordered into the atomic region. No synchronization statement exists on the control flow path from the aregion_end to the redundant load in block C. Therefore, the redundant load does not out-synchronize with the atomic region and can be eliminated.

Figure 8.2 depicts a similar atomic region, except that the atomic region contains a lock statement instead of an unlock statement. In the Java memory model, any previously executed unlock action synchronizes with a lock action. Therefore, no statement out-synchronizes with the depicted atomic region but there could be a statement that in-synchronizes with the atomic region.

Because no statement out-synchronizes with the atomic region, Lemma 8.5 enable the elimination of the redundant load in block C. Furthermore, Lemma 8.8 enables a statement inside the atomic region to be reordered with the aregion_end. This further enables the...
Figure 8.2: Example reorderings in an atomic region that contains a lock. (a) The atomic region contains a Java lock statement. Therefore, there could be a statement in the program that in-synchronizes with the region. (b) This prevents reordering and, thereby, optimization of the redundant load in block B, but otherwise exposes optimization opportunity by permitting other reorderings. For example, the dead store from Figure 8.1 can be optimized.

elimination of the dead store in block B. By Lemma 8.4, a statement that immediately precedes and does not in-synchronize with an atomic region may be reordered into the atomic region. No synchronization statement exists on the control flow path from the dead store in block A to the aregion_begin. Therefore, the dead store does not in-synchronize with the atomic region and can be eliminated.

8.4.1 Control Speculation and Safe Reordering

The definition of safe reordering of program statements has thus far ignored control-speculation within an atomic region, but the primary motivation for the atomic region abstraction is the simplicity in which it enables a compiler to employ control speculation.

Figure 8.3 depicts an example atomic region in which a highly-biased branch is converted into an assert statement. As described in Chapter 2.2, an assert statement is not considered a control operation by a compiler and it therefore enables a speculative relaxation of control
Figure 8.3: **Control dependences restrict optimization across atomic region boundaries.** (a) The store in basic block B is control dependent on the null-check in basic block A. (b) When an atomic region is created from these basic blocks the null-check is converted into an assert operation. The assert operation relaxes control dependences, but it is still invalid to hoist the store out of the atomic region. (c) An example of an invalid hoisting optimization is shown.

Therefore, statements which were previously control-dependent on a branch which has been converted into an `assert` must not escape an atomic region. For example, Figure 8.3(c) depicts an invalid reordering which should not be permitted. Because a branch in the atomic region has been converted into an assert, a store appears control independent and has been reordered out of the atomic region. This transformation is clearly incorrect because the store has become non-speculative even though it should depend on the successful commit of the atomic region. Even if the assert fires and the region is aborted the store will still execute and thereby expose a speculative value. This example motivates the following addition to the definition of control dependence.

**Definition** If statement \( s_x \) is control dependent on statement \( s_y \) and both \( s_x \) and \( s_y \) are contained in an atomic region, then statement \( s_x \) is also control dependent on every `aregion_begin` of the atomic region. This control dependence persists even if \( s_y \) is converted into an `assert` statement.

This refined definition prevents a control-speculative statement from being reordered...
out of an atomic region, because the definition of safe reordering requires that control dependencies be preserved. Reordering a control-speculative statement within an atomic regions is still permitted, however. The following proves the correctness of reordering control-speculative statements within an atomic region.

**Lemma 8.9.** Let program $P$ contain a statement $s_x$ which is control dependent on $s_y$, where $s_x$ and $s_y$ are adjacent and contained in the same atomic region. If $P$ is transformed into program $P'$ by converting $s_y$ into $s'_y$, where $s'_y = assert$, and by safely reordering $s_x$ and $s'_y$, then program $P'$ is equivalent to program $P$.

*Proof.* Let $E'$ be a valid execution of program $P'$. Suppose that the atomic region containing $s_x$ and $s'_y$ does not abort in $E'$, then $s'_y$ did not misspeculate. By Lemma 8.6, there must exist some execution $E$ of program $P$ such that $E'$ is equivalent to $E$.

Suppose that the atomic region $R_a$ containing $s_x$ and $s'_y$ does abort in $E'$, then the executed atomic region will contain no actions in $E'$. By the constraints on atomic region formation, the actions following the aborted region in program order must be equivalent to an execution of the original region from which $R_a$ was formed. In this case, an equivalent execution of the same region exists in $E$ (by Lemma 8.3, Lemma 8.4, and Lemma 8.5).

Thus, for any execution $E'$ of $P'$ there exists an equivalent execution $E$ of $P$. The converse argument holds, and program $P'$ and $P$ are equivalent. \qed
Chapter 9

Conclusion

Despite the growing emphasis on parallel processing and multiprocessors, I believe that improving single-thread performance remains paramount. The emphasis on parallelism is a practical response to limits in process scaling and the difficulty of building larger-window, wider-issue or more deeply pipelined processors. Certainly, recovering historical improvements in single-thread performance is unlikely. Nonetheless, opportunities remain.

This dissertation is evidence of this simple fact. It demonstrates that hardware and software can play a complementary role in exposing these additional opportunities. Hardware primitives designed for use by software can provide for power and complexity efficient improvements in performance. Specifically, the hardware atomicity primitive provides the means by which the atomic region abstraction enables software to more easily exploit speculative optimization opportunities.

I have detailed many of the performance opportunities that compiler writers pursue as well as the obstacles which often prevent them from being exploited. Speculative compiler optimization attempts to exploit many of these opportunities in the common case, and this concept is not new. Likewise, others have proposed hardware to enable software to more effectively employ speculative optimization.

The atomic region abstraction is but a furthering of this line of thinking, albeit a demonstrably effective one. It has been incorporated into two different dynamic optimization frameworks and shown to improve performance both in simulation and on a real hardware system. Practical solutions have been presented for all the major technical issues ranging from IR representation to misspeculation management.
That said, important questions are left unanswered. I have demonstrated the utility of the atomic region abstraction in improving single-thread performance, but it is unclear if performance potential remains. A limit study or design space exploration of different design choices, such as region formation, would help determine if additional potential opportunity exists.

In Chapter 6, the atomic region abstraction is prototyped in a JIT compiler and is shown to improve performance. The baseline hardware is an out-of-order x86 processor extended with the hardware atomicity primitive. The combination of type-based alias analysis [34] and out-of-order scheduling hardware means that hardware for memory alias detection is not strictly needed. However, aliases may still exist and alias detection hardware could further enable the removal of partially redundant memory operations.

In Chapter 7, I argue that the atomic region and superblock abstractions are synergistic. This is certainly true for a VLIW processor because of its need for effective static scheduling. However, other static scheduling techniques might make for a better match. For example, wavefront scheduling [9] is amenable to arbitrary region shapes and could allow for a more effective implementation of the atomic region abstraction.

Even more fundamentally, the atomic region abstraction has only been explored in the context of dynamic optimization frameworks. Whether or not it is viable in a static compiler is not known. To be sure, some runtime features would be necessary for misspeculation monitoring and disabling unprofitable asserts. However, other requirements might also exist and other issues may need to be solved.

These questions are left for derivative work. It is my hope that others might further the understanding of the atomic region abstraction and its performance potential. Barring that, I hope that the success of the atomic region abstraction encourages others to continue the pursuit of single-thread performance improvements.
References


151


[57] Intel Corporation. Intel 64 architecture memory ordering white paper, August 2007.


