Efficient Compiler and Runtime Support for Serializability and Strong Semantics on Commodity Hardware

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Aritra Sengupta, B.Tech, M.S.

Graduate Program in Computer Science and Engineering

The Ohio State University

2017

Dissertation Committee:

Michael D. Bond, Advisor

Atanas Rountev

P. Sadayappan

Abstract

Parallel software systems often have non-deterministic behavior and suffer from reliability issues. A major deterrent in the use of sound and precise bug-detection techniques is the impractical run-time overhead and poor scalability of such analyses. Alternatively, strong memory models or program execution models could achieve software reliability. Unfortunately, existing analyses and systems that provide stronger guarantees in ill-synchronized programs are either expensive or they compromise on reliability for performance. Building efficient runtime support for enforcing strong program semantics remains a challenging problem because of the overhead incurred to implement the complex mechanisms in the underlying system. The overhead could be because of restriction on reordering of operations in the compiler or the hardware, instrumentation cost in enforcing stronger program semantics through a dynamic analysis, the cost of concurrency control or synchronization mechanisms typically required in such an analysis, and the cost of leveraging specific hardware constructs. Researchers have proposed strong memory models based on *serializability of regions of code*—where regions of code across threads execute in isolation with each other. The run-time cost incurred to enforce strong memory models based on serializability stems from the unbounded number of operations that are monitored by an analysis. This thesis solves this critical problem of providing region serializability at reasonable overheads and yet demonstrating its efficacy in eliminating erroneous behavior from ill-synchronized programs. *Providing serializability of code regions using compiler*

techniques and dynamic analysis while leveraging both generic and specialized commodity hardware, at low overheads, across concurrent programs having diverse characteristics—can make enforcement of strong semantics for real-world programs practical and scalable. This work presents a memory model based on bounded regions, called *dynamically bounded region serializability* (DBRS) and establishes the memory model theoretically—contrasting it with other memory models. It provides a novel technique, called EnfoRSer, that enforces the memory model practically, leveraging not only dynamically bounded, but statically bounded, intra-procedural, acyclic regions of code. This technique utilizes a lightweight conflict-detection mechanism and strong compiler transformations to enforce DBRS at low overheads. Further, an additional dynamic analysis based on prior work demonstrates empirically DBRS's potential to eliminate real-world bugs. After establishing the benefits of DBRS, this thesis provides a mechanism to reduce the instrumentation overhead of DBRS enforcement by proposing a novel technique that efficiently hybridizes per-access locks and per-region locks based on the results of *static data race detection*, offline profiling, and a cost-benefit model. This technique demonstrates that for programs with low inter-thread dependences and high density of memory accesses, a hybridized synchronization scheme can enforce DBRS more efficiently. The first two techniques require advanced compiler transformations and per-access instrumentation for inter-thread conflict detection and resolution. These techniques also suffer from high overheads in programs with significant conflicts—pertaining to expensive conflict resolution. To purge the limitations of the first two techniques, this work provides a more practical and efficient solution to DBRS enforcement. This approach, called Legato, overcomes these complexities and uses widely available commodity *hardware transactional memory* (HTM) to enforce DBRS efficiently preserving all the benefits of the memory model. Applying commodity HTM to

enforce DBRS showcases a significant challenge—*prohibitive "startup" and "tear-down" cost of hardware transactions*. Further, the *best-effort* commodity HTM poses several challenges such as unknown abort location, transactions aborting for reasons other than capacity and conflict, and requiring an expensive fallback to non-speculative execution. We apply a novel technique that merges several regions into a single transaction in order to amortize the cost of starting and stopping transactions. Our approach resolves the other challenges with an algorithm based on control theory principles that enforces DBRS with lower run-time costs than our prior approaches. This technique eliminates the complexities of the previous two approaches and provides a simpler and a more efficient solution to DBRS enforcement. Overall, this dissertation, proposes a memory model with strong benefits in today's concurrent software and evaluates three distinct mechanisms to enforce the memory model—each improving on the limitations of the other. Our evaluation demonstrates that low-overhead enforcement of DBRS is indeed feasible in today's commodity hardware that eliminates several bugs in erroneous real-world programs; hence, advancing the state of the art.

Dedicated to my parents

Acknowledgments

My experience in graduate school over the last six years has been memorable and rewarding. I'll cherish it for the rest of my life. The journey of a Ph.D. student is often challenging and uncertain. To complete the journey, it requires strong support from individuals involved professionally and personally with the student. Prior to joining the Ph.D. program at The Ohio State University and during the course of my Ph.D., there have been several individuals who have influenced me positively and motivated me when I needed them the most. I would like to acknowledge them all. However, if I fall short of recognizing them all here, I apologize, and hopefully, you will understand and forgive the omission.

First, I would like to thank my advisor, Dr. Michael D. Bond. He has been extremely supportive as a mentor. Besides strong technical feedback and help on all aspects of this dissertation, I have learnt considerably from his exemplary leadership skills in driving a research group—where he led by example. During the course of my Ph.D., besides research, I have drawn inspiration from his work-life balance and his decision-making skills to name a few. I consider myself fortunate to have had him as my advisor since I have grown as a person during this association. I would like to thank him for showing faith in me and believing in my abilities.

I have had the wonderful opportunity to collaborate with Dr. Milind Kulkarni from Purdue University in all my research projects. I would like to thank him for excellent

technical insights and feedback on the projects that are part of this dissertation. I am still learning from the way he articulates research—separating the grain from the chaff.

Dr. Atanas Rountev and Dr. P. Sadayappan have been on my candidacy and dissertation committee. Further, I have taken several research and curricular courses with them. I would like to acknowledge their advice, guidance, and feedback. I'll take this opportunity to thank Dr. Radu Teodorescu and other professors in our department. I must appreciate the support of the staff members of the department whose efforts make the place a conducive environment for research and teaching.

It always helps to discuss ideas, concepts, and techniques related to research with your group members. I had numerous helpful discussions with the PLaSS group members. I must thank them: Swarnendu Biswas, Man Cao, Meisam Fathi-Salmi, Jipeng Huang, Jake Roemer, Minjia Zhang, and Rui Zhang. I must acknowledge the great company of my friends in Columbus: Rajaditya Mukherjee, Aniket Chakrabarti, Dhrubojyoti Roy, Swarnendu Biswas, Soumya Dutta, Bortik Bandyopadhyay, Subhrakanti Chakraborty, and Sayak Roychowdhury. They have always been by my side when I needed them. Some of my old friends have been like family. I would like to thank Shoumik Roychoudhury, Soumya Roy, Shinjini Datta, Saugata Bhattacharya, Tanmoy Sinha, Shounock Chakravarti, Paulami Basu, Ambarish Ghosh, and others who have often helped me to remain happy and composed. I must thank my colleagues from work who eventually turned into friends—Sujoy Chakraborty and Pralay Gupta—who always supported and encouraged me.

During my undergraduate days, I was fortunate to have worked with some excellent teachers and mentors. I would like to thank Dr. Margret Anouncia, Dr. Vijaya Sherly, and Dr. S. Sumathy from Vellore Institute of Technology University (VIT) for their encouragement and guidance that eventually helped me to pursue graduate studies. I must

also acknowledge my old friends from VIT who have always had a positive influence on me: Arnab Mallick, Arnab Dutta, Shubhadip Dasgupta, Aarthi Mandava, and Debolina Bandyopadhyay.

I am grateful to my family members for their endless love and encouragement. My sister and brother-in-law, Arundhati Sengupta and Soumyatanu Gupta respectively, always had faith in me and wished the best for me. My uncle and aunt, Asish Sengupta and Gopa Sengupta respectively, have played an important role in my life and were always loving and caring. I must thank my cousins I grew up with, Rupa Sengupta and Kaushik Sengupta. They have always wished the best for me and motivated me to fulfill my aspirations. I would also like to thank everyone I mentioned above for taking care of my parents while I have been thousands of miles away from home working on my Ph.D. I miss my grandparents who are not with us anymore and I wish they were there to see me defend my Ph.D. dissertation.

Finally, I would like to thank my parents. Having experienced the value of pursuing research and a doctoral degree themselves, they have inspired me in pursuing doctoral studies. Their own lifestyles have guided and influenced me in pursuing intellectual satisfaction over materialistic choices. It would have been virtually impossible for me to overcome the hurdles and remain focused during the Ph.D. program without their relentless support and sacrifice. I am indebted to them for everything including their tolerance and patience that they have shown towards me.

The research presented in this dissertation has been supported by grants from National Science Foundation: CCF-1421612, CSR-1218695, and CAREER-1253703.

Vita

May 2017	Ph.D., Computer Science and Engineering, The Ohio State University, USA.
Dec 2015	M.S., Computer Science and Engineering, The Ohio State University, USA.
May 2008	B.Tech, Computer Science and Engineering, Vellore Institute of Technology University, India.
June 16, 1986	Born: Calcutta (now Kolkata), India.

Publications

Research Publications

Aritra Sengupta, Man Cao, Michael D. Bond, and Milind Kulkarni. Legato: Endto-End Bounded Region Serializability Using Commodity Hardware Transactional Memory. In *International Symposium on Code Generation and Optimization (CGO'17)*, pages 1-13, Feb 2017.

Man Cao, Jake Roemer, Aritra Sengupta, and Michael D. Bond. Prescient Memory: Exposing Weak Memory Model Behavior by Looking into the Future. In *International Symposium on Memory Management (ISMM'16)*, pages 99-110, June 2016.

Man Cao, Minjia Zhang, Aritra Sengupta, and Michael D. Bond. Drinking from Both Glasses: Combining Pessimistic and Optimistic Tracking of Cross-Thread Dependences. In *Principles and Practice of Parallel Programming (PPoPP'16)*, pages 20:1-20:13, March 2016.

Aritra Sengupta, Man Cao, Michael D. Bond and Milind Kulkarni. Toward Ecient Strong Memory Model Support for the Java Platform via Hybrid Synchronization. In *International Conference on Principles and Practices of Programming on the Java Platform (PPPJ'15)*, pages 65-75, September 2015.

Aritra Sengupta, Swarnendu Biswas, Minjia Zhang, Michael D. Bond and Milind Kulkarni. Hybrid Static–Dynamic Analysis for Statically Bounded Region Serializability. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*, pages 561-575, March 2015.

Swarnendu Biswas, Jipeng Huang, Aritra Sengupta, and Michael D. Bond. DoubleChecker: Efficient Sound and Precise Atomicity Checking. In *Programming Langangauges Design and Implementation (PLDI'14)*, pages 28–39, June 2014.

Michael D. Bond, Milind Kulkarni, Man Cao, Minjia Zhang, Meisam Fathi Salmi, Swarnendu Biswas, Aritra Sengupta, and Jipeng Huang. Octet: Capturing and Controlling Cross-Thread Dependences Efficiently. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'13)*, pages 693-712, March 2013.

Fields of Study

Major Field: Computer Science and Engineering

Studies in:

Programming Languages and Software Systems	Prof. Michael D. Bond
High Performance Computing	Prof. P. Sadayappan
Theory and Algorithms	Prof. Anish Arora

Table of Contents

List of Tables

Table **Page**

List of Figures

Chapter 1: Introduction

Parallelism in the form of concurrency is now an integral part of performance-critical software. Limitations on fabrication cost, power efficiency, and heat dissipation on a single processor led to the growth of *multicore* systems. Multicore systems have spurred the need to write programs that efficiently utilize the parallel hardware. However, writing correct parallel programs remains notoriously difficult. Synchronization is at the heart of writing correct parallel programs but is expensive if not used judiciously. Programmers' primary challenge is to strike a balance between performance and correctness. Studies show that even well-tested software has reliability issues [LPSZ08, GN08].

In spite of years of research in concurrency bug detection and companies spending a significant amount in testing software, real-world software have bugs [ACJL13, U.S04, LPSZ08]. Typically, parallel programs have synchronization bugs like *data races*. A data race essentially means a *concurrent* and *conflicting* access to a shared memory location by at least two threads. The existence of data races has serious implications on the semantic guarantees in the program. Existing language memory models provide *strong guarantees* in absence of data races. However, in *racy* (with data races) programs, commonly used memory models offer very weak guarantees to avoid any overhead in providing the guarantees. It is hard to write programs without data races; therefore, mandating detection of data races in production or at least during software testing. Unfortunately, the state-of-the-art

data race detection is expensive and inappropriate for adoption because of unreasonable overhead [BCM10, FF09, BZBL15, ZLJ16, FF13, BCZ+17]. According to researchers, Adve and Boehm [AB10], *"Semantics for programs that contain data races seem fundamentally difficult, but are necessary for concurrency safety and debuggability. We call upon software and hardware communities to develop languages and systems that enforce data-race-freedom, and co-designed hardware that exploits and supports such semantics."* To alleviate this problem and provide stronger guarantees in presence of data races, researchers have proposed analyses and systems to automatically *enforce data-race-freedom*—enforce semantics equivalent to a data-race-free execution even in a program which by default is racy. Several runtime support systems and analyses have been developed to make the underlying program execution more restrictive in order to avoid data races or its detrimental effects on program behavior. Analyses and systems spread across strong memory models [OCFN13, MSM+10, SMN+11, LCS+10], alternative synchronization and programming models like *transactional memory* (TM) [HM93, HF03, DSS10a], and deterministic program execution systems [RDB99, OAA09, VLW+11, LWV+10]. The practicality of these systems still remains a concern especially in production systems where overheads and limited scalability in these analyses deter their use.

Common language memory models in mainstream programming languages like C++ and Java provide weak semantics in presence of data races [AH90, MPA05, BA08]. Therefore, these memory models are essentially *"weak"* with respect to semantics in racy programs. The pervasiveness of software systems afflicted with data races generated the need for *"stronger"* semantics—provided by *strong memory models*. However, strong memory models usually incur the cost of enforcement. Typically, the cost comes from restriction on reordering of operations in the compiler or the hardware [MSM+11, LNGR12, LNG10],

instrumentation cost in enforcing stronger semantics through an analysis [BZBL15], and concurrency control mechanisms typically required in such analyses [OCFN13]. The state-of-the-art techniques that guarantee strong program semantics on practical hardware are impractically expensive [BZBL15, OCFN13] or are incompatible with commodity systems—involving custom hardware [LCS+10, LNGR12, AQN+09, CTMT07, LDSC08, AAK+05, BDLM07, MBM+06, YBM+07, BGH+08, PAM+08, CCC+07]. Therefore, we believe, research should be focused on systems that achieve a balance between semantic guarantees and performance—to attain an acceptable tradeoff for practical systems—deployable on commodity hardware and ready for use.

1.1 Motivation

A *data race* is a pair (at least) of *conflicting* accesses—at least one access is a write, on shared memory where the accesses are not ordered by a *happens-before* relation—a partial order that is the union of program order and synchronization order [Lam78]—implying *concurrent* accesses. Multithreaded programs are often poorly synchronized in pursuit of performance, leaving data races in the programs that generate ill-defined behaviors under current memory models. State-of-the-art data race detection is expensive [BCM10, FF09, BZBL15, ZLJ16, FF13, BCZ+17] making developers averse to using them. The common language memory models, provide strong semantics only in absence of data races [BA08, MPA05, AB10]; hence are weak. Programs with data races need stronger guarantees. *Sequential consistency* (SC), provides stronger semantics, by ensuring threads execute strictly in program order of individual operations. Unfortunately, SC is restrictive on both the compiler and hardware [SS88, LNG10, MSM+11, LNGR12]. Instead of restricting individual operations, stronger guarantees can be provided at the

level of groups of operations—enforcing execution equivalent to a sequential ordering of code regions, that is enforcing *region serializability*. However, existing solutions that execute groups of operations atomically, rely on complex custom hardware that vendors are reluctant to manufacture [CTMT07, AQN+09, LDSC08, LCS+10, MSM+10, SMN+11] Replication-based solutions require additional cores for running duplicate instances of the application [OCFN13].

Other solutions, can define new programming models (in contrast to memory consistency models) avoiding lock-based synchronization—that are susceptible to synchronization errors in multithreaded programs—and instead introduce alternatives like transactional memory (TM). However, after years of research, *software transactional memory* remains expensive [ZHCB15, DSS10b, SMAT+07, WMvP+09]. *Hardware transactional memory* is available in commercial processors but it is limited in several respects—size of transactions, instructions supported in transactions, the cost of *starting and ending* transactions [RB13, YHLR13]—preventing its widespread adoption. Further, transactional programs can have weak semantics appertaining to the interaction of transactions and non-transactions (accesses outside transactions) [SMAT+07, DS09, HLR10].

This dissertation attempts to provide a solution to the problem of guaranteeing reliable program behavior in presence of data races at reasonable run-time overheads—*nearly an order of magnitude lesser than state-of-the-art detection of data races*—on commodity hardware.

1.2 Problem Statement

Figure 1.1(a) shows an example program with ill-synchronized critical sections where there is a data race on the boolean variable *flag* which is meant to signal initialization to the

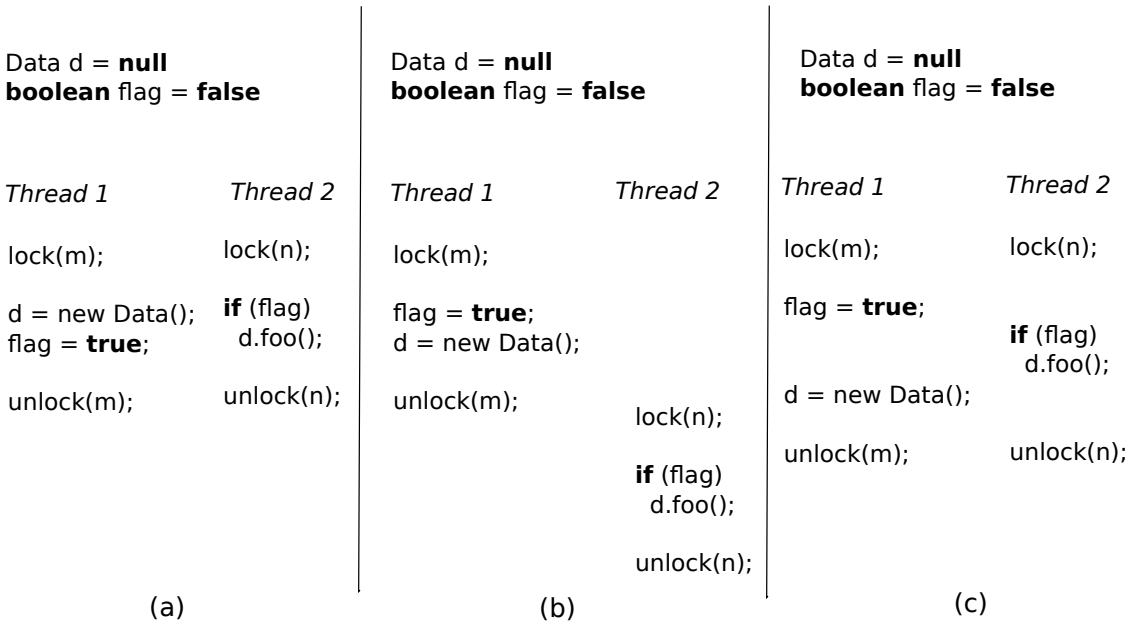


Figure 1.1: (a) An example program with ill-synchronized locks having a data race. (b) Transformed program but a region-conflict-free execution leading to a serializable execution (c) Transformed program with detrimental effect of the data race.

thread dereferencing the object reference *data*. Figure 1.1(b) shows an execution where the compiler reordered the two instructions on Thread1. However, the code regions execute on each thread without conflict, leading to a serializable execution. Figure 1.1(c) shows the detrimental effect of the compiler reordering where the object can be accessed in an uninitialized or partially initialized state when the regions conflict. If the runtime enforces program execution semantics shown in Figure 1.1(b), then the program is guaranteed to behave equivalently to an execution corresponding to a correctly synchronized, data-race-free program.

Providing strong semantics to real-world concurrent software is challenging. Solutions typically require custom hardware [AQN+09, LCS+10, LNGR12, CTMT07, LDSC08] or additional resources like processor cores [OCFN13] and pure software solutions incur high

overheads [BZBL15, LCFN12, OAA09, LWV+10]. In this dissertation, we explore novel techniques to enforce strong program semantics through hybrid static–dynamic analysis and runtime support for ill-synchronized concurrent software. The techniques explored do not require any customized hardware and yet incur reasonable overheads. The key insight behind this work is the observation that a large percentage of concurrency bugs can be removed by guaranteeing atomicity of *bounded* (limited number of dynamic instructions) code regions and favorably—atomicity of bounded regions can be provided efficiently at reasonable overheads. The efficiency could come from bounding the code regions carefully, employing hybrid static–dynamic analyses, and leveraging commercially available hardware in order to practically accomplish the enforcement of strong semantics in concurrent software. *This dissertation defines a new memory model for bounded region serializability, provides compiler and runtime solutions to efficiently enforce the memory model on commodity hardware, and demonstrates substantial bug elimination coverage of the memory model across diverse real-world software.*

The techniques proposed in this work enable atomic execution of code regions at reasonable overheads. Analyses developed in this work, demonstrate a good cost-benefit balance such that the enforced underlying memory model eliminates concurrency bugs, achieves good scalability, and yet incur moderate overheads—all on commercial hardware.

1.3 Contributions and Outline

In the following, we briefly outline the work presented in this dissertation and highlight the contributions of the thesis. This dissertation is organized as follows: we discuss background and motivation on strong memory consistency models, strong program execution semantics, techniques to reduce overheads of analyses and systems, and background

on commercial transactional hardware in **Chapter 2**. The following paragraphs give an overview of each chapter.

Programmers find it notoriously hard to avoid writing programs with data races. Existing memory consistency models on commodity hardware are either relaxed—incurring low run-time cost and allowing ill-defined behaviors in presence of data races [AH90], or they are stronger—allowing fewer erroneous behaviors in racy programs, but at the cost of prohibitive run-time overhead [OCFN13]. Hence, our work targets providing stronger guarantees to programs with data races, on commodity hardware and at reasonable run-time cost. **Chapter 3** proposes a memory model that provides atomicity of code regions. It restricts program behaviors to eliminate concurrency bugs without programmer intervention. The memory model is designed to strike a balance between the benefits of strong program semantics and the costs of providing those guarantees. The new memory model is called *dynamically bounded region serializability* (DBRS) that enforces serializability (atomicity) of bounded regions.

Chapter 4 describes a hybrid static–dynamic analysis called *EnfoRSer* that provides end-to-end support for DBRS. This technique requires the bounded regions to not only be dynamically bounded—*every region executing a limited number of operations*, but also statically bounded—*every static instruction executing at most once* in a region, to enable compiler transformations. Hence, EnfoRSer enforces *statically bounded region serializability* (SBRS), an instance of DBRS. EnfoRSer demonstrates that SBRS provides an excellent balance of practicality and strength: it is strictly stronger than SC, yet we show that the bounded nature of the regions offers opportunities for static–dynamic analysis to enforce end-to-end SBRS with reasonable performance on commodity systems. The key to EnfoRSer's operation is the interaction between static compiler transformations and the

runtime system. Our evaluation shows that EnfoRSer incurs reasonable run-time cost and yet eliminates several concurrency bugs that are exposed under weaker memory models.

A substantial amount of research has focused on the synchronization cost in analyses that provide stronger guarantees to programs. It is essential to reduce the cost of synchronization between threads [BKC+13, CZSB16] but a considerable overhead in these analyses comes from the cost of instrumenting each and every memory access in the program [FF13, BKC+13]. **Chapter 5** of this dissertation presents a way to ameliorate the instrumentation cost of an analysis that enforces SBRS. We enforce SBRS differently at even lower overheads than EnfoRSer. This technique called *EnfoRSer-H*, presented in **Chapter 5**, targets the reduction in instrumentation cost—the checks of ownership of locks at each memory access. EnfoRSer-H uses profiling to apply coarse-grained per-region locks or fine-grained per-access locks or both to regions—based on region characteristics, in order to minimize the cost of instrumentation. Our evaluation shows that this hybrid approach outperforms EnfoRSer presented earlier, for applications with low inter-thread conflicts and with a high density of accesses per region, and performs similar to EnfoRSer for other kinds of programs.

Finally, in **Chapter 6**, we present a novel approach that overcomes the limitation of the two prior techniques. This approach called *Legato* uses commodity *hardware transactional memory* (HTM) to avoid complex compiler transformations and heavy-weight instrumentation of EnfoRSer and EnfoRSer-H, and enforces DBRS at lower overheads. HTM as available in commercial processors poses several challenges that make its adoption non-trivial in enforcing DBRS. Legato uses a novel approach using control theoretic principles, that exploits HTM, to efficiently enforce DBRS.

This dissertation proposes novel solutions that provide atomicity of bounded code regions at low overheads—that justifies the practicality—of our proposed strong memory model based on serializability. The hallmark of serializability based memory models is that they allow several optimizations inside the regions—amortizing the overhead to provide the serializability. The thesis demonstrates efficient enforcement of strong program semantics in standard lock-based programs by eliminating several concurrency bugs at moderate overheads. This dissertation highlights multiple forms of atomicity enforcement techniques that lower synchronization and instrumentation overheads and proposes novel ways to utilize commercial transactional hardware in order to enforce strong program semantics. We demonstrate that our work successfully eliminates memory consistency and atomicity bugs in real-world software.

Chapter 7 describes related work relevant to strong memory consistency models and techniques to efficiently implement static and dynamic analysis. **Chapter 8** covers ideas that might augment the DBRS memory model or improve the techniques enforcing the memory model, further, in terms of efficiency and semantics. In **Chapter 9**, we conclude our dissertation, discussing the contribution and impact of our work and summarizing each chapter with respect to the overall context.

Chapter 2: Background and Motivation

This chapter describes closely related work for subsequent chapters in the dissertation. First, we discuss the background and related work on memory models and the need for stronger memory models with relatively low overhead, which motivates our work discussed in **Chapter 3** and **Chapter 4**. Second, we discuss related work on reducing overhead of dynamic analyses. This discussion provides background for our work discussed in **Chapter 5**. Third, we discuss background on *hardware transactional memory* (HTM) and present the basic semantics and programmability with a commercial HTM—which our work leverages in **Chapter 6**.

2.1 Essential Terminologies

Here we discuss some essential terminologies as part of background knowledge, essential to follow the contents in the remainder of this chapter.

Serializability. *Serializability* is a guarantee on groups of operations on objects modified by read and write accesses. It guarantees that the effect of execution of the groups of operations, is equivalent to a serial order of the groups (e.g. one at a time).

Program order. It is the union of each program thread's sequential order of operations. Each thread must obey the sequential semantics defined by the programming language.

Synchronization order. In a programming language that defines synchronization operations, a synchronization operation might order with—*synchronize with* another—for example, a release of a lock synchronizes with an acquire of the same lock. This order is a subset of all synchronization steps in the program.

2.2 Memory Model

A *memory model* defines the possible values for a load from shared memory [AB10]. This dissertation is concerned with language memory models that must be enforced *end-to-end*, i.e., must be enforced by the compiler and hardware with respect to the original source-level program—in contrast to hardware memory models, which only guarantee behaviors with respect to the compiled program.

An execution has a *data race* if two accesses are *conflicting* (they access the same shared, non-synchronization variable, and at least one is a write) and not ordered by the *happens-before relation*, a partial order that is the union of program and synchronization order [Lam78].

Generally speaking, memory models—including all of the memory models we discuss here—provide strong semantic guarantees for *data-race-free* (DRF) executions. In particular, DRF executions provide serializability of *synchronization-free regions* (SFRs) [AB10, AH90, LCS+10]. However, memory models differ in the guarantees they provide for *racy* (non-DRF) executions.

Strong memory model. *A strong memory model* is one that allows a read operation r to only return the value of a write operation w such that w is the last write operation across all threads—that is a sequentially consistent (refer Section 1.1) execution, and under such a model, the set of program execution behaviors permitted is a strict subset of the set of

program execution behaviors permitted under a weaker memory model. In the context of this dissertation, a strong memory model essentially provides the set of program execution behaviors as that of sequential consistency (SC) or a stronger model.

Weak memory model. *A weak memory model* allows a read operation r to return the value of a write operation w such that w is not the last write globally across all threads, meaning, w could be a *stale* value, from the past, or even a value that has not been written yet—a *future* value. In the context of this dissertation, a weak memory model is the one that allows a set of program execution behaviors not allowed by SC.

2.3 Need for Stronger Memory Models

Memory models were initially specified imprecisely by hardware manufacturers. However, they lacked clarity and synergy with compiler specifications. Later, common programming languages like C++ and Java specified the language memory models more formally—which the compiler and hardware should respect to guarantee the semantics of the memory model [BA08, MPA05]. Hardware continued to take benefit from optimizations like speculation on data accesses and control-flow [SI94, AFI+08, int12, SSA+11]. Compilers evolved along with the language memory models and the focus was to allow optimizations that are valid not only in sequential programs but also in the context of shared memory parallel programs. However, from the context of parallel computing, these optimizations, in the compiler or the hardware, make reasoning about the memory consistency model more complex. To avoid complexity of reasoning and reduce restriction on optimizations, initial language memory models and common language memory models of today, adopted a *data-race-free* model [AH90]—that provides well-defined semantics for programs without data races. Essentially these memory models are *weak*, with limited or no guarantees

for racy executions. However, with more software systems afflicted with hard to find data races [LPSZ08], researchers strongly believe in the need to enforce *data-race-freedom* in the underlying systems, in spite of the presence of data races [AB10]—meaning defining *strong* memory models.

2.3.1 Existing Weak Memory Models

If a program does not have a data race, that is every pair of conflicting accesses (i.e. accesses to the same shared variable where at least one is a write) is ordered by the happens-before relation [Lam78], then a weak memory model may be sufficient to provide the behavior that the programmer expects. However, programmers either find it hard to synchronize their programs correctly or leave data races in the programs intentionally for performance [TNGT08]. The *DRF0* memory model [BA08, MPA05, AH90] and it's variants including the Java memory model [MPA05] are most commonly used memory models but they provide very weak semantics in presence of data races.

The DRF0 memory model. Modern shared-memory languages such as Java and C++ use variants of the DRF0 memory model [BA08, MPA05, AH90]. DRF0 provides a strong guarantee for any DRF (data-race-free) execution. It guarantees *serializability* (i.e., atomicity) of *synchronization-free regions* (SFRs).

However, DRF0 provides weak guarantees for racy executions. As discussed, for DRF0 to provide stronger guarantees in racy executions, solutions are required across the compiler, runtime, and hardware—which potentially can complicate the model and incur overheads for both racy and non-racy executions. Therefore, DRF0 provides guarantees for only data-race-free executions, that necessitate careful considerations of synchronization operations and synchronization accesses only. The C++ memory model is close to the

DRF0 model [BA08] providing no guarantees for racy executions. Java tries to preserve the memory and type safety of racy executions [MPA05], resulting in a memory model that is a variant of DRF0, although the memory model has flaws [A08].

The Java memory model. The Java memory model (JMM) defines the behavior of shared memory accesses on programs running in a Java virtual machine (JVM) [MPA05]. Thus, the JMM applies not only to Java programs but also to programs written in other Java platform languages such as JRuby and Scala. DRF0 is not sufficient for Java or similar langauges that attempt to guarantee safety and security. Hence, Java attempts to provide safety and security in presence of untrusted code with data races. However, the task of mitigating damage in presence of data races, by providing memory and type safety, proved to be daunting [MPA05, A08]. In pursuit of enforcing safety, the JMM introduces a concept called *causality* that must be respected by compiler transformations in order to avoid so-called "out-of-thin-air results" (a concept that is not well defined but refers to executions that experts generally agree should be avoided and can violate memory and type safety) [MPA05, A08, BD14, FF10, CRSB16]. Although the JMM guarantees memory and type safety, it still permits unintuitive results such as loads of "stale" and "future" stored values, i.e., threads can observe other threads' events happening in different orders.

Furthermore, researchers have shown that common JVM compiler optimizations can violate JMM causality rules and permit out-of-thin-air results [A08]. Thus, JVMs do *not* actually enforce even the weak guarantees of the JMM, and no one knows how to guarantee causality (and thus memory and type safety) without severely limiting compiler optimizations [A08, BD14].

2.3.2 Existing Stronger Memory Models

Eliminating data races effectively and efficiently is a challenging, unsolved problem (e.g., [FF09, BLR02, NA07]). Moreover, developers often avoid synchronization in pursuit of performance, deliberately introducing data races that lead to unexpected, ill-defined behaviors. Adve and Boehm and Ceze et al. argue that languages and hardware must provide stronger memory models to avoid impossibly complex semantics [AB10, Boe12, CDLQ09]. According to Adve and Boehm: *"The inability to define reasonable semantics for programs with data races is not just a theoretical shortcoming, but a fundamental hole in the foundation of our languages and systems."* They *"call upon software and hardware communities to develop languages and systems that enforce data-race-freedom ..."* [AB10]. In pursuit of this overall goal, researchers have proposed stronger memory models at the language level, in the hardware, and end-to-end—meaning both across the compiler and the hardware.

Sequential consistency. *Sequential consistency* (SC) is an existing strong memory model where operations in each thread execute in program order but the global execution is an interleaving of these operations. Under SC, though reorderings are not allowed in a single processor, globally, all interleavings are still allowed. SC allows all interleavings in synchronization-free regions of code.

Providing end-to-end SC involves *limiting memory access reordering by both the compiler and hardware*, which slows programs and/or relies on custom hardware support that is unavailable [MSM+11, LNG10, AB10, RPA97, LNGR12, SNM+12, SS88, SFW+05]. Researchers have investigated the cost of its enforcement end-to-end, both in software and hardware [SFW+05, SS88, LNGR12, AQN+09]. It is difficult to provide SC end-to-end

cheaply without custom hardware support [LNGR12, AQN+09]. On existing compiler and hardware, end-to-end SC is hard to attain without incurring significant overhead, because *SC essentially requires restriction in reordering of individual operations—forbidding several compiler optimizations and limiting instruction-level-parallelism in the hardware.*

Though a significant body of research exists, that provides SC, the more fundamental concern is whether SC is sufficient to eliminate concurrency bugs. A key motivation behind the need for a new memory model is the observation that *SC is insufficient to eliminate many concurrency bugs even at the high expense of enforcing it*. Some operations that many programmers expect to execute atomically do not execute atomically even under SC (e.g., 64-bit integer accesses in Java; multi-access operations such as `x++` or `buffer[index++] = ...`). In addition, programmers assume atomicity of high-level operations and do not reason about interleavings of individual memory accesses [AB10]. Under SC, it is still difficult and unnatural for programmers to reason about all interleavings, and for program analyses and runtime support to deal with all interleavings. Researchers like Adve and Boehm argue that *"programmers do not reason about correctness of parallel code in terms of interleavings of individual memory accesses, and sequential consistency does not prevent common sources of concurrency bugs..."* [AB10].

Region serializability. Prior work introduces memory models based on *region serializability* (RS), in which regions of code appear to execute atomically and in program order [OCFN13, CTMT07, MSM+10, SMN+11, LCS+10, BZBL15, LDSC08, AQN+09]. RS is appealing not only because it has the potential to be stronger than SC, but also because it permits compiler and hardware optimizations within regions.

The strongest memory models are those based on serializability of *synchronization-free regions* (SFRs) [BZBL15, OCFN13, LCS+10]. Notably, SFR serializability provides the same semantics for *all* executions that DRF0 already provides for data-race-free executions, and it does not restrict compiler and hardware optimizations beyond DRF0's restrictions (essentially, no optimizations that cross synchronization operations). However, SFR-serializability-based memory models must support serializability of *unbounded* regions (since a thread may perform arbitrary work between two consecutive synchronization operations). Prior work provides serializability of unbounded regions: regions bounded either by synchronization or enclosed in transactions [BZBL15, OCFN13, LCS+10, HWC+04, AAK+05, BDLM07, MBM+06, YBM+07, BGH+08, PAM+08, CCC+07, HLR10]. Two main disadvantages in providing unbounded RS is as follows.

First, approaches that support serializability of unbounded regions, whether in software or hardware, tends to incur high costs and complexity. Software-based approaches slow programs by about 2X on average [OCFN13, BZBL15], although Ouyang et al.'s approach provides better performance if extra cores are available [OCFN13]. Hardware-based approaches can achieve low overhead by piggybacking on cache coherence events and extending cache metadata, but add significant complexity to support unbounded regions whose memory footprints exceed cache [LCS+10]. Similarly, *hardware transactional memory* (HTM) designs that support unbounded transactions incur high costs and complexity to track accesses and detect conflicts [HWC+04, AAK+05, BDLM07, MBM+06, YBM+07, BGH+08, PAM+08, CCC+07, HLR10].

Second, serializability of unbounded regions leads to potentially unintuitive progress guarantees (refer **Chapter 3** for more details) [OCFN13]). For example, the program below cannot terminate under serializability of SFRs, but (with the assumption of a fair scheduler)

is guaranteed to terminate under SC—as well as under serializability of *bounded* regions, discussed next.

Initially x = **false**, y = **false**

```
// Thread 1:          // Thread 2:
y = true;             x = true;
while (!x);           while (!y);
```

Bounded region serializability. A class of approaches that support *bounded RS*—in which each executed region performs a bounded amount of work—have the potential to avoid the cost and complexity associated with supporting unbounded regions. The focus of our work presented in this dissertation is on *end-to-end* bounded RS, which several existing approaches provide by enforcing atomic execution and restricting cross-region optimizations in both the compiler and hardware [MSM+10, SMN+11, AQN+09]. Nonetheless, all of these approaches choose region boundary points arbitrarily and thus cannot provide any guarantee beyond end-to-end SC. Since the memory model does not guarantee any particular region boundaries with respect to the program source, these approaches are in fact motivated by their authors as providing end-to-end SC. Furthermore, they *require custom hardware extensions to caches and coherence protocols* [MSM+10, SMN+11, AQN+09].

2.4 Reducing Overhead of Analyses

In this chapter, so far, we have discussed memory models and program semantics in presence of data races. Researchers have proposed several dynamic analyses techniques to build software systems that not only enforce strong memory models but spread across other related goals—like checking atomicity [FFY08, BHSB14], enabling multithreaded record and replay [LWV+10, VLW+11, BKC+15, HZD13, MMN09, HLZ10], and supporting other programming models using software transactional memory [SMAT+07, SATH+06,

ZHCB15, DSS10b]. These runtime support systems add instrumentation each and every program access to *control* inter-thread dependences. Typically, the instrumentation could be expensive—incurring a fixed, high cost at every memory access called as *instrumentation cost*. This fixed cost could be low and instead, the cost of detecting or controlling inter-thread dependences through the synchronization invoked by the instrumentation—called as *synchronization cost*, could be high. We discuss background on these two aspects of analyses and systems which check or enforce concurrency correctness properties like atomicity, determinism, and in the context of this dissertation—data-race-freedom.

2.4.1 Reducing Instrumentation Cost

Runtime analyses typically add instrumentation to the programs in order to detect or control interleaving between threads in a multithreaded program. For most analyses, if a program is free from data races then instrumenting synchronization points is sufficient. Since real-world software have data races, analyses need to capture dependences between accesses on different threads by instrumenting memory accesses. Data race detectors [FF09, BCM10, BZBL15], record and replay systems [BKC+15, MMN09, HZD13, HLZ10] or software transactional memory (STM) systems [ZHCB15, HG06, SMAT+07] typically instrument each and every memory access to account for these data races. These analyses therefore suffer from a significant instrumentation overhead. Prior work has used static analysis to reduce checks at program accesses. A significant work exists in this area with respect to reduction of instrumentation in dynamic data race detectors. Choi et al. uses intra-procedural analysis to identify accesses which are guaranteed to be race-free [CLL+02]. Flanagan and Freund proposes an effective and generic analysis to reduce checks on memory accesses that can be used for any dynamic data race detector [FF13].

Their analysis is on *lock-release-free* regions of code but can also operate intra-procedurally. These techniques are primarily for reducing the overhead of data race detectors and adapting them to reduce the overhead of analyses enforcing strong memory model behaviors can be challenging. In contrast, our work described in **Chapter 5**, targets reduction in instrumentation overhead based on limitations of our analysis that enforces the *dynamically bounded region serializability* (*DBRS*) memory model (**Chapter 3** and **Chapter 4**).

2.4.2 Reducing Synchronization Cost

Another key cost in these runtime systems, is the cost of synchronizing between threads to manage inter-thread dependences. Several runtime systems reduce synchronization overhead by using *biased* synchronization, that makes non-conflicting accesses fast by biasing the ownership of locks to a thread, but slows conflicting accesses substantially [BKC+13, RD06, KKO02, SGT96, vPG01, HG06]. If a thread owns a lock then no synchronization (i.e., inter-thread communication or an atomic operation) is required at an access. However, before another thread acquires the lock, it must *coordinate* with the owning thread to prevent a race condition since the owning thread can access without synchronization. Some techniques, including our prior work [BKC+13], support specialized *reader-writer* locks which preclude synchronization on read-shared accesses [SGT96, BKC+13, vPG01]. Biased locks perform very well for low-conflicting workloads but are expensive for high-conflicting workloads since resolving inter-thread conflicts require heavyweight synchronization. However, prior work tackles this issue by adaptively changing the locks from *biased or optimistic state* to *unbiased or pessimistic state* to avoid inter-thread communication [CZB14, CZSB16].

2.5 Transactional Memory

A multithreaded application often suffers from synchronization overhead to manage inter-thread dependences. Applications can use *fine-grained* locking with good scalability but higher proclivity towards incorrect synchronization; otherwise, they can use *coarse-grained* locking with inferior scalability but better programmability. To attain, *best of both worlds*, researchers have introduced transactional memory [HM93, RG01], where programmer specified code regions execute to appear atomically. Internally, the hardware executes code regions speculatively, rolling back the state before potential violations of atomicity. Therefore, programmers write code with coarse-grained reasoning but the hardware achieves the performance of fine-grained synchronization. Several proposed designs [HWC+04, AAK+05, BDLM07, MBM+06, YBM+07, BGH+08, PAM+08, CCC+07, HLR10] have influenced the growth of commercial hardware transactional memory [WGW+12, int12]. As background to **Chapter 6** we discuss an instance of commercial HTM—Intel's *Transactional Synchronization Extensions* (TSX).

2.5.1 Commodity Hardware Transactional Memory

There are instances of recent mainstream commodity processors supporting hardware transactional memory (HTM) to help developers exploit the potential of parallel hardware [int12, WGW+12, CCD+10, YHLR13]. In the context of this dissertation, we focus on Intel's Transactional Synchronization Extensions (TSX) since **Chapter 6** enforces the DBRS memory model using Intel's TSX. We efficiently leverage HTM (specifically Intel TSX) overcoming the limitations of the software-based techniques that are discussed in **Chapter 4** and **Chapter 5**. Intel TSX provides two interfaces: *hardware lock elision* (HLE) and *restricted transactional memory* (RTM). RTM allows programmers to provide

fallback actions when a transaction aborts, whereas HLE can only fall back to acquiring a lock [YHLR13, DKL+14].

RTM extends the ISA to provide XBEGIN and XEND instructions; upon encountering XBEGIN, the processor attempts to execute all instructions transactionally, committing the transaction when it reaches XEND. The XBEGIN instruction takes an argument that specifies the address of a fallback handler, from which execution continues if the transaction aborts. When a transaction aborts, RTM does the following: reverts all architectural state to the point when XBEGIN executed; sets a register to an error code that indicates the abort's proximate cause; and jumps to the fallback handler, which may re-execute the transaction or execute non-transactional fallback code. RTM flattens nested transactions [HLR10]. RTM also provides XABORT, which explicitly aborts an ongoing transaction, and XTEST, which returns whether execution is in an RTM transaction.

Chapter 3: A Memory Model Based on Bounded Region Serializability

In **Chapter 2**, we discussed the need for stronger memory models (refer section 2.3). In this chapter we propose a memory model that is based on providing serializability of code regions. It is challenging to provide atomicity to large synchronization-free regions of code at low overheads, especially in programs with data races. Several bugs in real-world code are pertaining to data races and do not involve the entire synchronization-free regions [LPSZ08, FF10, CRSB16, ZdKL+13]. Intuitively, providing atomicity of bounded code regions might eliminate several real-world concurrency bugs. In this dissertation, we investigate compiler and runtime techniques to provide strong semantics and a technique that leverages commodity hardware transactional memory to overcome the drawbacks of the software based approach.

Essentially, bounding the size of regions means bounding the number of dynamic operations executed in a code region. Bounding regions limits the cost of misspeculation if the regions are executed speculatively. Intuitively, to enable compiler techniques for enforcing atomicity of code regions, regions need to be bounded not only dynamically but statically as well. If regions are extremely small, several bad interleavings in buggy programs, will persist and can lead to unexpected behavior. In this chapter, we design a memory model that selects the code regions for atomicity enforcement carefully, *so as to achieve efficiency in performance and effectiveness in bug elimination*.

An discussed in **Chapter 2**, an alternative approach for providing *sequential consistency* (SC) is to provide *region serializability* (RS) of dynamically executed regions of code [LCS+10, MSM+10, SMN+11, GG91, AQN+09, EDLC+12, LDSC08]. RS can be advantageous since it allows compiler and hardware reordering of accesses within code regions. However, prior work that uses RS for SC relies on new hardware, adds high overhead, and/or *checks* SC instead of enforcing it, risking unexpected failures.

Recent work focuses on RS of regions bounded only by synchronization operations, a memory model we call *full-SFR RS* [LCS+10, OCFN13]. While full-SFR RS is clearly a strong memory model, it may not be realistic to enforce it—even with custom hardware support. Prior work relies on complex custom hardware and check (rather than enforce) full-SFR RS [LCS+10], or they rely on page-protection-based speculation which incurs high overhead on programs, nearly 100% or more (unless extra cores are available) [OCFN13].

In this chapter, we propose a memory model that is strictly stronger than SC and is intended to achieve a balance between strength and practicality of enforcement. In this chapter we describe the rationale behind selecting this memory model, the features of the memory model, potential advantages and disadvantages in comparison to other memory models, and progress guarantees under this memory model. In **Chapter 4**, we present an analysis called EnfoRSer that demonstrates the practical enforcement of this memory model on commodity hardware.

3.1 Dynamically Bounded Region Serializability

This work proposes a strong memory model that can be achieved with reasonable efficiency in commodity systems. Inspired by prior work that provides RS, our approach is

based on enforcing RS. Full-SFR RS is a very strong memory model, but it is inherently difficult to enforce without heavyweight support for conflict detection and speculation, which is expensive in software and complex in hardware. Therefore, our proposed memory model tries to balance strength and practicality.

Two main observations guide our decision. First, enforcing RS of *dynamically bounded* regions (i.e., each region executes a bounded number of operations) permits conservative conflict detection, since the cost of imprecision is bounded. Second, RS of not only dynamically but also *statically bounded* regions (i.e., an executing region executes each static instruction at most once) enables powerful static compiler transformations for making the regions atomic. Based on these insights, we propose a memory model called *dynamically bounded region serializability* (DBRS), that guarantees RS of dynamically bounded regions—regions demarcated at synchronization operations, loop back edges and method calls. Section 3.3 evaluates DBRS's potential in avoiding real-world program bugs. Here we discuss potential advantages and disadvantages of DBRS relative to other memory models. Theoretically DBRS can be enforced selecting other program points as region boundaries. However, the technique discussed in **Chapter 4** uses compiler transformations that are simplified when the regions are statically bounded—each region executes a static instruction once. **Chapter 4** describes this memory model as *statically bounded region serializability* (SBRS); *essentially the same memory model but with an additional constraint that the regions are statically bounded*.

3.2 Strength and Progress Guarantees of DBRS as a Memory Model

Potential advantages and disadvantages. DBRS makes software more reliable than under DRF0-based models. It not only provides SC, thus avoiding hard-to-reason-about SC

violations (e.g., double-checked locking errors), but also eliminates all violations of atomicity of dynamically bounded regions, e.g., common buggy patterns such as improperly synchronized read–modify–write accesses and accesses to multiple variables. The following code snippets will execute atomically under DBRS:

- `x += 42` (read–modify–write)
- `if (o != null) { ... = o.f; }` (check before use)
- `buffer[pos++] = value` (multi-variable operation)

Additionally, DBRS restricts possible program behaviors, making the job of various static and dynamic analyses or runtime systems simpler by assuming DBRS. Current analyses and runtimes either (unsoundly) assume there are no data races in the program,[1] or they consider the effects but incur increased overhead. The model checker CHESS considers many possible interleavings because of the effects of data races [MQ07, BM08]. Static dataflow analysis must account for racy executions to be sound [CVJL08]. Software-based multithreaded record & replay needs to deal with the possibility of data races [VLW+11]: RecPlay assumes data race freedom unsoundly to reduce costs [RDB99]; Chimera shows that handling data races could lead to prohibitively high overhead [LCFN12]; other approaches sidestep this problem but incur other disadvantages or limitations such as not supporting both online and offline replay [PZX+09, LWV+10] or requiring extra processor cores [VLW+11].

[1] In theory, a static or dynamic analysis for C++ can provide any behavior for racy executions and still be sound, since C++ (unlike Java) provides no semantics for racy executions [BA08]. Real-world programs have data races, so it is practical for analyses to behave reasonably in their presence.

Optimizations such as method inlining and loop unrolling, increase the size of dynamically bounded regions beyond the source-code level, eliminating more bugs. Since reorderings are allowed within the regions, it expands the scope of possible compiler reordering optimizations. Since these optimizations make regions strictly larger than they appear to be at the source-code level, the atomicity of source-code-level regions still holds.

DBRS is clearly not as strong as full-SFR RS, which will potentially eliminate more errors and allow fewer interleavings for static and dynamic analyses to consider. It might be more natural to reason about full-SFR RS, which requires considering only synchronization operations, than DBRS, which requires reasoning about method and loop boundaries as well. On the other hand, full-SFR RS requires interprocedural reasoning: a region is synchronization free only if (transitively) all of its callee methods are synchronization free.

We suspect that DBRS and full-SFR RS are advantageous *not* for programmability, but rather for enforcing behaviors that many programmers *already assume*.

Progress. A program can be *unable* to make progress under DBRS even though it *might* make progress under SC. For example, the following program *might* terminate—but is *not guaranteed* to terminate—under SC:

Initially x = false, y = false, done = false

```
// T1:
while (!done) {
  x = !x;
  y = !y;
}
```

```
// T2:
while (true) {
  if (x != y) break;
}
done = true;
```

In contrast, under DBRS, each loop body executes atomically, so the program *cannot* terminate.

Interestingly, *full-SFR RS* can impede the progress of a program for which SC guarantees progress, although full-SFR RS guarantees progress for any program for which *DRF0*

guarantees progress. For example, the following program always terminates under SC; cannot terminate under full-SFR RS (any terminating execution would violate full-SFR RS); and may or may not terminate under DRF0 (e.g., the compiler can legally hoist each load out of its loop):

Initially x = false, y = false

```
// T1:                  // T2:
y = true;               x = true;
while (!x);             while (!y);
```

Under DBRS, this program always terminates because regions are dynamically bounded.

Proof of progress under DBRS. If SC guarantees progress[2] for a program, DBRS guarantees progress for that program. Suppose SC guarantees progress for all executions of a program but there exists an execution e of the program under DBRS that does not guarantee progress. By the definition of DBRS, e is some serial interleaving of dynamically bounded regions. By the definition of serializability, each region appears to execute in isolation, and hence executes in accordance with program order. Moreover, each thread's regions appear in program order. Hence, e is some interleaving of operations that respects program order. Moreover, because each dynamically bounded region executes a bounded number of instructions, no thread in e can starve. Thus, e is equivalent to an SC execution with a fair thread scheduler (in particular, executing under SC with a thread scheduler that only context switches at region boundaries). Hence, there exists an SC execution that does not terminate which contradicts our assumption and hence proved by contradiction.

[2]We assume a fair scheduler that does not allow any thread to starve.

3.3 Avoiding Erroneous Behavior

This section describes a limited study of one of DBRS's potential benefits: its ability to automatically avoid errors caused by data races. To help expose such errors, we have implemented *adversarial memory* (AM), a dynamic analysis that helps expose behaviors that are allowed under the Java memory model [FF10, MPA05]. (Similar behaviors are possible under other language memory models including C++'s [BA08].) Under AM, each load from memory can choose from a buffer of stored values; the load may choose any value that does not violate established happens-before relationships. AM instruments potentially racy memory locations, identified by a sound dynamic race detector based on the FastTrack algorithm [FF09, BCM10].

We first identify potential errors by executing programs with AM. We use the DaCapo [BGH+06] and SPECjbb benchmarks[3], as well as the smaller Java Grande benchmarks [SBO01] evaluated by the AM paper [FF10]. In **Chapter 4**, we describe a technique call *EnfoRSer*, to practically enforce DBRS on commodity hardware. EnfoRSer is an efficient approach that leverages compiler transformations and two-phase locking. This section discusses results that highlight the potential of DBRS as a memory model using EnfoRSer as the technique. Note that this evaluation, to study the strength of DBRS as a memory model, could be implemented using other programming models like *transactional memory* (TM) [HM93] (although TM is inefficient in the context of DBRS: Section 4.5) or even a simple, inefficient global lock that enforces atomicity of dynamically bounded regions.

Since the EnfoRSer implementation enforces DBRS only in Jikes RVM's [AAB+05] (refer Section 4.3) optimizing compiler, our experiments forcibly compile all methods with

[3] `http://www.spec.org/jbb200{0,5}`, `http://users.cecs.anu.edu.au/~steveb/research/research-infrastructure/pjbb2005`

	Erroneous behavior		
	JMM (AM alone)	**SC** (perturbation & inspection)	**DBRS** (AM + EnfoRSer)
hsqldb6	Infinite loop	None	None
sunflow9	Null ptr exception	None	None
jbb2000	Corrupt output	Corrupt output	None
jbb2000	Infinite loop	None	None
sor	Infinite loop	None	None
lufact	Infinite loop	None	None
moldyn	Infinite loop	None	None
raytracer	Fails validation	Fails validation	None

Table 3.1: Errors exposed by adversarial memory (AM) that are possible under the Java memory model, and whether the errors are possible under SC and DBRS.

the optimizing compiler. Table 3.1 reports a row for each error exposed by AM. AM exposes two different errors in SPECjbb2000. Each result is repeatable across many trials. The *JMM* column shows erroneous behaviors allowed under the Java memory model, exposed using AM.

To evaluate DBRS's ability to avoid these errors, we use AM and EnfoRSer in the same execution. The compiler performs EnfoRSer's compiler transformations, followed by AM's instrumentation of racy loads and stores. We integrate EnfoRSer and AM by making AM aware of the happens-before edges established by EnfoRSer's locks (e.g., coordination between threads) (refer **Chapter 4** for the EnfoRSer mechanisms). The *DBRS* column shows its potential to successfully avoid all erroneous behavior, by providing atomicity of dynamically bounded regions, even in the presence of data races.

We have also determined whether these erroneous behaviors are possible under sequential consistency (SC). Despite the fact that these errors do not occur in typical runs (i.e., runs without AM), two of the errors are still possible under SC (the *SC* column):

SPECjbb2000 uses unsynchronized updates to a shared (64-bit) long field to keep track of elapsed time in milliseconds. Java makes no guarantees for the atomicity of accesses to the low and high 32 bits of a long [LY99, FF10]. We confirmed an atomicity violation by inspecting the code. We were not able to reproduce a violation since it would take a very long time to overflow the counter's low bits (AM in fact exposes a *visibility* error by reading a stale value). DBRS avoids the errors since it inherently makes accesses to long fields atomic.

In raytracer, multiple threads perform additions to a shared int field called checksum1:

```
checksum1 = checksum1 + val; // val is thread local
```

Under SC, the program still computes an incorrect checksum if two threads' load–add–store operations interleave. DBRS eliminates the error by making the load–add–store atomic. SC eliminates the other errors exposed by AM. We find that five out of six of these errors are *visibility* errors: a thread sees a "stale" value not possible under an SC execution. For each of these errors, a loop cannot terminate because it repeatedly sees a shared variable's stale value instead of an up-to-date value. The sixth error is in sunflow9, which throws an exception when it reads and then dereferences a stale value of null. We have been unable to expose this error under SC (or DBRS), suggesting that it is an SC violation.

In general, many real-world atomicity violations are possible under SC (e.g., [LPSZ08, ZdKL+13, LTQZ06]). If the region that requires atomicity is small enough, DBRS eliminates the error.

3.4 Summary

We have proposed a memory model that will provide serializability of dynamically bounded regions of code. Dynamically bounding regions should bound the cost of conflict detection and misspeculation. Several challenges exist in enforcing DBRS. The approach should use a lightweight mode of conflict detection and resolution. The approach might use a speculative execution technique that exploits compiler transformations. Compiler transformations, if used to provide atomicity of regions should tackle challenges like aliasing in static analysis, consider control-flow in the transformations and generate efficient code. The details of our algorithms that resolve these challenges and enforce this memory model are provided in **Chapter 4**. We empirically evaluate the utility of this memory model to demonstrate that DBRS not only provides SC but eliminates several other subtle errors due to violations of atomicity. We provide alternate, and more efficient form of software-based DBRS enforcement in **Chapter 5**. We extend our ideas to overcome any drawbacks of these software-based approaches by leveraging commodity hardware transactional memory to enforce this memory model in **Chapter 6**.

3.5 Impact and Meaning

DBRS as a memory model shows excellent coverage in eliminating harmful SC violations and atomicity violations from ill-synchronized code. Section 3.3, clearly demonstrates that the memory model is strictly stronger than SC. If compiler and runtime techniques could efficiently enforce atomicity of dynamically bounded regions, then potentially, the goal of enforcing data-race-freedom could be realized on commodity systems. The hallmark of DBRS is that it allows compiler and hardware optimizations within the

peripheries of a region. We expect, that the relaxed restrictions on optimization, could amortize the run-time overhead of the memory model.

Chapter 4: Hybrid Static–Dynamic Analysis for Statically Bounded Region Serializability

We have introduced our proposed memory model, DBRS, for providing strong semantics in **Chapter 3**. Serializability of dynamically bounded regions is an appealing proposition since we expect it to restrict several buggy interleavings in racy programs while still allowing optimizations within a region. The bounded nature of the regions should enable enforcement of DBRS at reasonable overheads. Our overall objective, to develop techniques that can be effective on commodity hardware, to enforce strong semantics can only be realized through software-based analyses that incur reasonable overheads. Next, we propose lightweight mechanisms to enforce atomicity of dynamically bounded regions. In this chapter, we explore a novel hybrid static–dynamic analysis called EnfoRSer to demonstrate that DBRS can be enforced practically on commodity systems. In Section 3.3, we have shown the bug elimination potential of DBRS by implementing a separate analysis to expose bugs and using EnfoRSer to eliminate the exposed bugs; here we will present the EnfoRSer analysis in detail.

EnfoRSer's static and dynamic components rely on regions being both *statically and dynamically* bounded; *statically bounded* means that when the region executes, *each static instruction executes at most once*. EnfoRSer thus enforces the same memory model as DBRS but calls it *statically bounded region serializability* (SBRS), which bounds regions

at loop back edges and method calls and returns, as well as synchronization operations. For the remainder of this chapter and **Chapter 5** we use the term SBRS for essentially the same memory model as DBRS since the compiler transformations require the regions to be statically bounded for simplicity and correctness.

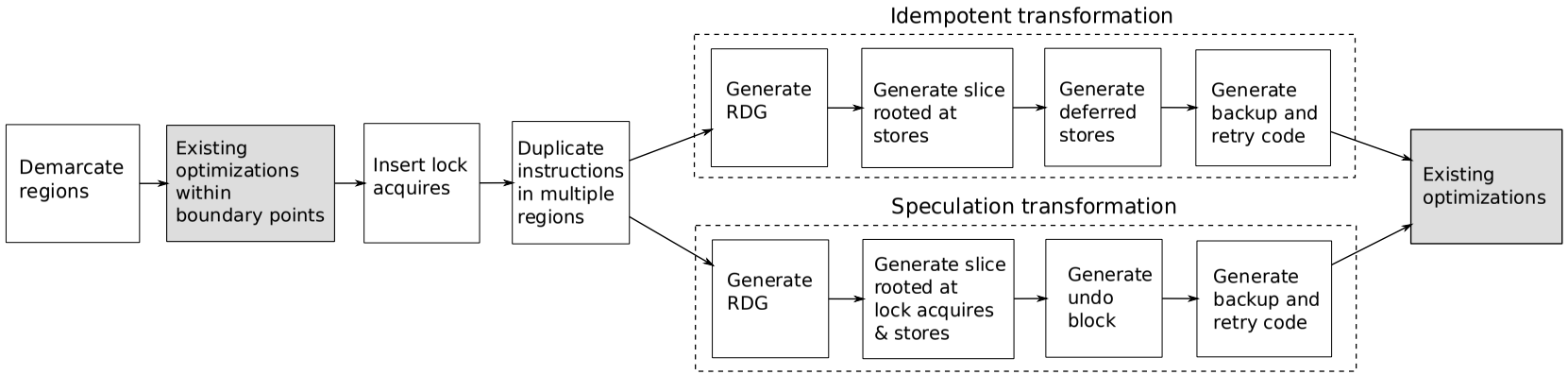


Figure 4.1: The transformations performed by an EnfoRSer-enabled optimizing compiler. Boxes shaded gray show existing compiler optimizations. Dashed boxes indicate transformations performed once on each region.

4.1 Design of EnfoRSer

This section describes EnfoRSer, a hybrid static–dynamic analysis that enforces end-to-end SBRS.

4.1.1 Overview

To enforce SBRS, EnfoRSer's compiler pass partitions the program into regions bounded by synchronization operations, method calls, and loop back edges. Dynamic analysis then enforces atomicity of regions. A naïve approach for enforcing atomicity is as follows. First, associate a "lock" with each potentially shared object[4] (e.g., by adding a header word to each object that tracks the object's ownership state). Henceforth, this chapter uses the term "lock" to refer to per-objects locks that are added by EnfoRSer and acquired by its instrumentation and are *not* available to programmers. Second, instrument the region to ensure that (1) a thread owns lock on an object before it is accessed in a the region and (2) the region releases locks only when it ends. This approach implements two-phase locking, so the execution will be equivalent to a serialized execution of regions.

This naïve approach has two limitations. First, because threads can access objects in any order, and per-access locks must be held until the end of the region to satisfy two-phase locking, a thread can deadlock while executing a region that attempts to acquire multiple locks. EnfoRSer solves this problem by applying a compile-time *atomicity transformation*: if a thread cannot acquire a lock because another thread holds it—we call this situation a "conflict," as all region conflicts result in a failed lock acquire—the transformed region is restarted, avoiding the deadlock. Second, acquiring a lock on each object access seems

[4]This chapter uses the term "object" to refer to any unit of shared memory.

to require an atomic operation such as compare-and-swap (CAS), which is expensive. EnfoRSer uses special lightweight locks that mostly avoid atomic operations [BKC+13] (Section 4.1.3).

The key challenge is ensuring that a transformed region can be safely restarted without losing atomicity. If a region restarts after writing to shared objects, then either those effects could be visible to other threads, or the region may behave incorrectly after restart. Either way, region's atomicity is violated. We design, implement, and evaluate two atomicity transformations that address this challenge. (We investigate two different transformations in order to explore the design space.)

The *idempotent* transformation modifies each region to defer stores until all per-object locks have been acquired. If a region encounters a conflict while acquiring a lock, it can safely restart since it executes idempotently up to that point. The challenge is generating correct code for deferring stores in the face of object aliasing and conditional statements.

The *speculation* transformation modifies the region to perform stores speculatively as it executes. The transformed region backs up the original value of stored-to memory locations. On a conflict, the modified region can restart safely by using the backed-up values to restore memory to its original state. The challenges are maintaining the backup values efficiently and restoring them correctly on retry. In summary, EnfoRSer's compiler pipeline operates as shown in Figure 4.1. First, it demarcates a program into regions (Section 4.1.2), and any optimizing passes in the compiler are performed, modified to be restricted within region boundaries. Next, the compiler inserts lock acquire operations before object accesses (Section 4.1.3). The compiler then performs a duplication transformation (Section 4.1.4) and builds an intermediate representation (Section 4.1.5), before transforming each region using one of the two atomicity transformations (Sections 4.1.6 and 4.1.7).

Finally, any remaining compiler optimizations are performed.

At run time, if a thread executing a region detects a potential conflict, it restarts the region safely, preserving atomicity. The atomicity of regions guarantees equivalence to some serialized execution of regions.

4.1.2 Demarcating Regions

The first step in EnfoRSer's compiler pass is to divide the program into regions; later steps ensure that these regions execute atomically. The following program instructions are region *boundary points*: synchronization operations, method calls, and loop back edges.[5] Since method calls are boundary points, regions are demarcated at method entry and return, i.e., EnfoRSer's analysis is intraprocedural. An EnfoRSer region is a maximal set of instructions such that (1) for each instruction i, all (transitive) successors of i are in the region provided that the successors are reachable without traversing a region boundary point; and (2) there exists a "start" instruction s such that each instruction i in the region is reachable from s by only traversing edges in the region. Note that an instruction can be in multiple regions statically, i.e., reachable from multiple boundary points without an intervening boundary point—a situation that Section 4.1.4 addresses.

Reordering within and across regions. If the compiler reorders instructions across region boundary points, region atomicity will be violated. Similar to DRFx's [MSM+10] soft fences, EnfoRSer prohibits inter-region optimizations, by modifying optimization passes, such as common subexpression elimination, that may reorder memory accesses to ensure

[5] A loop *back edge* is any control-flow edge to a loop header from a basic block dominated by the loop header. Loop *headers* are not boundary points.

reordering is restricted within boundary points. We find that prohibiting reordering across region boundaries impacts performance negligibly for our implementation.

4.1.3 Lightweight Reader–Writer Locks

The next step after demarcating regions is instrumenting program loads and stores with lock acquire operations. Acquiring traditional locks on every object access would incur prohibitive overheads. EnfoRSer instead uses lightweight reader–writer locks that differ from traditional locks in two key ways. (1) A lock is always "acquired" in some state such as write-exclusive or read-shared, and acquiring the lock does not require an atomic operation if the thread already holds the lock in a compatible state. (2) A thread never automatically "releases" a lock; instead, another thread may acquire the lock, but it must first *coordinate* with the thread(s) that currently hold the lock. The design and implementation of these reader–writer locks (called "locks" for the remainder of the chapter) are based closely on *Octet* locks [BKC+13]. We summarize their operation here.

Each object has a lock, denoted by o.lockState but not visible to programmers, that is always acquired in one of the following states:

- $\mathsf{WrEx_T}$: Thread T may read or write the object.
- $\mathsf{RdEx_T}$: T may read but not write the object.
- RdSh: Any thread may read but not write the object.

At each program store and load to the object referenced by o, the compiler inserts instrumentation denoted by acq_wr(o) and acq_rd(o), respectively, as the following pseudocode shows:

```
acq_wr(o); // instrumentation
o.f = ...; // program store

acq_rd(p); // instrumentation
... = p.g; // program load
```

The following pseudocode shows the definitions of acq_wr() and acq_rd() (T is the executing thread):

```
acq_wr(Object obj) {
  if (obj.lockState != WrEx_T)
    wrSlowPath(obj); /* change obj.lockState */
}

acq_rd(Object obj) {
  if (obj.lockState != WrEx_T &&
      obj.lockState != RdEx_T) {
    if (obj.lockState != RdSh)
      rdSlowPath(obj); /* change obj.lockState */
    load_fence; // see footnote 6
  }
}
```

To acquire a lock to obtain read or write access to its associated object, a thread T checks the state of the lock. If the lock's state is compatible with the access being performed (e.g., T wants to read or write an object in WrEx_T state), the lock is already "acquired" without any synchronization operations. This check is called the *fast path*.

Otherwise, a thread encounters a lock in an incompatible state, and it must change the lock's state. Table 4.1 shows all possible state transitions.[6] The first three rows (*Same state*) correspond to the fast path. The remaining rows show cases that must change a lock's state. Collectively, all operations that change a lock's state are called the *slow path*. An *upgrading* transition (e.g., from RdEx_T to RdSh) expands the set of accesses allowed by the lock; it requires an atomic operation to change the lock's state.

The slow path triggers a *conflicting* transition in cases where the program access conflicts with accesses allowed under the lock's current state. Because other thread(s) might

[6]EnfoRSer does not need nor use Octet's RdSh counter [BKC+13]. Instead, EnfoRSer ensures that reads to RdSh locks happen after the prior write by issuing a load fence on the RdSh fast path.

Code path(s)	Transition type	Old state	Program access	New state	Sync. needed
Fast	Same state	$\mathsf{WrEx_T}$	R/W by T	Same	None
		$\mathsf{RdEx_T}$	R by T	Same	
		RdSh	R by T	Same	
Fast & slow	Upgrading	$\mathsf{RdEx_T}$	W by T	$\mathsf{WrEx_T}$	Atomic op.
		$\mathsf{RdEx_{T1}}$	R by T2	RdSh	
	Conflicting	$\mathsf{WrEx_{T1}}$	W by T2	$\mathsf{WrEx_{T2}}$	Roundtrip coord.
		$\mathsf{WrEx_{T1}}$	R by T2	$\mathsf{RdEx_{T2}}$	
		$\mathsf{RdEx_{T1}}$	W by T2	$\mathsf{WrEx_{T2}}$	
		RdSh	W by T	$\mathsf{WrEx_T}$	

Table 4.1: State transitions for lightweight locks.

be accessing the object without synchronization, merely changing the lock's state—even with an atomic operation—is insufficient. Instead, the thread executing the conflicting transition, called the *requesting thread*, must *coordinate* with every other thread, called a *responding thread*, that has access to the object, to ensure the responding thread(s) "see" the state change. For $\mathsf{WrEx_T}$ and $\mathsf{RdEx_T}$ states, T is the responding thread; for RdSh states, every other thread is a responding thread.

A responding thread participates in coordination only at a *safe point*: a point that does not interrupt instrumentation–access atomicity (i.e., atomicity of lock acquire instrumentation together with the program access it guards). Conveniently, managed language virtual machines already place safe points throughout the code, e.g., at every method entry and loop back edge. Furthermore, all blocking operations (e.g., *program* lock acquires and I/O) must act as safe points. If a responding thread is actively executing program code, the requesting and responding threads coordinate *explicitly*: the requesting thread sends a request, and the responding thread responds when it reaches a safe point. Otherwise, the

responding thread is at a blocking safe point, so the threads coordinate *implicitly*: the requesting thread atomically modifies a flag that the responding thread will later see. Finally, in either case, the requesting thread finishes changing the lock's state and proceeds with the access.

In comparison to traditional locks that add synchronization at every access, the locks used by EnfoRSer add no synchronization at non-conflicting accesses (the fast path), but they require coordination with other threads at conflicting accesses (the slow path). This tradeoff works well in practice for many programs, which aim for good thread locality by design.

Avoiding deadlock. As described so far, locks are deadlock prone: two threads each waiting to acquire a lock held by the other thread will wait indefinitely. To prevent deadlock, if a thread T is waiting for other thread(s) to relinquish a lock, T *allows other threads to acquire locks held by* T.

This behavior has interesting implications for EnfoRSer. When a thread tries to acquire an object's lock, it might give up *other objects' locks acquired in the same region*. As a result, rather than suffering from deadlock, as in traditional two-phase locking, regions may lose atomicity when they conflict. Hence, EnfoRSer's transformed regions actually restart *whenever a lock may have been released*—indicated by responding to another thread's coordination request.

Example. Figure 4.2 shows a region instrumented with lock acquire operations (acq_rd and acq_wr). The rest of this section uses this region as a running example.

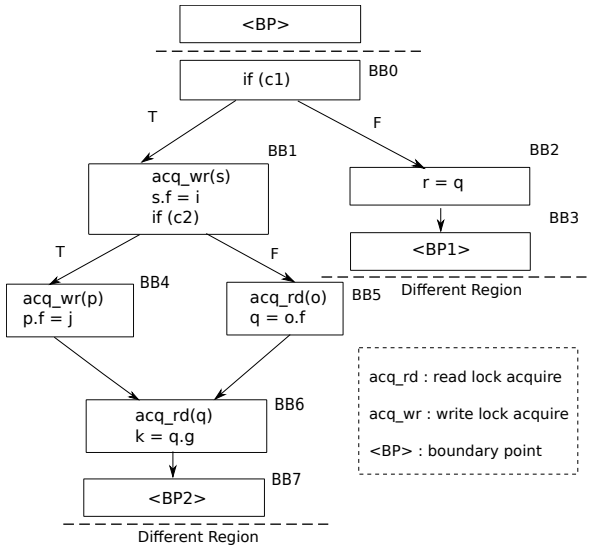


Figure 4.2: A region instrumented with lock acquire operations.

4.1.4 Duplication Transformation

If a lock acquire (acq_rd(o) or acq_wr(o)) experiences a conflict, it must restart the region. For the compiler to generate control flow for retry, the lock acquire must be statically in a single region.

The modified compiler thus performs a *duplication* transformation prior to atomicity transformations (Figure 4.1). This transformation replicates instructions and control flow so that every instruction (not including boundary points) is in its own region. First, a simple dataflow analysis determines which boundary point(s) reach each instruction (i.e., which region(s) each instruction resides in). Second, in topological order starting at method entry, the duplication algorithm replicates each instruction that appears in $k > 1$ regions $k - 1$ times, and retargets each of the original instruction's predecessors to the appropriate replicated instruction.

The duplication transformation does not duplicate any region boundary points. Since the number of region boundary points is constant for an instance of compiled code, duplication cannot result in an exponential blowup in code size.

4.1.5 The Region Dependence Graph

Region transformations require an intermediate representation to capture the relevant instructions that need to be modified. The atomicity transformations use static *slices* [HRB88]: the set of instructions that are data and control dependent on some instruction. Thus, for each region, an atomicity transformation first builds a *region dependence graph* (RDG), based on Ferrante et al.'s program dependence graph [FOW87], which captures both data and control dependences in a region. The RDG needs to track true (write–read) dependences, but not output (write–write) or anti (read–write) dependences. It can ignore loop-carried dependences since regions are acyclic.

EnfoRSer's RDG construction algorithm uses the standard reaching-definitions analysis to compute intraprocedural data dependences in a region. The algorithm treats lock acquire operations `acq_rd(o)` and `acq_wr(o)` like a use of the object referenced by `o`. Because RDG construction is performed at the region level, aliasing relationships that arise due to operations outside the region cannot be inferred. Hence, the RDG construction algorithm adopts a conservative, type-based approach, e.g., it assumes that a load from field `f` of object `o` can have a dependence with any earlier store to the same field `f` of object `p`, since `p` and `o` could potentially alias. It also conservatively assumes that loads from arrays may have dependences from stores to type-compatible arrays. The algorithm relies on the data dependences computed by the reaching-definitions analysis, as well as control dependences computed by a standard post-dominance analysis, to construct the final RDG.

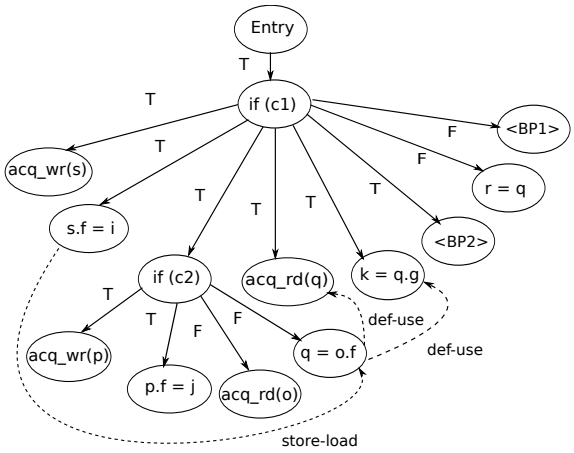


Figure 4.3: A region dependence graph for the region in Figure 4.2.

Figure 4.3 shows the RDG for the region from Figure 4.2. The dotted lines depict store–load dependences (between accesses to object fields or array locations) and def–use dependences (between definitions and uses of local variables). The construction algorithm's aliasing assumptions lead to a store–load dependence between `s.f = i` and `q = o.f`, since `s` and `o` may be aliased. The `Entry` node is the root of the RDG and has control dependence edges marked as `T` to all nodes (only one in this case) that are not control dependent on any other nodes. Other edges labeled `T` and `F` are control dependence edges from conditionals to dependent statements.

Slicing of the RDG. Each atomicity transformation must identify which parts of the region are relevant for the transformation. The transformation performs static program slicing of the RDG (based on static program slicing of the program dependence graph [HRB88]) to identify relevant instructions. A *backward* slice *rooted* at an instruction i includes all instructions on which i is *transitively* dependent. Dependences include both data (store–load

and def–use) and control dependences. A slice rooted at a set of instructions is simply the union of the slices of each instruction.

4.1.6 Enforcing Atomicity with Idempotence

The *idempotent* transformation defers program stores until the end of the region. The store-free part of the region which includes all lock acquires is side-effect-free, and executes *idempotently*, so it can be restarted safely if a lock acquire detects a potential region conflict.

To defer program stores, the transformation replaces each static store with an assignment of a new local variable. Figure 4.4 shows the region from Figure 4.2 after the idempotent transformation. The stores `s.f = i` and `p.f = j` from Figure 4.2 are replaced with assignments to fresh locals `i'` and `j'` in `BB1` and `BB3` of Figure 4.4. The two main challenges are (1) generating correct code for loads that might alias with stores so that they read the correct value and (2) generating code at the end of the region that performs the stores that should actually execute.

Loads aliased with deferred stores. Any load that aliases a deferred store needs to read from the local variable that backs up the stored value, rather than from the deferred store's memory location. The RDG in Figure 4.3 includes a store–load edge from `s.f = i` to `q = o.f` because RDG construction conservatively assumes all type-compatible accesses might alias. As shown in `BB1` of Figure 4.4, the transformation substitutes a definition of a new temporary variable `i' = i` in place of the store `s.f = i`. If `s` and `o` alias, then `q = i'` should execute in place of `q = o.f`; otherwise `q = o.f` should execute normally.

For each load, the code generation emits a series of alias checks to disambiguate the scenarios: one for each potentially aliased store that may reach the load. BB4–BB6 in Figure 4.4 show the result of this transformation. The transformation first emits the original load (BB4). Then, for each store on which the load appears to be dependent, the transformation generates an assignment from the corresponding local variable, guarded by an alias check (BB6, guarded by the check in BB5). By performing these checks in program order, the load will ultimately produce the correct value, even if multiple aliasing tests pass. If a possibly-aliased store executes conditionally, the transformed load is guarded by a conditional check in addition to the alias check to ensure that the original conditionals guarding the store are respected. A similar situation arises with array accesses, in which case the aliasing checks must not only compare the array references, but also the index expressions.

Although this approach may generate substantial code per load, in practice later compiler passes can simplify it significantly. For example, if the compiler can prove two object references definitely do or do not alias, constant propagation and dead code elimination will remove unnecessary checks.

Deferred stores. It is nontrivial to generate code at the end of a region to perform the region's stores, because the code should execute only the stores that actually would have executed. The transformation generates code at each region exit based on a slice of the region's stores from the RDG. It generates each deferred store using the mapping from static stores to local variables. To ensure a deferred store only executes if it would have in the original region, the slice includes all condition variables on which stores are control dependent. The generated code guards the deferred stores with these conditionals. For example, BB8 in Figure 4.4 checks condition c2, executing the deferred store in BB9 only

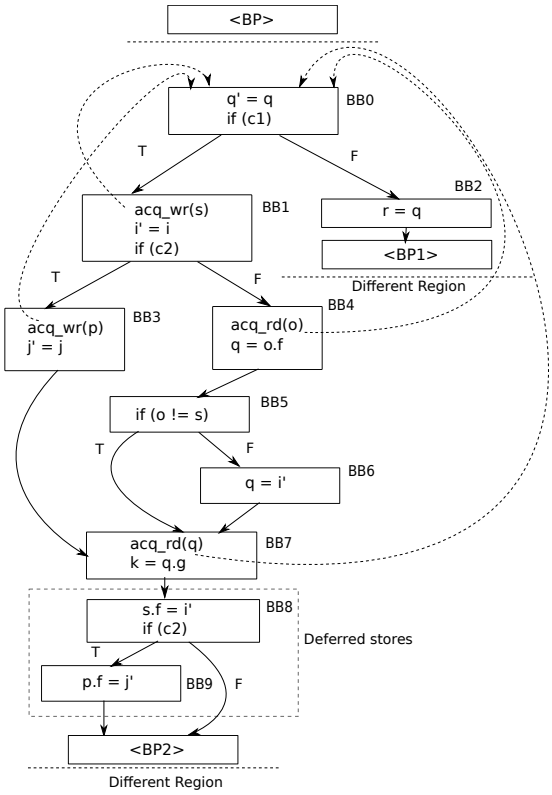


Figure 4.4: Region from Figure 4.2 after idempotent transformation.

if the store would have occurred in the original region (note that the same condition guards j' = j in BB3).

Note that it is possible that the conditional variables guarding stores are overwritten during the execution of the idempotent portion of the region; if so, those conditionals may resolve differently while executing deferred stores. To avoid this problem, the transformation "backs up" the results of any conditional in the idempotent region in fresh local variables, and uses these locals in the conditionals guarding deferred stores. Similarly, any store to an object field or array in the original region is dependent on the base reference of

the object or array. If these base references may change during execution, the transformation backs them up at the point of the original store and uses the backups when executing the deferred stores.

Handling side-effect on local variables. Although the idempotent transformation defers stores, the region can still have side effects if it modifies *local* variables that may be used in the same or a later region. For each region, the idempotent (and speculation) transformations identify such local variables and insert code to (1) back up their values in new local variables at region start and (2) restore their values on region retry. In Figure 4.4, local variable q is backed up in q' at the beginning of the region (BB0). If a lock acquire detects a potential conflict, the code restores the value of q from q' (not shown) and control returns to the region start (a dashed arrow indicates the edge is taken if the operation detects a potential conflict).

4.1.7 Enforcing Atomicity with Speculation

EnfoRSer's second atomicity transformation supports *speculative execution*. This transformation does not alter the order of operations in the region, but it transforms the region to perform stores speculatively: before a store executes, instrumentation backs up the old value of the field or array element being updated. If a lock acquire detects a possible conflict, the region restarts safely by "rolling back" the region's stores. Figure 4.5 shows the region from Figure 4.2 after applying the speculation transformation. The primary challenge is generating code that correctly backs up stored-to variables and (when a conflict is detected) rolls back speculative state.

Supporting speculative stores. Since regions are acyclic and intraprocedural, every static store executes at most once per region execution. Hence, the speculation approach's transformation can associate each store with a unique, new local variable. Before each store to a memory location (i.e., field or array element), the transformation inserts a load from the location into the store's associated local variable.

Generating code to roll back executed stores is nontrivial due to the challenge that *some* stores may not execute. The solution depends on the subpath executed through the region. To tackle this challenge, the transformation generates an *undo block* for each region. In reverse topological order, it generates an undo operation for each store s in the region that "undoes" the effects of s using the associated backup variable. To ensure that s is only undone if it executed in the original region, this undo operation executes conditionally. The transformation determines the appropriate conditions by traversing the RDG from s up through its control ancestors. Similar to that of the idempotent approach, conditional variables and object base references are backed up if necessary (if potentially modified later), so they can be used during rollback. The undo block in Figure 4.5 illustrates the result of this process. The store to p.f is topologically after the store to s.f, so it appears first in the undo block. The store to p.f is performed if both c1 and c2 are true, so the old value is restored under the same conditions. Likewise, the store to s.f is only performed, and hence only undone, if c1 is true.

Supporting retry. The transformation generates control flow for each lock acquire to jump to the appropriate location in the undo block if the lock acquire detects a potential conflict. The jump target is the first undo operation associated with a store that is a control-flow predecessor of the lock acquire. Figure 4.5 shows these conditional jumps with dashed

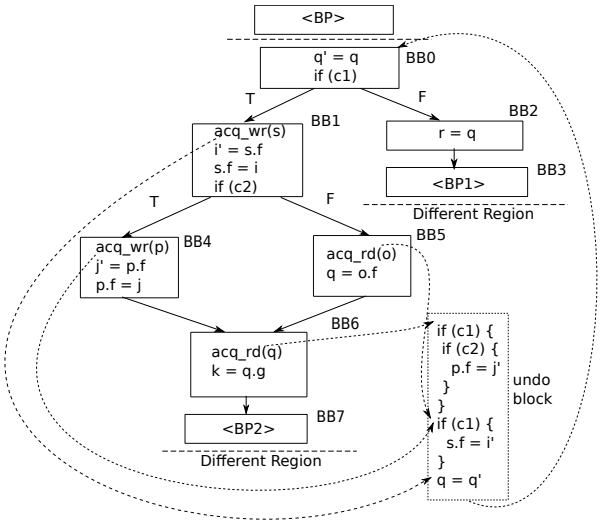


Figure 4.5: Region from Figure 4.2 after speculation transformation.

lines. As with the idempotent transformation, local variables must be backed up and restored during retry. In the figure, q is backed up in q' at the beginning of the region, and restored at the end of the undo block.

4.2 Optimizations

EnfoRSer's compiler analysis identifies accesses that do not need lock acquires, which in turn enables reducing the size of regions that need to be analyzed and transformed.

Statically redundant lock acquires. A lock acquire for an object access is "redundant" if along all paths a sufficiently strong lock acquire is performed on the same object in the same region. An acq_rd() is redundant if definitely preceded by acq_rd() or acq_wr() on the same object; an acq_wr() is redundant if definitely preceded by acq_wr() on the same object. A lock is guaranteed to be in a compatible state at a redundant lock acquire; another thread cannot change the state without triggering restart of the current region. The

compiler uses an intraprocedural, flow-sensitive dataflow analysis (based on an analysis used by prior work [BKC+13]) to identify redundant lock acquires.

Static data race detection. EnfoRSer's transformations can optionally use the results of sound static analysis that identifies all accesses that might be involved in data races. Remaining accesses are definitely data race free (DRF). An access that is DRF does not need a corresponding lock acquire. Prior work has also leveraged sound static race detection to simplify dynamic analysis [CLL+02, EQT07, vPG03, LCFN12].

Whole-program static analysis is somewhat impractical in the real world since it relies on all code being available and all call graph edges being known at analysis time, which is difficult in the presence of dynamic class loading and reflection. An implementation could handle unexpected dynamic behavior by dynamically recompiling *all* code to treat all accesses as potentially racy, or by using incremental static analysis to identify and recompile new potentially racy accesses. Our experiments sidestep this challenge by making all code and calls known to the static analysis.

Optimizing region demarcation. We optimize region demarcation to take advantage of optimizations that remove lock acquires. Optimized region demarcation distinguishes between *programmatic regions*—regions delimited at boundary points—and *enforced regions*—the regions whose atomicity is explicitly enforced by EnfoRSer's atomicity transformations. Optimized region demarcation starts and ends an enforced region at the first and last lock acquire of a programmatic region, respectively. Because any memory accesses outside of these boundaries are guaranteed to be DRF or guarded by locks earlier in the region, providing atomicity for these enforced regions is sufficient to guarantee atomicity for programmatic regions.

An access that has no lock acquire but is inside of an enforced region must still be handled by the atomicity transformations: the idempotent transformation handles every possible store–load dependence in an enforced region, and the speculation transformation backs up every store in an enforced region. Although the endpoint of an enforced region may come before the end of a programmatic region, EnfoRSer does not treat the end of an enforced region as a safe point, so a thread will not release locks between the end of the enforced region and the end of the programmatic region.

4.3 Implementation

We have implemented EnfoRSer in Jikes RVM 3.1.3, a high-performance Java virtual machine [AAB+05] that provides performance competitive with commercial JVMs. Our implementation is publicly available.[7]

Although the implementation targets a managed language VM, it should be possible to implement EnfoRSer's design for a native language such as C or C++. The main challenge would be adapting Octet to a native language [BKC+13].

Modifying the compilers. Jikes RVM uses two just-in-time dynamic compilers that perform method-based compilation. The first time a method executes, the *baseline* compiler compiles it directly from Java bytecode to machine code. If a method becomes hot, the *optimizing* compiler recompiles it at successively higher levels of optimization. The optimizing compiler performs standard intraprocedural optimizations. It performs aggressive method inlining but does not otherwise perform interprocedural optimizations.

[7] `http://www.jikesrvm.org/Research+Archive`

We modify the optimizing compiler to demarcate regions and restrict reordering across regions throughout compilation; to insert lock acquires (we use the publicly available Octet implementation of lightweight locks [BKC+13]); and to perform EnfoRSer's atomicity transformations on the optimizing compiler's intermediate representation (IR).

The baseline compiler does not use an IR, making it hard to implement atomicity transformations. Instead, in baseline-compiled code only, our implementation *simulates* the cost of enforcing SBRS without actually enforcing SBRS. The baseline compiler inserts (1) lock acquires, (2) instrumentation that logs each store in a memory-based undo log, and (3) instrumentation that resets the undo log pointer at each boundary point. This approach does not soundly enforce SBRS since it does not perform rollback on region conflicts. Since conflicts are infrequent and (by design) a small fraction of time is spent executing baseline-compiled code, this approach should closely approximate the performance of a fully sound implementation.

The compiler performs EnfoRSer's transformations on application and library code. Since Jikes RVM is written in Java, in a few cases VM code gets inlined into application and library code. Our prototype implementation cannot correctly handle inlined VM code that performs certain low-level operations or that does not have safe points in loops. To execute correctly, we identify and exclude 25 methods across all benchmarks and 10 methods in the Java libraries from EnfoRSer's transformations, instead inserting only lock acquires into these methods.

Demarcating regions. EnfoRSer demarcates regions at synchronization operations (lock acquire, release, and wait; thread fork and join), method calls, and loop back edges. These are essentially the same program points that are GC-safe points in Jikes RVM and other

VMs. Since Jikes RVM makes each loop header a safe point, the implementation bounds regions at loop headers instead of back edges. To simplify the implementation, we currently bound regions along special control-flow edges for switch statements.

The optimizing compiler is able to identify all synchronization operations when it compiles application and library code, since synchronization in Java is part of the language. The implementation does not bound regions at volatile variable accesses, which does not affect progress guarantees.

Object allocations can trigger GC, so they are safe points and thus boundary points in our implementation. A production implementation could either define regions as being bounded at allocation (non-array allocations already perform a constructor call); defer GC past allocations; or speculatively hoist memory allocation above regions.

Retrying regions. A region must retry if another thread may have performed accesses that conflict with the region's accesses so far. This case can occur only if the region, while waiting for coordination, responds to coordination requests (Section 4.1.3). In the idempotent approach, lock acquires use this criterion to decide whether to retry a region. However, the speculation approach cannot easily use this criterion: a region must not lose access to objects until after it executes its undo block—at which point the region must re-execute—so the decision to retry the region must be made before the region responds to coordination requests and potentially loses needed access to an object. The speculation approach instead triggers retry whenever a lock acquire takes the slow path. As a result, the idempotent approach provides lower retry rates than speculation—but the more precise retry criterion has a negligible performance impact (Section 4.4.3).

	Dynamic	Distinct
eclipse6	58,000	1
lusearch6	66	2
All other programs	0	0

Table 4.2: Total dynamic and distinct run-time exceptions in optimized code, rounded to two significant digits.

Retrying regions could lead to *livelock*, where conflicting regions repeatedly cause each other to retry. EnfoRSer could avoid livelock with standard techniques such as exponential backoff. Livelock is not an issue in our experiments, presumably because regions are short and conflicts are infrequent.

Runtime exceptions. Runtime exceptions present a problem for the idempotent transformation. Transformed regions can throw runtime exceptions (e.g., a null pointer exception when performing a program load). Simply letting execution exit the region (and jump to the innermost catch block) is incorrect: in the original code, some stores could have executed before the exception, while in the transformed code, no stores have executed. Exceptions are not a problem for the speculation approach: a thrown exception exits the region, preserving program semantics and region atomicity.

The idempotent approach could handle runtime exceptions by performing just-in-time *deoptimization*, transferring control to a deoptimized version of the method, e.g., a baseline-compiled version. By continuing execution at the same region boundary point, the exception would be thrown correctly, preserving semantics and SBRS.

Table 4.2 captures the scope of this problem. It shows how many runtime exceptions (exceptions whose types inherit from RuntimeException) are thrown from EnfoRSer-instrumented application and library methods that are compiled by the optimizing compiler. (Non-runtime exceptions can only be thrown using Java's throw instruction, which is a region boundary.) Overall, the programs throw few (dynamic and distinct) exceptions, so deoptimization should have reasonable performance. These runtime exceptions could occur inside of the transformed regions and then possibly violate program semantics, but we have not encountered any problems in practice.

Guarantees on native code. Java can call into C code using *native* methods. To access the Java heap from native methods, *Java Native Interface* (JNI) is used. In theory, we could instrument the JVM to handle heap accesses through JNI. However, our current implementation does not support SBRS with respect to heap accesses from native methods since the benchmarks that we use do not make such accesses.

4.4 Evaluation

This section evaluates EnfoRSer's run-time characteristics and performance.

4.4.1 Methodology

Benchmarks. The experiments execute our modified Jikes RVM on the multithreaded DaCapo benchmarks [BGH+06] versions 2006-10-MR2 and 9.12-bach (2009) with the large workload size (excluding programs Jikes RVM cannot run), distinguished by suffixes 6 and 9; and fixed-workload versions of SPECjbb2000 and SPECjbb2005.[8]

[8] `http://www.spec.org/jbb200{0,5}`, `http://users.cecs.anu.edu.au/~steveb/research/research-infrastructure/pjbb2005`

Platform. We build a high-performance configuration of Jikes RVM that adaptively optimizes the application as it runs (FastAdaptive) and uses the default high-performance garbage collector and adjusts the heap size automatically.

Experiments run on an AMD Opteron 6272 system with eight 8-core processors running Linux 2.6.32. We limit experiments to only four processors (32 cores) due to an anomalous result with 64 cores where EnfoRSer actually outperforms the baseline for some programs. (We find that adding *any* kind of instrumentation can improve performance, due to anomalies we have been able to attribute to Linux thread scheduling decisions [BKC+13].)

We have also evaluated EnfoRSer on an Intel Xeon E5-4620 system with four 8-core processors running Linux 2.6.32, in order to test EnfoRSer's sensitivity to the system architecture. On this platform, EnfoRSer adds nearly the same overhead as on the AMD platform (within 1% relative to baseline execution).

Static race detection. EnfoRSer can use any sound static analysis to identify definitely data-race-free (DRF) accesses (Section 4.2). We use the publicly available implementation of Naik et al.'s 2006 race detection algorithm, *Chord* [NAW06]. By default, Chord is unsound (i.e., it misses races) because it uses a may-alias lockset analysis.[9] We thus *disable* lockset analysis, using only Chord's thread escape and thread fork–join analyses to identify DRF accesses.

The programs we evaluate use reflection and custom class loading, which present a challenge for static analysis. To handle reflection, we use a feature of Chord that executes the program to identify reflective call sites and targets. Chord does not run eclipse6 correctly in our environment, so EnfoRSer runs eclipse6 assuming all accesses are racy.

[9]We have confirmed that there is no available implementation of their 2007 algorithm, which uses conditional must-not-alias analysis to be sound [NA07].

jython9 has custom-loaded classes that present a challenge for Chord, so EnfoRSer assumes all accesses in custom-loaded classes are racy. We cross-checked Chord's results with a dynamic race detector's output; we found one class (in jbb2005) that Chord does not analyze at all (for unknown reasons), so EnfoRSer fully instruments this class.

Static analysis is a one-time cost, incurred only when the program changes. In any case, Chord's cost is low: analyzing any of the programs we evaluate takes at most 90 seconds.

4.4.2 Run-Time Characteristics

The *Threads* columns of Table 4.3 report the total number of threads created and the maximum number of live threads.

The table's remaining columns report characteristics of executed regions, averaged across 10 trials. The *Insts.* columns report total dynamic IA-32 instructions and dynamic instructions per executed region, measured without any EnfoRSer instrumentation. The programs each execute billions of instructions, divided into regions that execute 21–36 IA-32 instructions each on average. Across all programs, each region executes 27 instructions on average. EnfoRSer's regions are comparable in size to DRFx's statically bounded regions (about 10 instructions per region [MSM+10]).

The last two columns show that the vast majority of regions do not detect a conflict. The idempotent approach provides a substantially lower retry rate than the speculation approach, since the idempotent transformation uses a more precise retry check (Section 4.3), but we have found that making the check more precise does not significantly improve the idempotent approach's performance. The program pjbb2005 has the highest retry rate; unsurprisingly it has the highest rate of conflicts [BKC+13]. The high cost of coordination to handle these conflicts leads to high overhead for both the idempotent and speculation

	Threads		Insts.	Insts. /	Dyn. regions retried	
	Total	Live	executed	region	Idem.	Spec.
eclipse6	18	12	6.1×10^{10}	26	<0.01%	<0.01%
hsqldb6	402	102	6.5×10^{9}	30	<0.01%	0.42%
lusearch6	65	65	1.7×10^{10}	28	<0.01%	<0.01%
xalan6	9	9	1.1×10^{11}	31	0.04%	0.87%
avrora9	27	27	4.2×10^{10}	38	<0.01%	0.73%
jython9	3	3	7.9×10^{10}	30	<0.01%	<0.01%
luindex9	2	2	2.6×10^{9}	25	<0.01%	<0.01%
lusearch9	c	c	1.7×10^{10}	28	<0.01%	<0.01%
pmd9	5	5	4.3×10^{9}	23	<0.01%	0.03%
sunflow9	$2\times c$	c	1.3×10^{11}	31	<0.01%	<0.01%
xalan9	c	c	7.7×10^{10}	27	0.05%	1.11%
pjbb2000	37	9	2.4×10^{10}	28	<0.01%	0.21%
pjbb2005	9	9	5.8×10^{10}	23	0.24%	3.32%

Table 4.3: Dynamic execution characteristics. A few programs launch threads proportional to the number of cores c, which is 32 in our experiments.

	Without static race detection					With static race detection				
	0	**1**	**2–3**	**4–7**	**≥8**	**0**	**1**	**2–3**	**4–7**	**≥8**
eclipse6	14%	20%	26%	15%	25%	14%	20%	26%	15%	25%
hsqldb6	7%	26%	37%	24%	6%	12%	24%	35%	23%	6%
lusearch6	21%	26%	7%	22%	24%	47%	13%	7%	10%	23%
xalan6	13%	30%	22%	13%	22%	37%	27%	18%	6%	12%
avrora9	6%	17%	7%	17%	53%	7%	18%	16%	19%	40%
jython9	25%	33%	11%	17%	14%	88%	10%	2%	0%	0%
luindex9	3%	29%	26%	26%	16%	44%	21%	21%	9%	5%
lusearch9	7%	26%	20%	20%	27%	19%	17%	15%	18%	31%
pmd9	10%	47%	16%	15%	12%	68%	12%	7%	9%	4%
sunflow9	20%	16%	22%	22%	20%	38%	29%	17%	11%	5%
xalan9	12%	39%	18%	9%	22%	40%	33%	9%	7%	11%
pjbb2000	22%	18%	24%	25%	11%	29%	24%	20%	20%	7%
pjbb2005	9%	19%	16%	45%	11%	13%	20%	14%	44%	9%

Table 4.4: Percentage of dynamic regions executed with various complexity (static accesses), without and with identifying statically DRF accesses to optimize region demarcation.

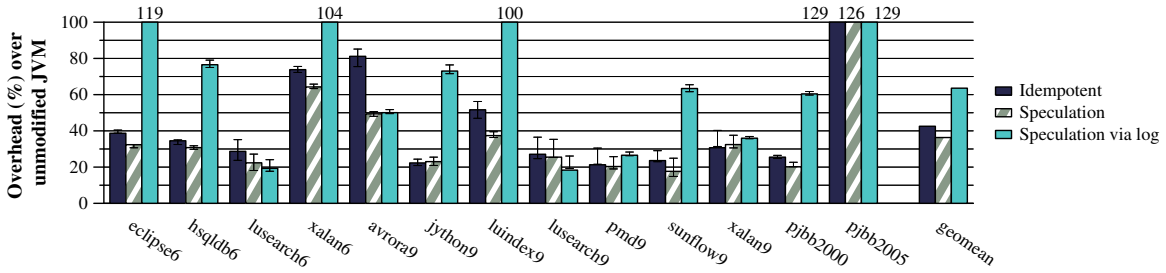


Figure 4.6: Run-time overhead over an unmodified JVM of providing SBRS with EnfoRSer's two atomicity transformations and speculation that uses a log to simulate a stripped-down version of STM.

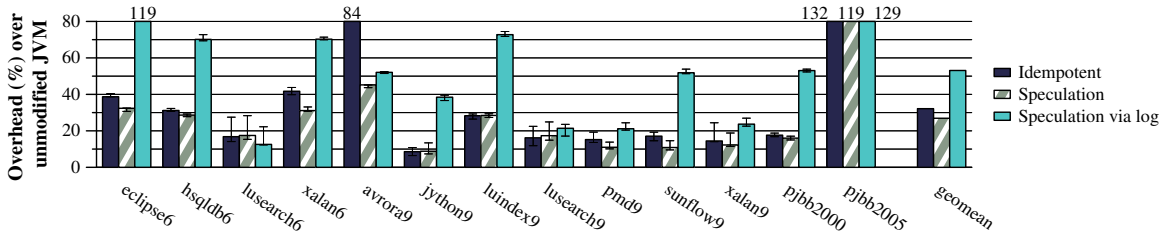


Figure 4.7: Run-time overhead over an unmodified JVM when transformations use whole-program static analysis to identify definitely data-race-free accesses. Otherwise, configurations are the same as Figure 4.6.

approaches (Section 4.4.3). Table 4.4 evaluates the complexity of executed regions. We measure a region's complexity using the *static* count of accesses in the region, after performing the shrinking optimization from Section 4.2. Unsurprisingly, static data race detection helps simplify regions, i.e., executed regions tend to be less complex overall. Even so, all programs still rely on EnfoRSer's transformations for a significant number of complex regions (i.e., regions with $\geq$2 static accesses).

4.4.3 Performance

Figure 4.6 shows the overhead EnfoRSer's transformations add over unmodified Jikes RVM, *without* making use of static race detection. Each bar is the median of 15 trials

(to minimize effects of machine noise); each bar shows 95% confidence intervals centered at the mean. The compiler duplicates instructions that are in multiple regions statically, adding 7% alone (not shown). The idempotent and speculation approaches add 43 and 36% total overhead, respectively.

The idempotent approach incurs costs to buffer and replay stores correctly and to check for aliasing at loads. Speculation is the better-performing approach since (in the common, non-conflicting case) it incurs only the cost of backing up each store's old value to a local variable. In a more aggressive compiler, the idempotent approach might have an advantage by enabling more intra-region reordering.

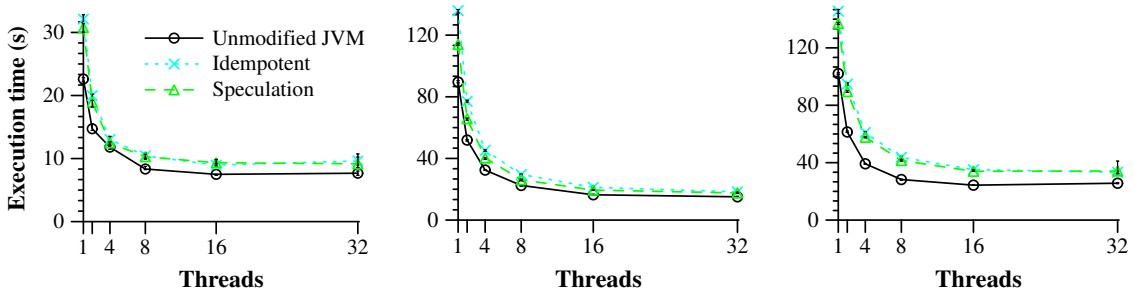
EnfoRSer adds the highest overhead for pjbb2005 (126–129%). This overhead primarily comes from the lightweight locks, which alone add 114% to pjbb2005 due to a relatively high conflicting access rate [BKC+13], which leads to expensive coordination among threads. EnfoRSer could achieve lower overhead for high-conflict executions by making use of "hybrid" lightweight locks that adaptively avoid coordination for high-conflict objects [CZB14].

Static race detection. Figure 4.7 shows the same configurations when the transformations make use of statically identified DRF accesses. Identifying DRF accesses eliminates about a quarter of the overhead on average: the idempotent and speculation transformations enforce SBRS at 32 and 27% overhead, respectively. Although using static race detection seems to have a small adverse effect on idempotent performance for pjbb2005, this effect is likely due to high run-to-run variation for this program (we have confirmed that the confidence intervals overlap).

Log-based speculation. The *Speculation via log* configuration in Figures 4.6 and 4.7 measures a stripped-down version of STM that uses memory-based undo logs instead of local variables for backing up stores, but still relies on EnfoRSer's lightweight locks and efficient conflict detection. On average it adds 64 and 53% total overhead, without and with sound static race detection, respectively, significantly more than EnfoRSer's speculation approach. The memory-based log incurs these high costs despite adding just one of several costs that STMs typically incur over EnfoRSer (Section 4.5).

Compilation time. The overheads in Figures 4.6 and 4.7 naturally include the costs of just-in-time compilation. EnfoRSer slows the optimizing compiler by performing additional analyses and transformations, and by bloating the internal representation (IR) and slowing downstream passes. Since EnfoRSer's analysis is intraprocedural, its complexity scales well with program size. We find that compile time increases over unmodified Jikes RVM by 2.5X and 2.1X without static race detection, and 1.9X and 1.7X with static race detection, for the idempotent and speculation approaches, respectively. However, the effect of EnfoRSer's compilation overhead on overall execution time is modest: a few percent, relative to baseline execution time.

Scalability. Three of our evaluated programs support executing with a variable number of application threads (Table 4.3 in Section 4.4.2). Figure 4.8 shows execution time for 1–32 application threads for three configurations: the unmodified JVM and EnfoRSer's idempotent and speculation approaches. We show idempotent and speculation configurations that do *not* use static race detection; scalability is similar for configurations that use static race detection.



(a) lusearch9 (b) sunflow9 (c) xalan9

Figure 4.8: Scalability of EnfoRSer's approaches, compared with the unmodified JVM, across 1–32 application threads.

The overhead added by EnfoRSer's approaches scales with additional application threads. This is unsurprising since the atomicity transformations should affect only single-thread performance. We expect EnfoRSer's scalability to be affected only by Octet's lightweight locks, which can incur different coordination costs with different numbers of threads. Our results, which concur with prior work [BKC+13], suggest that Octet overhead scales well with additional threads.

Contribution. Overall, these results show that EnfoRSer enforces SBRS at a reasonable cost of 27% average overhead (using the speculation approach with sound static race detection). To our knowledge, this represents the lowest reported overhead of any kind of end-to-end RS enforcement on commodity systems. The best-performing prior work (which, admittedly, provides full-SFR RS) requires additional available cores to avoid adding over 100% overhead [OCFN13]—a fundamentally different, replication-based approach that would *not* benefit from identifying statically DRF accesses.

4.5 Comparison with Transactional Memory

In this section we compare our approach to provide atomicity of statically bounded regions with *transactional memory* (TM) systems.

Software transactional memory. TM guarantees atomicity of programmer-annotated code regions [HM93, HF03]. EnfoRSer's approach is analogous to using *software* TM (STM) [HF03] to provide atomicity of every statically bounded region. In particular, EnfoRSer's idempotent transformation behaves similarly to a lazy-versioning STM, while its speculation transformation behaves similarly to an eager-versioning STM. However, EnfoRSer provides atomicity much more efficiently than STM for three reasons:

1. STMs maintain read/write sets (i.e., the last transaction(s) to read and write each object) to detect precisely whether an access conflicts with another thread's ongoing transaction. In contrast, EnfoRSer avoids maintaining read/write sets, at the cost of false region conflicts. The rollbacks these false conflicts trigger incur only a small penalty since statically bounded regions are short.

2. STMs that use eager or lazy versioning maintain *undo logs* or *redo logs*, respectively. At each store, the STM appends an entry to the memory-based log containing the address of the stored-to location and its old or new value. EnfoRSer can instead map each store to a dedicated local variable for deferring or backing up the store, because regions are statically bounded.

3. The locks used by EnfoRSer are a lighter-weight mechanism for conflict detection than STMs typically use, as long as relatively few accesses conflict. (Recent work introduces an STM that uses similar lightweight locks [ZHCB15].)

Hardware TM. Hammond et al. introduce a memory model and associated hardware where all code is in transactions [HWC+04]. However, manufacturers have been reluctant to incorporate general-purpose *hardware* TM (HTM) support into already-complex cache and memory subsystems.

Intel's recently released Haswell architecture provides *restricted transactional memory* (RTM) support with an upper bound on shared-memory accesses in a transaction [YHLR13]. It might seem at first that RTM would be well suited to enforcing SBRS. However, two studies find that the overhead of an RTM transaction is substantial: the startup and tear-down costs of each transaction are about the same as three compare-and-swap operations [RB13, YHLR13]. In contrast, EnfoRSer's approach avoids atomic operations at most accesses, most likely achieving substantially lower overhead than an approach based on atomic operations or RTM transactions (except perhaps for high-conflict workloads). In **Chapter 6**, we empirically evaluate an implementation of SBRS using HTM and show evidence of the high run-time overhead incurred. Further, **Chapter 6** proposes a novel technique to overcome the limitations of enforcing SBRS using Intel's RTM and proposes an approach that outperforms EnfoRSer.

4.6 Summary

EnfoRSer leverages hybrid static–dynamic analysis to ensure that statically bounded, synchronization-free regions execute atomically on commodity systems. In this work, we showcase the benefit of a strong memory model based on atomicity of code regions. We demonstrate that strong semantics can be achieved at a reasonable efficiency in commodity systems and EnfoRSer can help make SBRS the default memory model.

Until now we have explored the two techniques to enforce atomicity of statically bounded regions. These techniques require instrumentation at each memory access since they are based on two-phase locking. We could potentially develop techniques which avoid instrumentation at each memory access and still provide atomicity of regions. Our overall goal is to provide strong semantics on commodity hardware. While these software techniques are effective, efficient conflict detection remains an issue for high-conflicting programs. Potentially, hardware transactional memory (HTM) could be leveraged for more efficient conflict detection using specialized hardware support although HTM possesses several other challenges. In the remainder of the dissertation, we discuss and evaluate solutions to the aforementioned challenges.

4.7 Impact and Meaning

The key implication of this work lies in the bug elimination coverage of SBRS at the moderate run-time cost that it incurs. The evaluation of SBRS's strength shows that most of the bugs exist within the purview of a bounded region—justifying the design. End-to-end guarantees based on serializability simplifies reasoning at the level of individual operations in the compiler or the hardware. SBRS via EnfoRSer provides practical evidence as a case in point for simplified semantics and efficient region serializability on commodity hardware.

Chapter 5: Hybrid Synchronization for Statically Bounded Region Serializability

In **Chapter 3**, we introduced a memory model called *dynamically bounded region serializability* (DBRS) that guarantees serializability of regions that are intraprocedural, acyclic, and synchronization free. **Chapter 4** introduces an enforcement mechanism for DBRS called *EnfoRSer* that transforms the compiled program so that it acquires a *per-object lock* at each memory access. Dynamically bounded region serializability (DBRS) is described as statically bounded region serializability (SBRS) in **Chapter 4** and in this chapter since the compiler transformations used—require the regions to be statically bounded. A difficult-to-avoid cost of EnfoRSer is that instrumentation must acquire a lock at most memory accesses. Furthermore, EnfoRSer modifies the compiler to transform statically bounded regions to execute either idempotently or speculatively; the transformed code adds run-time overhead (Section 5.1). In this work, we refer to EnfoRSer's per-object locks as *dynamic locks* because they are associated with run-time objects, and we refer to the version of EnfoRSer that uses dynamic locks as *EnfoRSer-D*.

The goal here is to provide SBRS with lower overhead than the technique discussed in the previous chapter. The high-level direction is to reduce the instrumentation cost for regions that conflict rarely, if ever. The mechanism for reducing instrumentation is to enforce atomicity using coarse-grained locking at the region level, instead of the object

level. In order for a region to ensure atomicity by acquiring a lock, the approach must ensure that two regions acquire the same lock if they might race with each other (i.e., if the regions each have one of a pair of potentially racy accesses). We refer to these locks as *static locks* because they are associated with static program sites. A *site* is a static memory access; it is identified uniquely by a method and a bytecode index. We refer to EnfoRSer that uses static locks to guard regions as *EnfoRSer-S*. EnfoRSer-S's approach often leads to over-synchronization that harms scalability significantly. Static locks thus must be applied judiciously: they should be applied to a region only when they will not incur significant unnecessary contention; other regions should use dynamic locks.

We thus introduce a *hybrid* version of EnfoRSer called *EnfoRSer-H* that selectively applies static and dynamic locks. EnfoRSer-H makes use of an *assignment algorithm* that chooses, for each site, whether to use static or dynamic locks, subject to the constraint that all sites that race with each other must use the same kind of locking. The assignment algorithm makes its decisions based on a cost model and run-time profiling information.

We have implemented EnfoRSer-S and EnfoRSer-H in Jikes RVM, a high-performance Java virtual machine (JVM) [AAB+00, AAB+05], on top of our existing EnfoRSer-D implementation (Section 4.3) [SBZ+15]. To avoid the engineering challenge of invalidating and recompiling all methods affected by run-time locking changes, our evaluation instead uses a methodology that runs two iterations of the application: the first iteration collects profile information using static locks only; then the JVM recompiles all methods based on the assignment algorithm; and finally the second iteration executes using this new locking scheme.

Our evaluation compares the run-time characteristics and performance of EnfoRSer-D, EnfoRSer-S, and EnfoRSer-H on benchmarked versions of large, real, multithreaded

applications [BGH+06]. EnfoRSer-D's use of dynamic locks minimizes contention but incurs lock acquire overhead at each potentially racy memory access. On the other hand, EnfoRSer-S's exclusive use of static locks reduces the number of lock acquire operations but leads to high contention and thus substantial run-time slowdowns. By hybridizing dynamic and static locks, EnfoRSer-H is able to get the benefits of both approaches. Most programs cannot benefit much from static locks, so EnfoRSer-H performs almost identically to EnfoRSer-D; notably, EnfoRSer-H does not harm performance in these cases but instead correctly chooses to use dynamic locks. Since most programs fall into this category, EnfoRSer-H's overall improvement over EnfoRSer-D is modest: EnfoRSer-H incurs 26% run-time overhead on average, compared with 27% for EnfoRSer-D. However, for a few programs that can benefit from static locks, EnfoRSer-H's selective use of static locks leads to substantially lower run-time overhead than EnfoRSer-D. Overall, EnfoRSer-H demonstrates opportunities for exploiting low-contention parts of applications to provide a strong memory model, SBRS, with lower overhead than the previous technique.

5.1 Motivation

EnfoRSer-D's run-time costs. EnfoRSer-D transforms regions to perform fine-grained locking at the object level. This approach avoids contention and achieves high scalability because locks only conflict when regions perform conflicting accesses to objects. However, EnfoRSer-D's approach has two main drawbacks that lead to high run-time overhead. First, EnfoRSer-D adds instrumentation costs to every program memory access (except for accesses that are statically redundant in a region or statically data race free; Section 5.3). Although biased reader–writer locks provide low overhead for the common case—non-conflicting lock acquires—this overhead is still significant when incurred at nearly every

memory access [BKC+13]. Second, EnfoRSer-D's atomicity transformations (the idempotent and speculation transformations) add additional run-time costs in order to support retrying regions (refer **Chapter 4**) [SBZ+15].

These two main run-time costs of EnfoRSer-D can be seen in Figure 4.5. First, each memory load and store is preceded by a lock acquire operation. Second, EnfoRSer-D has transformed the region to execute speculatively: the region backs up stored-to memory locations and locals, and it adds an undo block. Although by design the undo block executes infrequently, the conditional control flow edges add additional complexity that limits optimizations within the region. Furthermore, the additional local variables used to back up values are live into the undo block, which adds register pressure.

The rest of this chapter explores an alternative to EnfoRSer-D's transformation that acquires *a single lock* for the entire region, avoiding both instrumentation at every memory access and transformations that ensure atomicity. However, acquiring a lock on the whole region—which involves acquiring the same lock for every pair of regions that potentially race with each other—adds high contention in many cases. The challenge is to apply these region locks judiciously.

5.2 Hybridizing Static and Dynamic Locks

This section describes the design of *EnfoRSer-H*, a version of EnfoRSer that enforces SBRS through a combination of per-object dynamic locks and per-region static locks. The high-level intuition of EnfoRSer-H is that using *static* locks to enforce SBRS incurs less instrumentation overhead than using dynamic locks (see the previous section for details of the overheads introduced by EnfoRSer-D), but using *dynamic* locks means that regions are less likely to conflict. Hence, EnfoRSer-H adopts a hybrid scheme that uses static locks

to provide atomicity where we expect few conflicts (reducing the likelihood of expensive contention), while using dynamic locks when regions often execute concurrently (where static locks would result in significant contention). The following sections describe: (a) how SBRS can be enforced using per-site static locks, and how static locks be coarsened to the granularity of regions to reduce overhead; (b) how SBRS can be enforced using a combination of static and dynamic locks; and (c) a policy for selecting the right combination of static and dynamic locks.

5.2.1 EnfoRSer-S: Enforcing SBRS with Static Locks

We first present a version of EnfoRSer that enforces SBRS with static locks only, called EnfoRSer-S. EnfoRSer-S operates as follows: first, it instantiates a global set of static locks, L, for the program. Next, it associates *each access site* (hereafter called "site" for brevity) in a region with one of those static locks; the set of static locks acquired in a region r is $L(r)$. Because $L(r)$ is statically known, EnfoRSer-S can acquire all of the locks in the set at the beginning of the region, in some canonical order. Like in EnfoRSer-D, a region does not allow locks to be released in the middle of the region.

This strategy provides two-phase locking while avoiding the possibility of deadlock (because of the canonical ordering of static locks), without the need for complex atomicity transformations, as in EnfoRSer-D. However, this strategy alone does not suffice to guarantee atomicity. Consider two regions r_1 and r_2, with a site s_1 in r_1 that reads an object o, while a site s_2 in r_2 writes to that same object. To enforce SBRS, EnfoRSer-S must ensure that r_1 and r_2 cannot execute simultaneously. Note that EnfoRSer-D inherently provides this guarantee, due to its dynamic, per-object locking: because both s_1 and s_2 access o, they will both attempt to acquire a lock on o, triggering a conflict that EnfoRSer-D arbitrates.

EnfoRSer-S, on the other hand, uses static, *per-site* locks; we must ensure that the locks that EnfoRSer-S acquires in r_1 and r_2 prevent the regions from executing simultaneously.

We note that a pair of sites $\langle s_1, s_2 \rangle$ presents a problem for enforcing SBRS only if three conditions are met: (i) both sites might access the same object; (ii) at least one of those accesses writes to the object; and (iii) the regions containing s_1 and s_2 could execute simultaneously in the original program. If any of those three conditions are not met, then executing r_1 and r_2 simultaneously does not violate atomicity. In other words, the problem arises if and only if s_1 and s_2 *race*. Hence, EnfoRSer-S can guarantee SBRS by ensuring that all pairs of sites that could race with each other *use the same static lock*. In other words, for any two regions r_a and r_b that contain sites s_a and s_b that race, EnfoRSer-S can guarantee SBRS by ensuring that $L(r_a) \cap L(r_b) \neq \emptyset$.

Hence, EnfoRSer-S uses the following strategy for *static lock assignment* when choosing which lock to associate with each site. It begins by using a sound static race detector to determine which sites might race with each other. EnfoRSer-S uses this information to construct an equivalence relation, $RACE$, over the set of sites in the program where $s_1\ RACE\ s_2$ if and only if:

1. s_1 races with s_2 according to the race detector (which is a symmetric property); or
2. $s_1 = s_2$; or
3. $\exists s_3 \,.\, s_1\ RACE\ s_3 \wedge s_2\ RACE\ s_3$

Note that $s_a\ RACE\ s_b$ does *not* imply that s_a and s_b race with each other according to the static race detection analysis. $RACE$ is an equivalence relation that adds reflexivity and transitivity to the symmetric property "(potentially) races with."

Then, for each equivalence class in $RACE$, EnfoRSer-S assigns a single static lock.[10] Hence, for each site s, EnfoRSer-S assigns it some lock l, which is associated with s's equivalence class in $RACE$, and every site s' that races with s has been assigned l as well. When this lock assignment algorithm is used with the locking strategy outlined above, EnfoRSer-S enforces SBRS using only static locks while avoiding deadlock.

Coarsening static locks. As described so far, the locking strategy for EnfoRSer-S requires that a lock be acquired for *every site* in a region. Acquiring a lock for every site in a region—including those that do not execute due to control flow—is likely to add overhead similar to EnfoRSer-D's, which also acquires a lock for every memory access. To mitigate this overhead, we observe that locks can be *coarsened*: while each equivalence class in $RACE$ must use a single lock to ensure correctness, there is no reason that different equivalence classes must be assigned different locks.

EnfoRSer-S thus uses a simple, region-based strategy for lock coarsening: in the pursuit of low overhead, it uses a *single* static lock for each region. In other words, all sites in a given region are assigned the same lock. To formalize this, let us define the $SRSL$ relation as an equivalence relation where $s_1\ SRSL\ s_2$ if and only if s_1 and s_2 are in the same region.[11] $SRSL \cup RACE$ is also an equivalence relation, and EnfoRSer-S ensures that each equivalence class in this combined relation uses the same lock.

[10]Note that some sites may not appear in RACE at all. These sites do not require locks, as they cannot lead to an SBRS violation.

[11]$SRSL$ is an acronym for "same region (static locks)." It applies only to sites that use static locks, not dynamic locks. This distinction is irrelevant for EnfoRSer-S, which only uses static locks, but is relevant for EnfoRSer-H (Section 5.2.2).

Hence, each region r acquires a *single* lock when it begins, and any region r' that races with r (i.e., contains a site that races with a site in r) will acquire the same, single static lock when it begins.

5.2.2 EnfoRSer-H: Enforcing SBRS with Static and Dynamic Locks

While enforcing SBRS with EnfoRSer-S is sound, we note that it can be overly conservative in deciding whether to prevent two regions from executing simultaneously, and hence *over-serialize* execution. If two regions contain sites that *may* race with each other according to the static race detector, EnfoRSer-S will introduce locks that prevent those regions from executing concurrently. However, this locking is subject to three sources of imprecision:

1. Because of control flow in regions, locks may be acquired that prevent two regions from executing simultaneously even though during that particular execution of the regions, one or both of the racing sites may not actually execute.

2. Static race detectors identify sites that *may* race. At run time, these races may not occur because either (a) the inherent imprecision of static analysis means that these sites *never* race, or (b) even though under some circumstances the sites race, in this particular execution the two sites access different objects and so no race occurs.

3. Because EnfoRSer-S assigns locks according to the $RACE$ relation (more precisely, according to the $RACE \cup SRSL$ relation), two sites may use the same lock because they each race with a third site, even though these two sites can never race with each other.

Note that these sources of imprecision are not equally problematic: acquiring static locks when those locks are unlikely to result in conflicts does not cause much contention, while acquiring static locks that introduce unnecessary conflicts, results in over-serialization. Note, too, that none of these sources of imprecision afflict EnfoRSer-D: its per-object locks always acquire exactly the locks necessary to prevent two conflicting regions from executing simultaneously.

We thus introduce EnfoRSer-H, a *hybrid* version of EnfoRSer that uses both static locks at the region level and dynamic locks at the memory access level. For each equivalence class in $RACE \cup SRSL$, EnfoRSer-H uses either static or dynamic locks, depending on the tradeoff between instrumentation overhead (which favors static locks) and likelihood of conflict (which favors dynamic locks).

We note that the $SRSL$ ("same region (static locks)") relation applies only to accesses that use *static locks*. If two sites s_1 and s_2 are in the same region, s_1 $SRSL$ s_2 only if both s_1 and s_2 use static locks. $SRSL$ is thus defined in part according to EnfoRSer-H's choice of static versus dynamic locks for each site.

EnfoRSer-H proceeds directly from the definition of EnfoRSer-S. EnfoRSer-S uses a single, static lock to protect each equivalence class of $RACE \cup SRSL$. EnfoRSer-H allows some of those equivalence classes to be protected using EnfoRSer-D's dynamic, per-object locks instead. In other words, each site in the region is either protected by a static lock (which is acquired at the beginning of the region, a la EnfoRSer-S) or by a dynamic lock (which is acquired on demand, right before the access, a la EnfoRSer-D). As long as all the sites in a $RACE \cup SRSL$ equivalence class use the same *type* of lock (static or dynamic), atomicity of regions is still ensured.

Note that, if *any* site in a region uses dynamic locks, the site must be transformed using one of EnfoRSer-D's atomicity transformations to support restart. For regions that have a mix of static and dynamic locks, the static locks are acquired before the transformed region executes.

5.2.3 Choosing Which Locks to Use

Section 5.2.2 described how EnfoRSer-H can operate with different sites using different types of locks to enforce atomicity and hence provide SBRS. An obvious question to ask is how to determine which sites should use which type of lock. Static locks should be used when excessive serialization is unlikely—when sites do not often conflict. Dynamic locks, on the other hand, should be used when the benefit of determining more precisely *when* regions conflict makes up for higher instrumentation overheads. Because these are run-time properties, EnfoRSer-H uses profiling to collect information about the behavior of regions in the program. This profiling information is then used by a lock *assignment algorithm* that determines which type of lock should protect each site. This section describes that process.

Profiling data. Run-time profiling collects three pieces of data:

1. for each site s, the execution frequency of the region containing s;
2. for each site s, the number of conflicting lock acquires that occur when using a static lock, based on the $RACE$ equivalence class, in order to protect s's region; and
3. a mapping from sites to the region(s) that contain them statically (sites may appear in multiple regions statically, e.g., because of code expansion optimizations such as inlining).

The first two pieces of data are execution frequencies computed by compiler-inserted instrumentation. The third is computed by the just-in-time compiler.

In order to collect this data, each method is first compiled so that each site in a region is guarded by a static lock associated with its $RACE$ equivalence class—which is equivalent to EnfoRSer-S without considering the $SRSL$ relation. The compiler inserts instrumentation at each static lock acquire to count execution frequency and the frequency of conflicting acquires. The compiler updates the mapping from sites to regions as it compiles each region.

Assigning locks. Under EnfoRSer-H, some sites are guarded by static locks while other sites use dynamic locks. The number of possible configurations for a hybrid implementation is 2^N, where N is the number of equivalence classes in the $RACE$ equivalence relation. Each equivalence class can be independently set to use static locks or dynamic locks, but all of the sites in an equivalence class must use the same type of lock for correctness. Note that in practice, there are fewer than 2^N possible configurations: for a given configuration, the use of the $SRSL$ relation to perform lock coarsening means that equivalence classes in $RACE$ that have sites that appear in the same region must use the same type of lock; assigning one $RACE$ class to use static locks may ultimately require that several other $RACE$ classes also use static locks.

EnfoRSer-H uses a greedy algorithm, as shown in Algorithm 1, to determine which $RACE$ classes should use static locks and which should use dynamic locks. The algorithm begins by assuming that all regions use static locks. Based on this assumption, it computes the associated $SRSL$ relation. Starting with this assignment, the algorithm computes an *estimated cost* according to a function estimateCost. This estimated cost is a function of

Algorithm 1 EnfoRSer-H's lock assignment algorithm.

```
 1: buildSRSLRelation()
 2: initialCost ← estimateCost()
 3: for Site s in RACE do
 4:     sType ← s.getLockType()
 5:     if sType is static lock type then
 6:         s.setLockType(DynamicLockType)          # Change the lock type to per-object type
 7:         markAllSitesInRaceEqvClassAsDynamicLockType(s)
 8:         buildSRSLRelation()
 9:         currentCost ← estimateCost()
10:         if currentCost ≤ initialCost then
11:             initialCost ← currentCost
12:         else
13:             s.setLockType(StaticLockType)
14:             resetAllSitesInRaceEqvClassAsStaticLockType(s)
15:             buildSRSLRelation()
16:         end if
17:     end if
18: end for
```

the expected number of executed lock acquires and conflicting lock acquires. This section describes estimateCost in more detail below.

Given a lock configuration and an estimated instrumentation cost, the algorithm proceeds greedily. It iterates through each $RACE$ equivalence class, switching all of the class's sites to use dynamic locks (if they have not already been assigned dynamic locks). The algorithm propagates the effects of this switch by recalculating the $SRSL$ relation (since sites using dynamic locks are not included in $SRSL$). The algorithm then computes the cost of this new configuration. If the cost is less than the previous configuration, then the algorithm continues on to the next iteration with the new configuration, i.e., with the sites in the current $RACE$ equivalence class using dynamic locks. Otherwise, the algorithm continues with the previous configuration, i.e., with the sites in the current $RACE$ equivalence class using static locks.

While this greedy algorithm may not find the optimal (i.e., lowest estimated cost) configuration, our experience is that it is effective in finding better configurations when they exist. For example, our evaluation shows that in cases where profiling indicates that static locks incur many conflicts, the algorithm correctly chooses an assignment that instruments the program with mostly dynamic locks.

Estimating execution cost. At a high level, EnfoRSer-H predicts the costs that an execution will incur from acquiring locks, given profiling data and a candidate assignment of locks to be static or dynamic.

The basic intuition behind the estimate are two sources of overhead when enforcing SBRS using EnfoRSer-H:

1. The *fast-path cost*. This is the cost of acquiring the necessary locks to execute a region, assuming conflict freedom. For a region r, its estimated fast-path cost is:

   $$C_{fp}(r) = locks(r) \times T_{fp}$$

   where T_{fp} is the fast-path cost factor, and $locks(r)$ is estimated as follows.

   If the region r is protected by a static lock, one static lock acquire executes for the entire region. The function estimates this frequency as the *maximum* frequency of all sites in the region:

   $$locks(r) = \max_{s \in r}(\text{freq}(s))$$

   where $\text{freq}(s)$ is number of times that a lock executed for site s according to profiling.

   If the region is protected by dynamic locks, the fast-path cost is estimated by summing the frequencies of all sites in the region:

   $$locks(r) = \sum_{s \in r}(\text{freq}(s))$$

although it is an overestimate because it does not account for control flow.

2. The *conflict cost*. This is the cost of a conflict when a lock's ownership must be transferred to a new thread (Section 5.1). For a region r, its estimated conflict cost is:

$$C_{\mathit{confl}}(r) = \mathit{conflicts}(r) \times T_{\mathit{confl}}$$

where T_{confl} is the conflict cost factor, and $\mathit{conflicts}(r)$ is estimated as follows.

If the region r is protected by a static lock, our experience suggests that merging locks using the $SRSL$ relation leads to a higher conflict rate than for the maximum conflicts of all sites in the region, so we instead compute the *sum* of all sites' conflicts in the region:

$$\mathit{conflicts}(r) = \sum_{s \in r} (\text{conflicts}(s))$$

where conflicts(s) is the number of conflicting lock acquires for s according to profiling.

If r is protected by a dynamic lock, the algorithm cannot estimate the number of conflicts for dynamic locks because EnfoRSer-H collects profiling information using static locks based on $RACE$. Since dynamic locks tend to conflict infrequently compared to static locks, we assume no conflicts for dynamic locks:

$$\mathit{conflicts}(r) = 0$$

Summing up all these costs across all regions provides an estimate of the instrumentation cost for a given lock assignment. Section 5.2.4 discusses the impact of the two cost factors, T_{fp} and T_{confl}.

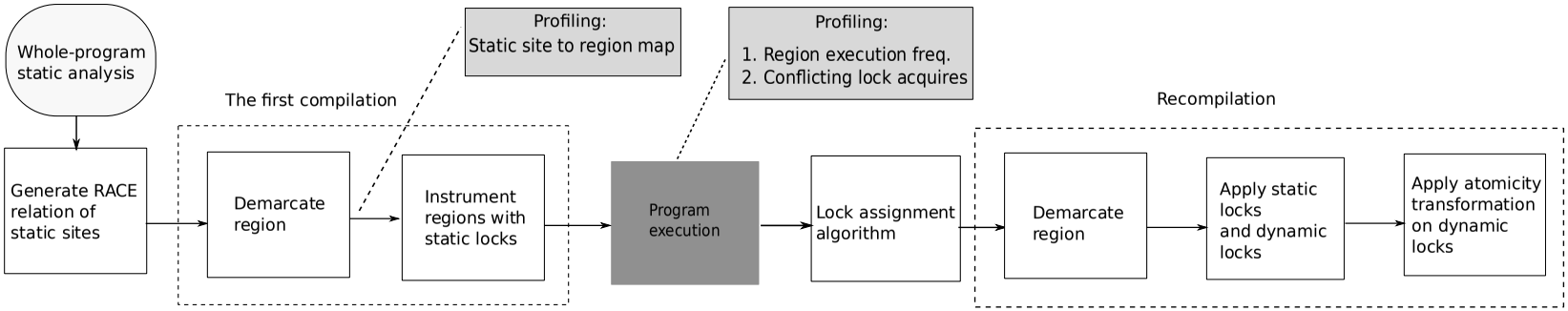


Figure 5.1: EnfoRSer-H's optimization pipeline.

Overview of EnfoRSer-H's optimization pipeline. Figure 5.1 illustrates EnfoRSer-H's optimization process. Each method is initially compiled so that each of its regions uses static locks based solely on the $RACE$ relation, as well as counters for collecting profile information. The second compilation uses the results of the assignment algorithm to use the chosen combination of static and dynamic locks.

5.2.4 Improving the Cost Estimate

This section describes a few extensions to the estimateCost function that help to improve its accuracy.

Conservative conflict prediction. We note that EnfoRSer-H's profiling counts a site's conflicts using static locks merged according to the $RACE$ equivalence class, but EnfoRSer-H chooses static locks so that all sites in the $RACE \cup SRSL$ equivalence class will use the same static lock. As a result, any two sites in this equivalence class that did not conflict with each other during profiling, may now conflict with each other with the coarsened locks.

To provide a better prediction of this effect, the estimateCost function computes the number of predicted conflicts for a site by taking the maximum number of conflicts measured for any site in the same $RACE \cup SRSL$ equivalence class:

$$computedConflicts(s) = \max_{s \in e} \text{profiledConflicts}(s)$$

where e is the $RACE \cup SRSL$ equivalence class for s.

Considering redundant lock acquires in the model. EnfoRSer-D and EnfoRSer-H use an optimization that removes redundant dynamic locks: if an earlier access in a region guarantees that the appropriate lock will already be held for a given access, the lock acquire for the latter access can be elided (Section 4.2) [SBZ+15]. The cost model accounts

for this effect by internally performing this *redundant lock elimination* optimization when computing the estimated fast-path cost for a region.

Ratio of costs of a conflict and a fast path. The two parameters in the cost model that drive the instrumentation overhead estimates are the cost of a fast path, T_{fp}, and the cost of a conflict on statically-locked regions, T_{confl}. The ratio of these two parameters captures how biased the assignment algorithm is toward static versus dynamic locks: static locks incur conflicts, and are selected against if T_{confl} is high, while dynamic locks require more fast paths, and are selected against if T_{fp} is high. Adjusting the ratio between these parameters leads to different lock assignment outcomes. In our experiments, we have found that EnfoRSer-H's performance is quite stable across a wide range for the ratio of T_{confl} to T_{fp} (50–300), suggesting that it is most important to identify a key set of outlier regions when performing lock assignment.

5.3 Implementation

We have implemented the EnfoRSer configurations in Jikes RVM 3.1.3 [AAB+00, AAB+05],[12] a Java virtual machine that provides performance competitive with commercial JVMs [BZBL15]. Our implementation builds on the publicly available implementation of our previous work, which provides only the EnfoRSer-D algorithm as discussed earlier (Section 4.3) [SBZ+15]. We have now made our implementations of EnfoRSer-D, EnfoRSer-S, and EnfoRSer-H publicly available on the Jikes RVM Research Archive.[13]

Jikes RVM uses two just-in-time compilers at run time. The first time a method executes, Jikes RVM compiles it with the *baseline* compiler, which generates unoptimized

[12] `http://www.jikesrvm.org`

[13] `http://www.jikesrvm.org/Resources/ResearchArchive/`

machine code directly from Java bytecode. When a method becomes hot, Jikes RVM compiles it with the *optimizing* compiler at successively higher optimization levels. As in the EnfoRSer-D's implemenetation, we modify *only the optimizing compiler* to enforce SBRS, since EnfoRSer's transformations require a compiler internal representation (Section 4.1.5). While this approach is technically unsound, it approximates the performance of a fully sound implementation because, by design, execution spends most of its time in optimized code.

As in the EnfoRSer-D's implemenetation (Section 4.1.1), the compiler limits reordering across region boundaries (synchronization operations, method calls, and loop back edges) in order to preserve SBRS. After letting the compiler perform some standard optimizations, the optimizing compiler performs EnfoRSer transformations, followed by additional standard optimizations, which help to "clean up" the code generated by EnfoRSer's transformations [SBZ+15].

5.3.1 Methodology

Profiling and recompilation. EnfoRSer-S and EnfoRSer-H require profile information. Both EnfoRSer-S and EnfoRSer-H require information from the optimizing compiler about the $SRSL$ relation, i.e., which sites are compiled into which regions. Although in theory $SRSL$ could be computed statically, in practice inlining decisions and other optimizations expand regions and increase sites which appear in the same statically bounded regions. In addition, EnfoRSer-H relies on run-time profiling to compute the number of lock acquires and conflicts when using static locks. Both EnfoRSer-S and EnfoRSer-H initially choose static locks based only on the $RACE$ relation, which enables collecting run-time profile information about conflicts within each $RACE$ equivalence class.

In theory, an implementation could perform online profiling, updating $SRSL$ and adjusting the locking strategy for sites on the fly. This approach would require recompiling all methods affected by the use of new locks and new locking strategies. While such an approach is possible, we have not undertaken this engineering effort. Instead, our implementation runs two iterations of each program. The first iteration collects profile information: it executes code compiled entirely with static locks based only on the $RACE$ equivalence class. Note that the $SRSL$ equivalence classes are not known during the first iteration, so the first iteration cannot use code compiled based on $SRSL$.

After the first iteration completes, Jikes RVM recompiles all methods that have been compiled by the optimizing compiler; when the optimizing compiler recompiles a method during this phase, it uses the profile information from the first iteration. For the EnfoRSer-S configuration, the compiler uses the $SRSL$ relation in order to compute static region locks based on $RACE \cup SRSL$. EnfoRSer-H runs the assignment algorithm, which uses run-time profile information about static locks in order to decide whether to use static or dynamic locks for each site. For sites that use static locks, EnfoRSer-H uses the $RACE \cup SRSL$ relation in order to use one lock for each region while preserving correctness. EnfoRSer-D ignores the profiling information and uses dynamic per-object locks, and is thus equivalent to our prior work [SBZ+15].

The second iteration then executes using the recompiled methods. Our evaluation reports the cost of the second iteration only.

This two-iteration methodology thus represents an optimistic performance measurement: it excludes compilation time and the cost of the assignment algorithm, and it uses profile information on a prior identical run. On the other hand, this methodology does not

account for the fact that EnfoRSer-S and EnfoRSer-H have the potential to *reduce* compilation time (and thus execution time, since JIT compilation can affect program execution time) by avoiding EnfoRSer-D's complex atomicity transformations.

	EnfoRSer-D		**EnfoRSer-S**		**EnfoRSer-H**					
	Total		Total		Static locks		Dynamic locks		Total	
	Executed	Confl.	Executed	Confl.	Executed	Confl.	Executed	Confl.	Executed	Confl.
hsqldb6	4.7×10^{8}	5.9×10^{5}	2.1×10^{8}	5.2×10^{5}	2.3×10^{7}	3.8×10^{3}	4.4×10^{8}	5.6×10^{5}	4.7×10^{8}	5.7×10^{5}
lusearch6	1.5×10^{9}	3.9×10^{3}	3.8×10^{8}	3.3×10^{8}	4.1×10^{7}	3.7×10^{2}	1.4×10^{9}	3.6×10^{3}	1.4×10^{9}	4.0×10^{3}
xalan6	6.1×10^{9}	1.4×10^{7}	2.3×10^{9}	1.5×10^{9}	2.0×10^{8}	1.7×10^{2}	6.1×10^{9}	1.5×10^{7}	6.3×10^{9}	1.5×10^{7}
avrora9	4.0×10^{9}	4.0×10^{6}	1.1×10^{9}	7.9×10^{8}	3.8×10^{8}	1.0×10^{1}	3.9×10^{9}	4.0×10^{6}	4.3×10^{9}	4.0×10^{6}
luindex9	1.3×10^{8}	7.1×10^{1}	6.8×10^{7}	1.8×10^{1}	6.0×10^{7}	3.0×10^{0}	6.5×10^{6}	3.3×10^{1}	6.6×10^{7}	3.6×10^{1}
lusearch9	1.5×10^{9}	1.2×10^{4}	4.7×10^{8}	4.4×10^{8}	1.0×10^{8}	3.1×10^{3}	1.4×10^{9}	9.7×10^{2}	1.5×10^{9}	4.0×10^{3}
sunflow9	7.2×10^{9}	1.6×10^{4}	3.3×10^{9}	6.0×10^{7}	2.1×10^{9}	3.2×10^{2}	2.3×10^{9}	1.2×10^{4}	4.4×10^{9}	1.2×10^{4}
xalan9	4.2×10^{9}	2.0×10^{7}	1.3×10^{9}	6.2×10^{8}	1.2×10^{8}	3.4×10^{4}	4.1×10^{9}	2.0×10^{7}	4.2×10^{9}	2.0×10^{7}
pjbb2000	1.2×10^{9}	9.5×10^{5}	6.2×10^{8}	2.6×10^{8}	1.2×10^{8}	1.4×10^{3}	1.1×10^{9}	9.5×10^{5}	1.2×10^{9}	9.5×10^{5}
pjbb2005	6.3×10^{9}	4.0×10^{8}	1.8×10^{9}	1.0×10^{9}	1.1×10^{9}	2.5×10^{3}	4.4×10^{9}	4.2×10^{8}	5.5×10^{9}	4.2×10^{8}

Table 5.1: Dynamic lock acquire operations executed, and the number of them that result in a conflicting lock state transition requiring coordination among threads. For EnfoRSer-H, the table shows the breakdown for static and dynamic locks.

5.4 Evaluation

This section compares the run-time characteristics and performance of EnfoRSer-D, EnfoRSer-S, and EnfoRSer-H.

Static race detection. EnfoRSer-S and EnfoRSer-H rely on conservative static data race detection in order to compute the $RACE$ relation. We use Naik et al.'s whole-program static data race detector, implemented as the publicly available tool *Chord*.[14] We disable Chord's lockset analysis, which is unsound (has false negatives) because it uses may-alias analysis [NAW06].[15] The resulting analysis identifies accesses as data race free (DRF) based on static thread escape analysis and fork–join analysis [NAW06]. We use Chord's built-in mechanism to handle reflection, which resolves reflective calls by running the input program prior to the static analysis.

In prior work, we used the same configuration of Chord in order to identify definitely DRF accesses, which can forgo EnfoRSer-D locks (Section 4.2) [SBZ+15]. For a fair comparison, our evaluation continues to perform this optimization for EnfoRSer-D.

Benchmarks. We evaluate the EnfoRSer implementations using benchmarked versions of large, real-world multithreaded applications: the DaCapo benchmarks [BGH+06] versions 2006-10-MR2 and 9.12-bach (2009), distinguished with names suffixed with '6' and '9', using the large workload size; and fixed-workload versions of SPECjbb2000 and SPECjbb2005.[16] We exclude benchmarks that unmodified Jikes RVM cannot execute. In

[14] `http://pag.gatech.edu/chord`

[15] We have confirmed with Naik that the sound version of Chord's analysis, which uses conditional must-not alias analysis, is not available [NA07].

[16] `http://www.spec.org/jbb200{0,5}`, `http://users.cecs.anu.edu.au/~steveb/research/research-infrastructure/pjbb2005`

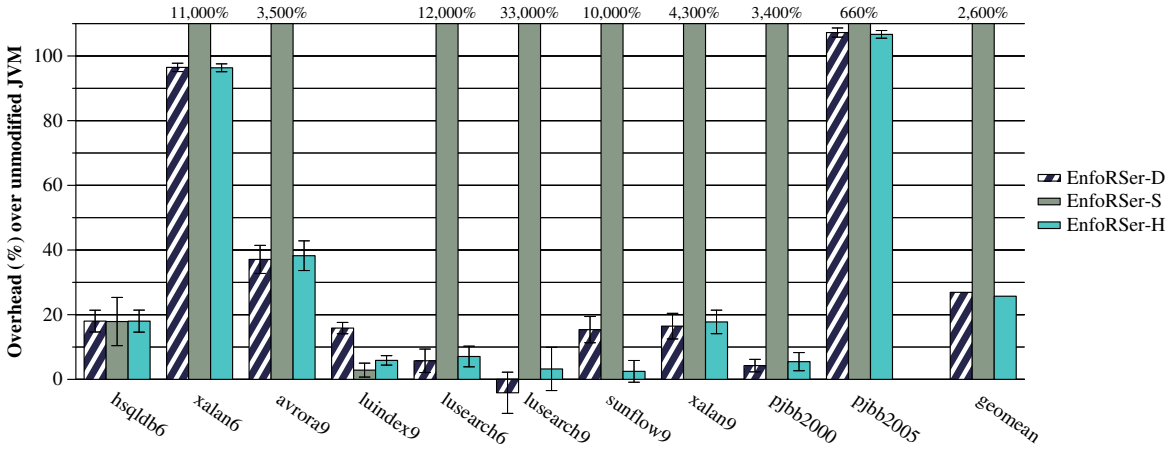


Figure 5.2: Run-time overhead of providing SBRS with EnfoRSer configurations that use dynamic locks, static locks, or a hybrid of both.

addition, we exclude eclipse6 since Chord fails when analyzing it. We exclude jython9 and pmd9 since Chord does not report any potential data races in these programs; we suspect that jython9, which has limited multithreaded behavior, in fact has no data races, while Chord does not understand pmd9's multithreading based on futures. We cross-checked Chord's results with a dynamic race detector's output [FF09, BCM10]; we found one class (in jbb2005) that Chord does not analyze at all (for unknown reasons), so EnfoRSer-S, EnfoRSer-D and EnfoRSer-H fully instrument this class.

Platform. The experiments execute a high-performance configuration of Jikes RVM with the default high-performance garbage collector (FastAdaptiveGenImmix). The experiments execute on an Intel Xeon E5-4620 system with four 8-core processors (32 cores total) running Linux 2.6.32.

5.4.1 Run-Time Characteristics

Table 5.1 reports how many lock acquires execute, and how many of them incur a conflicting transition, for each EnfoRSer configuration. Each result is the mean of five trials of a statistics-gathering configuration of EnfoRSer-D, EnfoRSer-S, or EnfoRSer-H. For each configuration, the first column under *Total* reports the total number of lock acquire operations executed, and the second column reports how many of those operations result in a conflicting lock state transition, requiring coordination among threads.

As expected, EnfoRSer-S acquires a single lock per region, rather than per memory access, so it performs fewer lock acquires than EnfoRSer-D. However, EnfoRSer-S incurs many more conflicts than EnfoRSer-D, except for two programs (hsqldb6 and luindex9). EnfoRSer-S incurs many more conflicts than EnfoRSer-D because of the imprecision of detecting conflicts wtih static locks, for reasons described in Section 5.2.2.

For luindex9 and sunflow9, EnfoRSer-H achieves its desired goal, reducing lock acquires by 39–49% relative to EnfoRSer-D, without substantially affecting the fraction of conflicts, which is already very low. As Section 5.4.2 shows, EnfoRSer-H is able to lower run-time overhead relative to EnfoRSer-D for these programs.

For other programs, EnfoRSer-H does not substantially affect how many lock acquire operations execute compared with EnfoRSer-D. At the same time, EnfoRSer-H does not significantly increase total lock acquires nor conflicting lock acquires. EnfoRSer-H does increase the total number of lock acquires slightly for xalan6 and avrora9 compared with EnfoRSer-D; this possibly unintuitive result is a side effect of the fact that EnfoRSer-H's static lock acquires execute at the beginning of a region (i.e., they are not sensitive to control flow in the region), while EnfoRSer-D's dynamic lock acquires execute only when their corresponding access executes.

We note that for pjbb2005, EnfoRSer-H reduces lock acquire operations compared with EnfoRSer-D. However, EnfoRSer-H also incurs more conflicts than EnfoRSer-D; the rate of conflicts is high for pjbb2005, so even a modest increase can incur high overhead, which may explain why our performance evaluation shows no statistically significant difference in overhead between the two configurations for pjbb2005 (Section 5.4.2).

5.4.2 Performance

Figure 5.2 shows the overhead of EnfoRSer-D, EnfoRSer-S, and EnfoRSer-H over baseline execution. Each bar is the mean of at least 10 trials; each bar has a 95% confidence interval centered at the mean.[17] As in Section 5.4.1, for each configuration including the baseline (unmodified Jikes RVM), Jikes RVM executes two iterations. The first iteration always uses static locks based solely on the $RACE$ relation. Before the second iteration, the optimizing compiler recompiles methods according to the EnfoRSer configuration: EnfoRSer-S and EnfoRSer-H compute $RACE \cup SRSL$ equivalence classes, and EnfoRSer-H uses the assignment algorithm to choose a combination of static and dynamic locks. In the baseline configuration, the optimizing compiler recompiles all methods without any instrumentation.

For most programs, EnfoRSer-S incurs substantially higher overhead than EnfoRSer-D, which is a result of EnfoRSer-S causing many more conflicting lock acquires than EnfoRSer-D (Table 5.1). EnfoRSer-S incurs an average overhead of 2600% overhead (a 27X slowdown) over baseline execution. This result shows that static locks must be applied judiciously in order to be effective.

[17]For some programs with high execution time variance, we have run additional trials in an effort to reduce confidence interval sizes.

EnfoRSer-S outperforms or matches EnfoRSer-D's performance for hsqldb6 and luindex9, which is to be expected from the low number of conflicts it occurs in Table 5.1.

Overall, EnfoRSer-H is able to improve the performance of enforcing SBRS compared with EnfoRSer-D, but the average improvement is modest: from 27% to 26%, a 4% reduction in overhead. This result is unsurprising given the results from Table 5.1: the reduction is modest overall, and significant benefits from EnfoRSer-H are limited to a few programs.

For two programs, luindex9 and sunflow9, EnfoRSer-H reduces overhead substantially compared with EnfoRSer-D, 63% and 87% reduction in overhead, respectively, over the baseline. Note that for luindex9, EnfoRSer-S also incurs low overhead compared to EnfoRSer-D since it reduces lock acquires without increasing the number of conflicting acquires, as this program inherently has few conflicts. This result follows directly from Table 5.1, which shows that EnfoRSer-H performs about half as many lock acquire operations as EnfoRSer-D for each of these programs.

For all other programs, EnfoRSer-H does not perform significantly worse than EnfoRSer-D. (EnfoRSer-H appears to perform slightly worse for xalan6, lusearch9, and xalan9, but the confidence intervals overlap.) This outcome results from EnfoRSer-H's assignment algorithm finding relatively few executing regions that can use static region locks without incurring significant contention (as Table 5.1 suggests), so it conservatively uses dynamic per-object locks in most cases, and thus EnfoRSer-H's performance resembles EnfoRSer-D's performance in these cases.

These results show the opportunities for and limitations of hybridizing locks to enforce SBRS efficiently. While static locks reduce the number of executed lock acquires compared with dynamic locks, their conservatism introduces many false conflicts. EnfoRSer-H judiciously uses static locks only where profiling and the cost model predict few enough

conflicts to not outweigh the reduction in lock acquires, achieving better performance than either EnfoRSer-D or EnfoRSer-S can achieve alone. While the overall performance improvement is modest, EnfoRSer-H generally does not hurt performance relative to EnfoRSer-D or EnfoRSer-S, and it has the potential to improve performance significantly in cases that can benefit from a hybrid of static and dynamic locks.

5.5 Summary

EnfoRSer-H further strengthens the techniques of providing strong semantics for common Java programs. Though the implementation is for Java, the techniques are general enough to be extended to other languages. The low-cost enforcement of SBRS in real-world programs is more practical with EnfoRSer-H. EnfoRSer already reduces the synchronization overhead for enforcement of SBRS by using lightweight reader-writer locks [BKC+13]. However, it does not focus on the instrumentation overhead, especially for benchmarks where atomicity with two-phase locking could be expensive. EnfoRSer-H bridges that gap by using profile guided instrumentation. It meets dynamic locking constraints by addressing the constraints based on static race detection results—to enforce SBRS correctly—while still using multiple synchronization mechanisms in a single execution.

EnfoRSer-D and EnfoRSer-H make SBRS practical on commodity hardware, but still, incurs significant overhead for programs with more that 0.1% of inter-thread conflicts of all dynamic memory accesses. Both EnfoRSer-D and EnfoRSer-H incur the cost of heavyweight compiler analysis and transformations which affect execution time in a runtime that uses a JIT compiler. In the remainder of the dissertation, we address these pitfalls that stem from software-based conflict detection and expensive compiler analysis. In particular, we

resolve these issues by relying on specialized hardware—hardware transactional memory (HTM) that provides conflict detection and rollback mechanism inherently in the hardware. However, commodity HTM poses several practical challenges. Next, we investigate ways to overcome the challenges in adopting HTM to enforce SBRS.

5.6 Impact and Meaning

The insights and empirical findings discussed with the EnfoRSer-H technique have significant implications. EnfoRSer-H shows directions to capitalize on low-conflict code regions in enforcing bounded region serializability. It advances the state of the art by reducing the overhead in enforcement of SBRS on commodity systems—by exploiting dynamic program characteristics. It shows empirically, the importance of single-threaded overhead while designing dynamic analyses in applications with high-density of accesses. Future research can consider hybridizing the granularity of instrumentation for performance and leveraging static analysis for correctness, for other runtime support mechanisms in concurrent systems.

Chapter 6: End-to-End Dynamically Bounded Region Serializability Using Commodity Hardware Transactional Memory

Chapter 4 and **Chapter 5** provide support for the end-to-end memory model called *dynamically bounded region serializability* (DBRS) introduced in **Chapter 3**. DBRS guarantees atomic execution for regions of code between well-defined program points: loop back edges, method calls and returns, and synchronization operations. This guarantee with respect to the original program has the potential to simplify the job of analyses, systems, and developers. EnfoRSer, described in **Chapter 4** requires complex compiler transformations, and its performance is sensitive to a program's memory access communication patterns (Sections 4.4).

This chapter introduces an approach called *Legato* that provides end-to-end DBRS using commodity *hardware transactional memory* (HTM) [HLR10, HM93, YHLR13]. Our implementation targets Intel's *Transactional Synchronization Extensions* (TSX) [YHLR13], which is widely available on mainstream commercial processors. While enforcing DBRS with commodity HTM seems straightforward, commodity HTM has two relevant limitations. First, the cost of starting and stopping a transaction is relatively high [YHLR13, RB13]. We find that a naïve implementation of DBRS that executes each dynamically

bounded region (DBR) in its own transaction, slows programs on average by 2.7X (Section 6.4.3). Second, commodity HTM is "best effort," meaning that *any* transaction might abort, requiring a non-transactional fallback.

Legato overcomes the first challenge (high per-transaction costs) by merging multiple DBRS into a single transaction. However, longer transactions run into the second challenge, since they are more likely to abort for reasons such as memory access conflicts, private cache misses, and unsupported instructions and events. Legato thus introduces a dynamic, online algorithm that decides, on the fly, how many DBRS to merge into a transaction, based on the history of recently attempted transactions, considering both *transient* and *phased* program behavior. For the (rare) single-DBR transactions that commodity HTM cannot commit, Legato falls back to a global lock to ensure progress.

Our evaluation compares our implementation of Legato with the closest related prior work, which is our work—EnfoRSer as described in **Chapter 4** [SBZ+15], on benchmarked versions of large, real, multithreaded Java applications [BGH+06]. In addition to outperforming EnfoRSer on average, Legato provides two key advantages over EnfoRSer. First, Legato provides stable performance across programs, whereas EnfoRSer's approach is sensitive to an execution's amount of shared-memory communication. For programs with more communicating memory accesses, EnfoRSer slows them considerably, and Legato provides significantly lower overhead. Second, Legato adds significantly less compiler complexity and costs than EnfoRSer—reducing execution costs for just-in-time compilation. Overall, Legato demonstrates the potential benefits—and the inherent limits—of using commodity HTM to enforce strong memory consistency.

6.1 Commodity HTM's Limitations and Challenges

Chapter 2 provides background on HTM and describes the programmability with Intel's HTM (TSX). Here we elaborate its properties that constrains its adoption for serializability enforcement. Intel's commodity HTM has two main constraints that limit its direct adoption for providing region serializability with low overhead: best-effort completion and run-time overhead. Prior work that leverages Intel TSX for various purposes has encountered similar issues [YHLR13, RB13, RUJ14, AOS15, ZLJ16, LGPS15, MKTD14, LXG+14, OS15]. (Prior work uses TSX for purposes that are distinct from ours, e.g., optimizing data race detection. An exception is observationally cooperative multithreading [OS15]; **Chapter 7**.)

Best-effort speculative execution. The TSX specification explicitly provides no guarantees that *any* transaction will commit. In practice, transactions abort for various reasons:

Data conflicts: While a core executes a transaction, if any other core (whether or not it is in a transaction) accesses a cache line accessed by the transaction in a conflicting way, the transaction aborts.

Evictions: In general, an eviction from the L1 or L2 private cache (depending on the TSX implementation) triggers an abort. Thus a transaction can abort simply because the footprint of the transaction exceeds the cache.

Unsupported instructions: The TSX implementation may not support execution of certain instructions, such as CPUID, PAUSE, and INT within a transaction.

Unsupported events: The TSX implementation may not support certain events, such as page faults, context switches, and hardware traps.

Other reasons: A transaction may abort for other, potentially undocumented, reasons.

Overheads of transactional execution. A transactional abort costs about 150 clock cycles, according to Ritson and Barnes [RB13]—which is on top of the costs of re-executing the transaction speculatively or executing fallback code.

Even in the absence of aborts, simply executing in transactions incurs overhead. Prior work finds that each transaction (i.e., each non-nested XBEGIN–XEND pair) incurs fixed "startup" and "tear-down" costs approximately equal to the overhead of three uncontended atomic operations (e.g., test-and-set or compare-and-swap) [RB13, YHLR13]. Prior work also finds that transactional reads have overhead [RB13], meaning that executing a transaction incurs non-fixed costs as well.

6.2 Legato: Enforcing DBRS with HTM

Enforcing the DBRS memory model with HTM appears to be a straightforward proposition. The compiler can enclose each dynamically bounded region (DBR) in a hardware transaction; the atomicity of hardware transactions thus naturally provides region serializability. Unfortunately, this straightforward approach has several challenges. First, the operations to begin and end a hardware transaction are quite expensive—nearly the cost of three atomic operations per transaction [RB13, YHLR13] (Section 6.1). Because DBRs can be quite short, incurring the cost of three atomic operations per region can lead to significant overhead; indeed, on the benchmarks we evaluate, we find that simply placing each DBR in a hardware transaction leads to 175% overhead over an unmodified JVM (see Section 6.4.3). On the other hand, if a transaction is large, a second problem arises: commodity HTM is best-effort, so transactional execution of the DBR is not guaranteed to

complete, e.g., if the memory footprint of a transaction exceeds hardware capacity (Section 6.1). Third, even short regions of code may contain operations such as page faults that cannot execute inside an HTM transaction.

6.2.1 Solution Overview

We propose a novel solution, called *Legato*,[18] to enforce DBRS at reasonable overheads, overcoming the challenges posed by expensive HTM instructions. Our key insight lies in amortizing the cost of starting and ending transactions by merging multiple DBRs into a single hardware transaction. Note that merging DBRs into a larger atomic unit does not violate the memory model: if a group of DBRs executes atomically, the resulting execution is still equivalent to a serialization of DBRs.

Rather than merging together a fixed number of regions into a single transaction (which could run afoul of the other problems outlined above), we propose an adaptive merging strategy that varies the number of regions placed into a hardware transaction based on the history of recent aborts and commits, to adapt to transient and phased program behavior. As transactions successfully commit, Legato increases the number of regions placed into a single transaction, further amortizing HTM overheads. To avoid the problem of large transactions repeatedly aborting due to HTM capacity limitations, Legato responds to an aborted transaction by placing fewer regions into the next transaction. Section 6.2.2 describes Legato's merging algorithm in detail.

Legato's merging algorithm addresses the first two issues with using HTM to enforce DBRS. The third issue, where HTM is unable to complete a single region due to incompatible operations, can be tackled by using a fallback mechanism that executes a DBR using a global lock, which is acceptable since fallbacks are infrequent.

[18] In music, "legato" means to play smoothly, with no breaks between notes.

6.2.2 Merging Regions to Amortize Overhead

If Legato is to merge DBRs together to reduce overhead, the obvious question is how many regions to merge together to execute in a hardware transaction. At a minimum, each region can be executed separately, but as we have discussed, this leads to too much overhead. At the maximum, regions can be merged until "hard" boundaries are hit: operations such as thread fork and monitor wait that inherently interrupt atomicity. While this maximal merging would produce the lowest possible overhead from transactional instructions, it is not possible in practice, due to practical limitations of transactional execution, as well as specific limitations introduced by commodity HTMs, such as Intel's TSX. We identify three issues that exert downward pressure on the number of regions that can be executed as a single transaction:

1. Conflicts between transactions will cause one of the transactions to abort and roll back, wasting any work performed prior to the abort. Larger transactions are likely to waste more work. Predicting statically when region conflicts might happen is virtually impossible.

2. Intel's HTM implementation has capacity limits: while executing a hardware transaction, caches buffer transactional state. If the read and write sets of the transaction exceed the cache size, the transaction aborts *even if there is no region conflict*. In general, larger transactions have larger read and write sets, so are more likely to abort due to capacity limits. While in principle it might be possible to predict when a transaction can exceed the HTM's capacity limits, in practice it is hard to predict the footprint of a transaction *a priori*.

3. Finally, there are some operations that Intel's HTM cannot accommodate within transactions. We call these operations "HTM-unfriendly" operations. Any DBR that contains an HTM-unfriendly instruction not only cannot be executed in a transaction (and hence must be handled separately), but it clearly cannot be merged with other regions to create a larger transaction. Some of these HTM-unfriendly operations can be identified statically, but other HTM-unfriendly operations, such as hard page faults, are difficult to predict.

Note that each of these issues is, essentially, dynamic. It is hard to tell whether merging regions together into a transaction will trigger an abort. Moreover, the implications of each of these issues for region merging is different. Region conflicts are unpredictable and inherently transient, meaning that an abort due to a region conflict is *nondeterministic*. While executing larger transactions may result in more wasted work, it is often the case that simply re-executing the transaction will result in successful completion (though the transaction may inherently be high conflict, in which case merging fewer regions and executing a smaller transaction may be cost effective, or even necessary to make progress).

On the other hand, if transactions abort due to capacity limits or HTM-unfriendly instructions, re-executing exactly the same transaction will likely still result in an abort. If a transaction aborts due to a capacity constraint, it may be necessary to merge fewer regions into the transaction prior to attempting to re-execute the transaction. Furthermore, if a transaction aborts due to an HTM-unfriendly instruction, it might be necessary to execute *just that region* using a non-HTM fallback mechanism (see Section 6.3) to make progress.

While all of these issues push for merging fewer regions into each transaction, minimizing overhead argues for merging as many regions as possible into a single transaction.

Basic Merging Algorithm

Putting it all together, mitigating the overhead of HTM while accounting for its limits suggests the following dynamic approach to merging regions:

- Begin a transaction prior to executing a region. Execute the region transactionally.
- If execution reaches the end of the region without aborting, execute the *following* region as part of the *same* transaction.
- If a transaction aborts and rolls back, determine the reason for its abort. If the abort was transient, such as due to a page fault or a region conflict, retry the transaction. If the abort was due to capacity limits or HTM-unfriendly instructions, retry the transaction but *end the transaction prior to* the region that caused the problem. Begin a new transaction before executing the next region.

While this basic merging algorithm is attractive, and greedily merges as many regions as possible into each transaction, there are several practical limitations. First, note that a transaction has to abort at least once before it can commit, which will waste a lot of work. Second, when a transaction aborts, Intel's HTM implementation *provides no way to know* ***when or where*** *the transaction aborted*. In other words, we cannot know which DBR actually triggered the abort—there is no way to communicate this information back from an aborted transaction—which means that we do not know when to end the re-executed transaction and begin the next one.

Thus we cannot rely on tracking the abort location to determine when to stop merging regions and commit a transaction. Instead, we need a *target* number of regions to merge

```
1  if (--T.regionsExec == 0) {
2    XEND();
3    T.regionsExec = T.controller.onCommit();
4    _eax = -1; // reset HTM error code register
5    abortHandler:
6    if (_eax != -1) { // reached here via abort?
7      T.regionsExec = T.controller.onAbort();
8    }
9    XBEGIN(abortHandler);
10   // if TX aborts: sets _eax and jumps to abortHandler
11 }
```

Figure 6.1: Legato's instrumentation at a DBR boundary. T is the currently executing thread. When a transaction has merged the target number of regions (T.regionsExec == 0) or aborts (control jumps to abortHandler), the instrumentation queries the controller (Figure 6.2) for the number of DBRs to merge in the next transaction.

together. While executing a transaction, if the target number of regions is met, the transaction commits (even if more regions could have been merged into the transaction), and the next region executes in a new transaction.

Figure 6.1 shows the instrumentation that Legato uses at region boundaries to implement this target-based merging strategy. T.regionsExec is a per-thread integer that tracks how many more DBRs to merge together in the current transaction. Once the target number of regions have been merged, the transaction commits, and the instrumentation queries a *controller* to determine the next target (line 3). If a transaction aborts, the instrumentation queries the controller for a new target (line 7).

Note that the controller logic executes only at transaction boundaries (lines 2–9). In contrast, each region boundary executes only an inexpensive decrement and check (line 1).

So how should the controller determine the merge target? In general, if transactions are successfully committing, then it is probably safe to be more aggressive in merging regions together. On the other hand, if transactions are frequently aborting, Legato is probably

being too aggressive in merging and should merge fewer regions into each transaction. Next we describe Legato's controller design.

A Setpoint Controller

Because different programs, as well as phases during a program's execution, have different characteristics—e.g., different region footprints and different likelihoods of region conflicts—we cannot pick a single merge target throughout execution. Instead, our approach should try to infer this target dynamically. Borrowing from the control theory literature, we call the target number of regions the *setpoint*, and the algorithm that selects the setpoint the *setpoint algorithm*.

The key to the setpoint algorithm is that there are two targets: the setpoint, setPoint, which is the "steady state" target for merging regions together, and the current target, currTarget, which is the target for merging regions together *during the current transaction's execution*. (The controller returns currTarget's value to the instrumentation in Figure 6.1.)

We distinguish between these two targets for a simple reason. Legato attempts to execute regions together in a transaction until it hits the setpoint, at which point the transaction commits. But what happens if the transaction aborts before hitting the setpoint? There are two possible inferences that could be drawn from this situation:

1. The abort is a temporary setback, e.g., due to a capacity limit exceeded by this combination of regions. It may be necessary to merge together fewer regions to get past the roadblock, but there is no need to become less aggressive in merging regions overall. In this case, it might be useful to temporarily reduce currTarget, but eventually increase currTarget so that currTarget $=$ setPoint.

2. The abort reflects a new phase of execution for the program—for example, moving to a higher-contention phase of the program—where it might be prudent to be less aggressive in merging regions. The right approach then is to lower setPoint, in order to suffer fewer aborts.

These considerations lead to the design of Legato's control algorithm, which is expressed as a finite state machine with three states, described in Figure 6.2. Each thread uses its own instance of the state machine. In the STEADY state, whenever a transaction commits, the controller views this as evidence that merging can be more aggressive, so it increases setPoint and sets currTarget to setPoint. When a transaction aborts, the controller decreases both setPoint and currTarget. An abort moves the controller into PESSIMISTIC state, which continues to decrease currTarget if more aborts occur. If a transaction *commits* while the controller is in PESSIMISTIC state, the controller does not assume that it is safe to become more aggressive immediately. Instead, the controller keeps the current target, and it moves to OPTIMISTIC state, where it can be aggressive again.

Note that currTarget increases and decreases *geometrically*, while setPoint increases and decreases *arithmetically*. In other words, the current target fluctuates quickly to capture transient effects such as capacity limits, while the setpoint, which represents the steady-state merge target, fluctuates more slowly, to capture slower effects such as changing phases of the program.

Our experiments use a value of 2 for C_{decr}. The idea is that, given the lack of knowledge about the abort location, it is equally probable that a transaction aborted during its first half as its second half. Our experiments use fixed values for S_{inc} and S_{dec} chosen from a sensitivity study (Section 6.4.3).

STEADY

On Commit:

1. setPoint $\leftarrow$ setPoint $+ S_{inc}$
2. currTarget $\leftarrow$ setPoint

On Abort:

1. setPoint $\leftarrow$ setPoint $- S_{dec}$
2. currTarget $\leftarrow \min($currTarget$/C_{decr},$ setPoint$)$
3. $\Rightarrow$ **PESSIMISTIC** state

PESSIMISTIC

On Commit:

1. $\Rightarrow$ **OPTIMISTIC** state

On Abort:

1. currTarget $\leftarrow$ currTarget $/ C_{decr}$

OPTIMISTIC

On Commit:

1. currTarget $\leftarrow \min($currTarget $\times C_{incr},$ setPoint$)$
2. **if** currTarget $=$ setPoint **then** $\Rightarrow$ **STEADY** state

On Abort:

1. currTarget $\leftarrow$ currTarget $/ C_{decr}$
2. $\Rightarrow$ **PESSIMISTIC** state

Figure 6.2: A state machine describing transitions in the setpoint algorithm when onCommit and onAbort are called by the instrumentation. In all cases, the state machine returns currTarget.

6.2.3 Designing a Fallback

Legato requires a fallback mechanism when it encounters a single DBR that cannot be executed transactionally (e.g., if the region contains an HTM-unfriendly instruction). The fallback mechanism must allow the region to execute in a non-speculative manner *while still maintaining atomicity* with respect to other, speculatively executing regions. Because we find that the fallback is needed infrequently, Legato uses a simple, global spin lock to provide atomicity in these situations. If a single-DBR transaction (i.e., currTarget $=$ 1) aborts, Legato's instrumentation acquires a global lock for the duration of the DBR, ensuring atomicity with respect to other, non-speculative regions. To ensure atomicity with respect to other, *speculatively executing* regions, each speculative region must check whether the global lock is held, and abort if so. Existing speculative lock elision approaches use similar logic to elide critical sections [RG01, ZWAT+08, RHH09, YHLR13, DKL+14].

We modify the instrumentation from Figure 6.1 so that, immediately prior to committing a transaction, Legato checks whether the global lock is held. If it is, Legato conservatively aborts the transaction.

6.2.4 Discussion: Extending Legato to Elide Locks

Although individual DBRs are bounded by synchronization operations such as lock acquire and release operations, a multi-DBR transaction may include lock acquires and releases. Thus, it is entirely possible for an executed transaction to include both the start and end of a critical section. Therefore, with modest extensions to its design, Legato can naturally execute the critical section *without* acquiring the lock, in a generalization of lock elision [RG01, ZWAT+08, RHH09, YHLR13, DKL+14]. Legato's approach can even execute several critical sections, including nested critical sections, inside a single transaction.

Although extending the design is minor, changing the implementation's handling of program locks would involve major changes to the locking and threading subsystems, so we leave this exploration for future work.

6.3 Implementation

Although Legato's approach is distinctly different from EnfoRSer's [SBZ+15], we have built Legato in the same JVM instance as our publicly available implementation of EnfoRSer, to minimize any irrelevant empirical differences between the two implementations. We have made our Legato implementation publicly available on the Jikes RVM Research Archive.

We extend Jikes RVM to generate TSX operations by borrowing from publicly available patches by Ritson et al. [RUJ14].

Compilation and instrumentation. EnfoRSer, modifies only the optimizing compiler, due to the complexities of transforming programs to provide the memory model ; it modifies baseline-compiled code to simulate the performance of enforcing DBRS (refer Section 4.3).

Jikes RVM's dynamic compilers automatically insert *yield points*—points in the code where a thread can yield, e.g., for stop-the-world garbage collection or for online profiling [AG05]—into compiled code. Yield points are typically at each method's entry and exit and at loop back edges. Thus they demarcate *statically and dynamically* bounded regions, which EnfoRSer uses to enforce statically bounded region serializability (DBRS). Unlike EnfoRSer, Legato does not require *statically* bounded regions. Nevertheless, Jikes RVM's yield points provide convenient region boundaries, so Legato adopts them. Legato

instruments all yield points and synchronization operations as DBR boundaries. For efficiency and to avoid executing the yield point handler in a transaction, Legato inserts its region boundary instrumentation so that the yield point executes in the instrumentation slow path (see Figure 6.1). This instrumentation performs the yield point logic only after ending a transaction and before starting the next transaction.

Handling HTM-unfriendly events and instructions. For instructions and events that we can identify at compile time as HTM unfriendly (Section 6.2.2), which are often in internal JVM code, the implementation forcibly ends the current transaction, and starts another transaction after the instruction. For unexpected HTM-unfriendly instructions and events, Legato already handles them by falling back to a global lock after a single-DBR transaction aborts.

Our implementation executes application code, including Java library code called by the application, in transactions. Since compiled application code frequently calls into the VM, transactions may include VM code. If the VM code is short, it makes sense to execute it transactionally. However, the VM code may be lengthy or contain HTM-unfriendly instructions. We thus instrument VM code's region boundaries (yield points) like application boundaries—except that VM code does not start a new transaction after committing the current transaction. Since some VM code is "uninterruptible" and executes no yield points, Legato unconditionally ends the current transaction at a few points we have identified that commonly abort (e.g., the object allocation "slow path"). Whenever a thread ends a transaction in VM code, it starts a new transaction upon re-entering application code.

6.4 Evaluation

This section evaluates the performance and run-time characteristics of Legato, compared with alternative approaches.

6.4.1 Setup and Methodology

EnfoRSer implementation. EnfoRSer provides multiple compiler transformations for ensuring atomicity; our experiments use EnfoRSer's *speculation transformation* because it performs best (refer Section 4.4) [SBZ+15].

EnfoRSer can use the results of whole-program static race detection analysis (refer Section 4.4 and Section 5.4) [SBZ+15, SCBK15]. However, to avoid relying on the "closed-world hypothesis" (**Chapter 7**) and to make EnfoRSer more directly comparable with Legato (which does not use whole-program static analysis), our experiments do *not* use any whole-program static analysis. Legato does not rely on any pre-processing of programs with whole-program static analysis, simplifying its deployability compared to the version of EnfoRSer that uses the results of static analysis.

Environment. The experiments run on a single-socket Intel Xeon E5-2683 system with 14 cores and one hardware thread per core (we disable hyperthreading), running Linux 3.10.0. Although Intel has disabled TSX in this processor because of a known vulnerability, the vulnerability does not affect non-malicious code, so we explicitly enable TSX.

	Threads		Dynamic events		
	Total	Live	Accesses	DBRs	Acc. / DBR
eclipse6	18	12	1.6×10^{10}	5.0×10^{9}	3.1
hsqldb6	402	102	6.8×10^{8}	4.4×10^{8}	1.5
lusearch6	65	65	3.1×10^{9}	9.7×10^{8}	3.2
xalan6	9	9	1.3×10^{10}	5.2×10^{9}	2.5
avrora9	27	27	7.9×10^{9}	1.4×10^{9}	5.6
jython9	3	3	6.6×10^{9}	4.7×10^{9}	1.4
luindex9	2	2	3.9×10^{8}	1.5×10^{8}	2.6
lusearch9	c	c	3.0×10^{9}	8.9×10^{8}	3.4
pmd9	5	5	6.4×10^{8}	3.7×10^{8}	1.7
sunflow9	$2\times c$	c	2.3×10^{10}	3.4×10^{9}	6.9
xalan9	c	c	1.2×10^{10}	4.7×10^{9}	2.6
pjbb2000	37	9	2.6×10^{9}	1.2×10^{9}	1.7
pjbb2005	9	9	9.1×10^{9}	3.6×10^{9}	2.5

Table 6.1: Dynamic execution characteristics. Three programs spawn threads proportional to the number of cores c, which is 14 in our experiments. The last three columns report memory accesses executed, statically bounded regions executed, and their ratio.

Benchmarks. In our experiments, modified Jikes RVM executes (1) the large workload size of the multithreaded DaCapo benchmarks [BGH+06] versions 2006-10-MR2 and 9.12-bach (2009), distinguished by suffixes 6 and 9, using only programs that Jikes RVM can run, and (2) fixed-workload versions of SPECjbb2000 and SPECjbb2005.[19]

[19] `http://www.spec.org/jbb200{0,5}`, `http://users.cecs.anu.edu.au/~steveb/research/research-infrastructure/pjbb2005`

	Staccato		Legato							
				DBRs /	Acc. /	Abort rate				
	Trans.	Abort rate	Trans.	trans.	trans.	Total	(conf	cap	fallb	other)
eclipse6	5.0×10^9	<0.1%	4.4×10^7	110	360	14.4%	(2.1%	3.3%	<0.1%	9.0%)
hsqldb6	4.2×10^8	<0.1%	8.8×10^6	48	77	13.8%	(1.7%	2.3%	<0.1%	9.8%)
lusearch6	8.9×10^8	0.8%	3.5×10^7	25	88	31.0%	(24.9%	0.6%	0.2%	5.4%)
xalan6	5.1×10^9	0.1%	2.3×10^8	22	57	10.5%	(7.2%	0.6%	<0.1%	2.7%)
avrora9	1.4×10^9	3.2%	1.1×10^8	13	71	41.7%	(13.6%	0.4%	3.7%	24.0%)
jython9	4.5×10^9	<0.1%	3.5×10^7	130	190	8.0%	(0.8%	5.1%	<0.1%	2.1%)
luindex9	1.5×10^8	<0.1%	1.5×10^6	100	260	16.7%	(0.9%	4.9%	<0.1%	10.9%)
lusearch9	8.2×10^8	0.2%	2.8×10^7	29	110	22.7%	(15.9%	0.6%	<0.1%	6.1%)
pmd9	3.5×10^8	1.2%	1.5×10^7	23	43	41.2%	(35.5%	1.5%	0.2%	4.0%)
sunflow9	3.3×10^9	0.3%	5.3×10^7	62	440	20.8%	(17.3%	2.5%	<0.1%	1.0%)
xalan9	4.5×10^9	0.2%	2.2×10^8	20	55	14.8%	(8.5%	0.7%	<0.1%	5.7%)
pjbb2000	1.2×10^9	<0.1%	3.9×10^7	31	53	7.2%	(3.4%	0.53%	<0.1%	3.2%)
pjbb2005	3.6×10^9	0.4%	5.0×10^8	7.2	18	28.1%	(14.8%	0.1%	<0.1%	13.2%)

Table 6.2: Transaction commits and aborts for Staccato and Legato, and average DBRs and memory accesses per transaction for Legato.

Execution methodology. For each configuration of EnfoRSer and Legato, we build a high-performance Jikes RVM executable, which adaptively optimizes the application as it runs and uses the default high-performance garbage collector, which adjusts the heap size automatically.

Each performance result is the mean of 15 runs, with 95% confidence intervals shown. Each reported statistic is the mean of 8 statistics-gathering runs.

6.4.2 Run-Time Characteristics

The *Threads* columns in Table 6.1 report the total number of threads spawned and maximum number of live threads for each evaluated program. The *Dynamic events* columns report the number of executed memory accesses, the number of executed dynamically bounded regions (i.e., the number of region boundaries executed), and the average number of memory accesses executed per DBR. Each program executes between hundreds of millions and tens of billions of memory accesses. The average DBR size for each program varies from 1.4 to 6.9 executed memory accesses.

Table 6.2 reports committed and aborted transactions for the default configuration of Legato that uses the setpoint-based merging algorithm (right half), compared with a configuration of Legato that does no merging, which we call *Staccato*[20] (left half). The two *Trans.* columns show the number of committed transactions for Staccato and Legato, respectively. The *DBRs / trans.* column shows the number of DBRs executed per committed transaction for Legato. Note that Staccato executes one DBR per transaction.[21] As the

[20] In music, "staccato" means to play each note disconnected from the others.

[21] Although one might expect the number of transactions executed by Staccato to be exactly equal to the number of regions per benchmark from Table 6.1, they differ because (i) the measurements are taken from different runs, and (ii) in Staccato, not every region executes in a transaction, due to the fallback mechanism.

table shows, on average Legato reduces the executed transactions by 1–2 orders of magnitude compared with Staccato. The *Acc. / trans.* column shows that a typical transaction in Legato executes dozens or hundreds of memory accesses.

While the reduction in committed transactions represents the *benefit* provided by Legato's merging of DBRs into transactions, Legato's *Abort rate* columns represent the *cost* of merging. The *Total* column shows the total number of aborts, as a percentage of total committed transactions (the *Trans.* column). Most programs have an abort rate between 10% and 30%, which is substantial. In contrast, the abort rate for Staccato is significantly lower: typically less than 1% and at most 3.2%. The values in parentheses show the breakdown of aborts into four categories (again as a percentage of committed transactions): conflict and capacity aborts; explicit aborts when the global fallback lock is held (Section 6.2.3); and other reasons (Section 6.1). For most programs, conflicts are the primary cause of aborts; conflicts are difficult to predict since they involve cross-thread interactions. In general, it is worthwhile for Legato to "risk" conflicts in order to merge transactions and avoid per-transaction costs. The second-most common cause of aborts is "other"; although RTM's abort error code provides no information about these aborts, by using the Linux perf tool we find that most of these aborts are due to system calls inside the VM, existing exceptions in the program, and system interrupts.

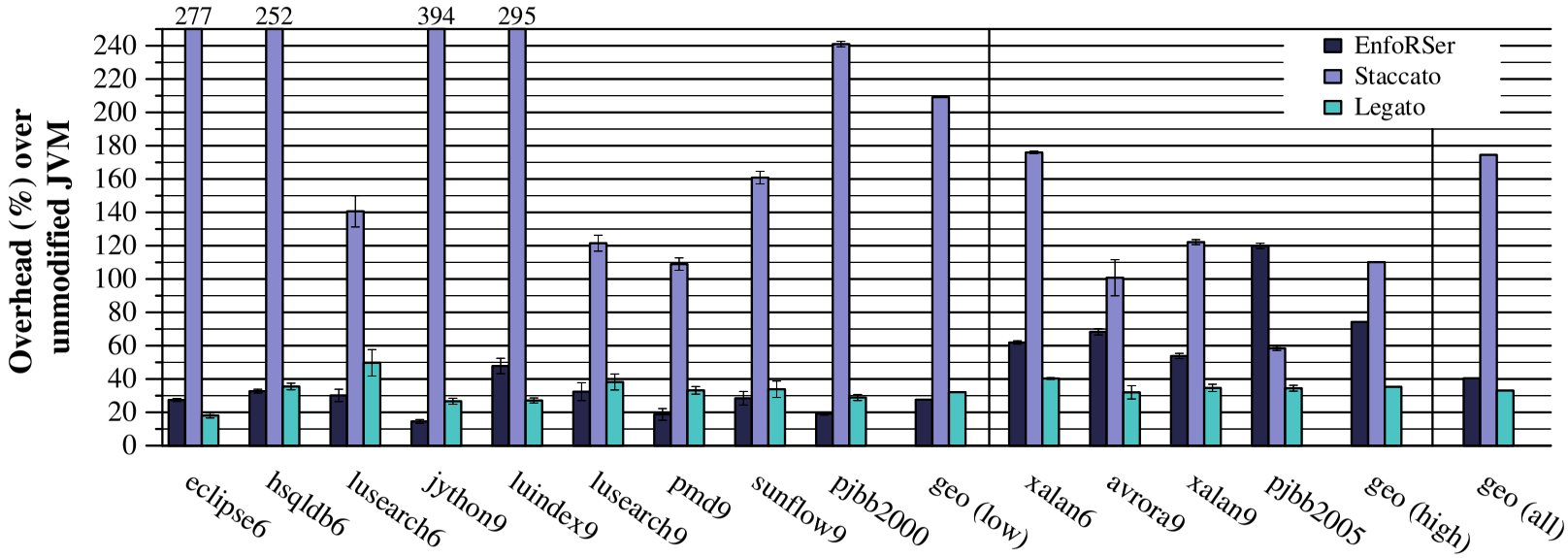


Figure 6.3: Run-time overhead of providing DBRS with three approaches: software-based EnfoRSer; Staccato, which executes each DBR in a transaction; and the default Legato configuration that merges multiple DBRs into transactions. The programs are separated into low- and high-communication behavior, with component and overall geomeans reported.

6.4.3 Performance

Figure 6.3 compares the performance of three approaches that provide DBRS: EnfoRSer [SBZ+15]; Staccato, which executes each DBR in its own transaction; and the default Legato configuration. Each result is the run-time overhead over baseline execution (unmodified Jikes RVM). Since EnfoRSer adds dramatically different overhead depending on the amount of cross-thread communication between memory accesses (Section 4.4), the figure divides the programs into whether they have low or high rates of communicating memory accesses, leading to low or high EnfoRSer overhead, respectively. We categorize programs as "high communication" if $\geq$0.1% of all memory accesses trigger biased reader–writer locks' "explicit communication" protocol [SBZ+15, BKC+13].

For the four high-communication programs, EnfoRSer adds nearly 60% overhead or more; for pjbb2005, EnfoRSer adds over 120% overhead. Unsurprisingly, pjbb2005 has the highest percentage of accesses involved in cross-thread communication, 0.5% [BKC+13].

In contrast, the run-time overhead of Staccato is not correlated with cross-thread communication, but is instead dominated by the costs of starting and ending transactions. On average, Staccato adds 175% run-time overhead due to the high cost of frequent transaction boundaries. This result shows that applying commodity HTM naïvely to provide DBRS is a nonstarter, motivating Legato's contributions.

Legato improves over both Staccato and EnfoRSer by amortizing per-transaction costs and by avoiding EnfoRSer's highly variable costs tied to communicating memory accesses. By merging transactions, Legato reduces average run-time overhead to 33%, compared with 175% without merging, despite incurring a significantly higher abort rate than Staccato (Table 6.2). Legato's overhead is relatively stable across programs, and is at most 50% for

any program. On average, Legato adds 32% and 35%, respectively for the low- and high-communication programs. Notably, Legato's average overhead for high-communication programs (35%) is significantly lower than EnfoRSer's average overhead for these programs (74%). Not only does Legato significantly outperform EnfoRSer on average (33% versus 40%, respectively), but Legato's overhead is more stable and less sensitive to shared-memory interactions.

Legato's performance breakdown. We used partial configurations to find more insights into the 33% run-time overhead of Legato. Instrumentation at region boundaries adds an overhead of 12%. The remaining overhead of 21% comes from beginning and ending transactions, aborting and running inside transactions. We have found it diffcult to concretely separate out these costs.

Sensitivity study. As presented in Section 6.2.2, Legato maintains a per-thread setpoint, setPoint, that it adjusts in response to transaction commits and aborts. The setpoint algorithm's state machine (Figure 6.2) uses S_{inc} and S_{dec} to adjust setPoint. By default, Legato uses the parameters $S_{inc} = 1$ and $S_{dec} = 10$; our rationale for these choices is that the marginal cost of an aborted transaction is significantly more than the cost of executing shorter transactions.

Here we evaluate sensitivity to the parameters S_{inc} and S_{dec}. Figure 6.4 shows the run-time overhead of Legato with each combination of 1 and 10 for these values. The first configuration, *Legato S_inc = 1, S_dec = 10*, corresponds to the default Legato configuration used in the rest of the evaluation. The second configuration, *Legato S_inc = 10, S_dec = 1*, shows that performance is sensitive to large changes in these parameters; its high overhead validates the intuition behind keeping the S_{inc}/S_{dec} ratio small. The last two

configurations both set $S_{inc} = S_{dec}$; their overheads fall between the overheads of the first two configurations.

The *magnitudes* of S_{inc} and S_{dec} represent how quickly the algorithm adjusts to phased behavior. We find that for other values of S_{inc} and S_{dec} within 2X of the default values (results not shown), the magnitudes do not significantly affect average performance. Instead, the results suggest that the S_{inc}/S_{dec} *ratio* is most important, and a ratio of about 0.1 provides the best performance on average.

Dynamic vs. fixed setpoints. Legato uses a dynamic setpoint algorithm that adjusts the setpoint based on recent execution behavior. An alternative approach is to use a fixed setpoint throughout the entire execution, which cannot handle differences between programs or a program's phases. Section 6.5 evaluates this alternative, finding that the best fixed setpoint differs from program to program. While the best fixed setpoint for each application provides similar performance to Legato's dynamic setpoint algorithm, Legato does not require per-application tuning.

Compilation time. For both EnfoRSer and Legato, the dynamic, just-in-time compilers in Jikes RVM insert instrumentation into the compiled code. EnfoRSer performs complex compiler analyses and transformations that generate significantly more code than the original program; unlike Legato, EnfoRSer analyzes, transforms, and instruments memory accesses, branches, arithmetic instructions, and other instructions. In contrast, Legato limits its instrumentation to region boundaries. In a just-in-time compilation setting, the additional compilation time translates into extra execution time for several reasons: (1) the extra compilation time itself (although Jikes RVM uses a separate optimizing compiler thread); (2) the effect on downstream optimizations and transformations that must handle

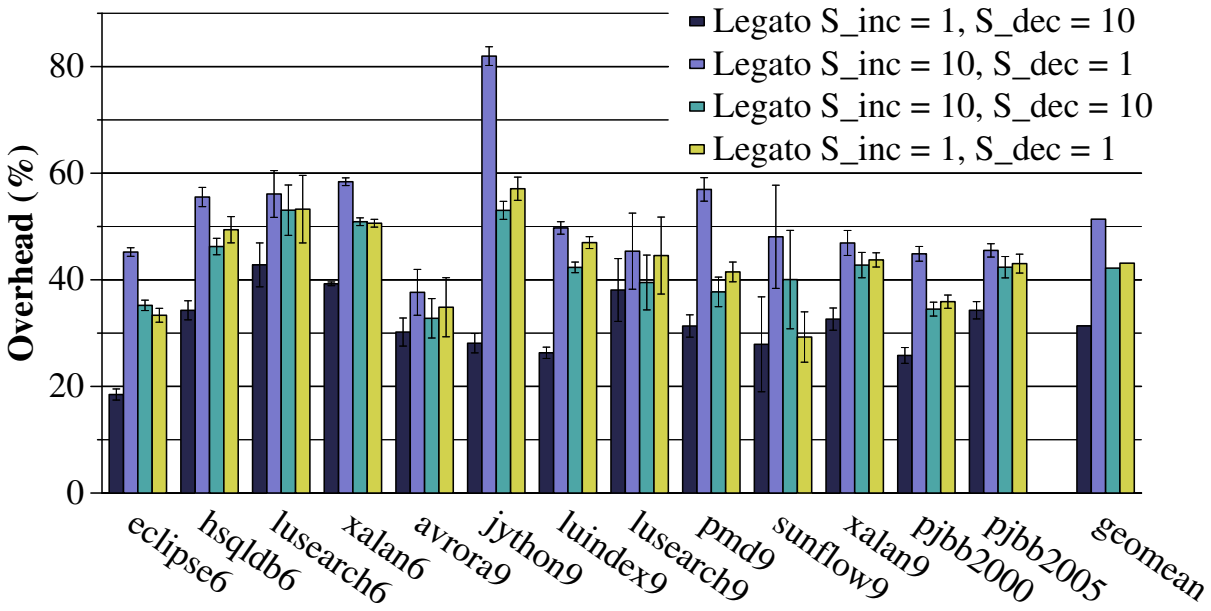


Figure 6.4: Sensitivity of run-time performance to the parameters to Legato's dynamic setpoint algorithm.

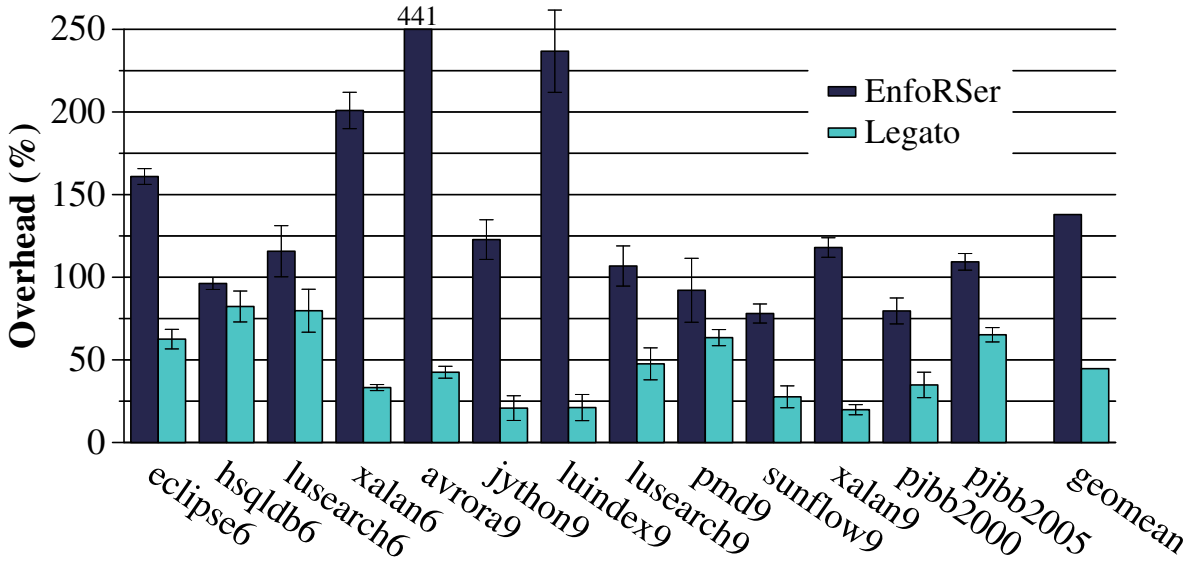


Figure 6.5: Just-in-time compilation overhead of providing DBRS with EnfoRSer versus Legato.

more code; and (3) the opportunity cost of not being able to optimize code as quickly, and thus executing less-optimized code overall.

Figure 6.5 shows the compilation time added by EnfoRSer and Legato for the same executions used for Figure 6.3. We emphasize that since compilation occurs at run time, these costs are already reflected in overall execution time; in fact, Legato's compilation overhead advantage is somewhat muted by the fact that Jikes RVM's adaptive optimization system [AFG+00] naturally performs less compilation as compilation becomes more expensive (i.e., Legato optimizes more methods than EnfoRSer). The figure shows that on average Legato adds one-third as much compilation overhead over the baseline (unmodified Jikes RVM) as EnfoRSer: 138% versus 45%. These results show that the best-performing existing approach that targets commodity systems, relies on complex, costly compiler analyses and transformations. In contrast, Legato uses only relatively simple instrumentation at region boundaries (cf. Figure 6.1).

Space overhead. Section 6.6 reports space overhead of EnfoRSer and Legato over an unmodified JVM. EnfoRSer and Legato incur average space overhead of 29% and 2%, respectively, mainly because EnfoRSer adds per-object metadata while Legato does not.

6.5 Fixed Versus Dynamic Setpoints

Figure 6.7 evaluates the performance of Legato's dynamic setpoint algorithm, compared with an alternative strategy of using a fixed setpoint for the entire execution. *Legato default (dynamic setpoint)* is the default Legato configuration evalauted in Section 6.4. The figure shows a horizontal line indicating the overhead of default Legato. *Legato w/fixed setpoint* uses a fixed setpoint (i.e., a constant value of setPoint for the entire execution). The figure plots the run-time overhead of *Legato w/fixed setpoint* for various fixed setpoint

values, from 2 to 4096. Each point is the mean of 10 trials, with the interval showing 95% confidence intervals.

Most programs have a fixed setpoint at which they incur the lowest run-time overhead. However, the best-performing setpoint is not constant across benchmarks. Legato's default algorithm closely replicates the peformance of the best fixed setpoint on certain instances and fails on other occasions. The absence of a best-perfoming fixed setpoint across benchmarks proves that rigorous per-application profiling is required to use *Legato w/fixed setpoint* effectively. In contrast, the default Legato algorithm can avoid per-application profiling and yet provide performance close to the best-known fixed setpoint. On average, across benchmarks, the default algorithm improves performance compared to any single fixed setpoint (Figure 6.7(n)).

6.6 Space Overhead

This section provides a comparison of space overhead added by EnfoRSer and Legato. We define the memory usage of an execution as the maximum heap memory in use after any full-heap garbage collection (GC). The experiments use the default, high-performance, generational GC in Jikes RVM and allow GC to adjust the heap size automatically (Section 6.4.3).

Figure 6.6 shows that the memory usage overhead incurred (over an unmodified JVM) by EnfoRSer (29%) is significantly higher than by Legato (2%). We believe EnfoRSer's space overhead is higher mainly because EnfoRSer adds a word to each object's header to act as a lightweight biased reader–writer lock (Section 4.1.3) [SBZ+15, BKC+13]. In contrast, Legato relies on the intrinsic conflict detection capabilities of commodity HTM. Another potential factor is that EnfoRSer's compiler passes use heavyweight analysis and

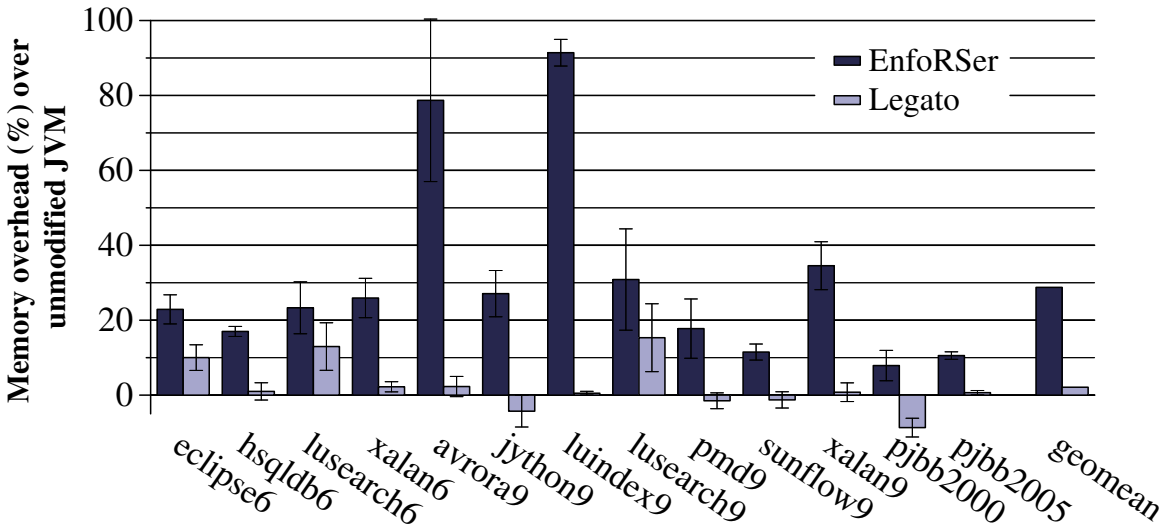


Figure 6.6: Space overhead of providing SBRS with EnfoRSer versus Legato.

transformations at run time, generating many objects and triggering more frequent GCs, although we expect most of these objects to be short-lived and not survive full-heap GCs. Legato provides repeatedly lower space overhead than the *baseline* for pjbb2000, presumably because Legato's allocation behavior leads to triggering GC at points when the working set size is relatively smaller than the points when GC occurs in the baseline.

6.7 Summary

Among prior approaches providing bounded RS, only one, our work discussed in **Chapter 4** and **Chapter 5** [SBZ+15, SCBK15] does not rely on custom hardware. In this chapter, our focus was also on supporting commodity systems—which now provide best-effort HTM. While EnfoRSer has relatively low overhead (36% and 40% on the architecture used by Legato), it suffers from several serious drawbacks. First, it uses complex compiler analyses and transformations that limit adoption and affect execution time in a just-in-time

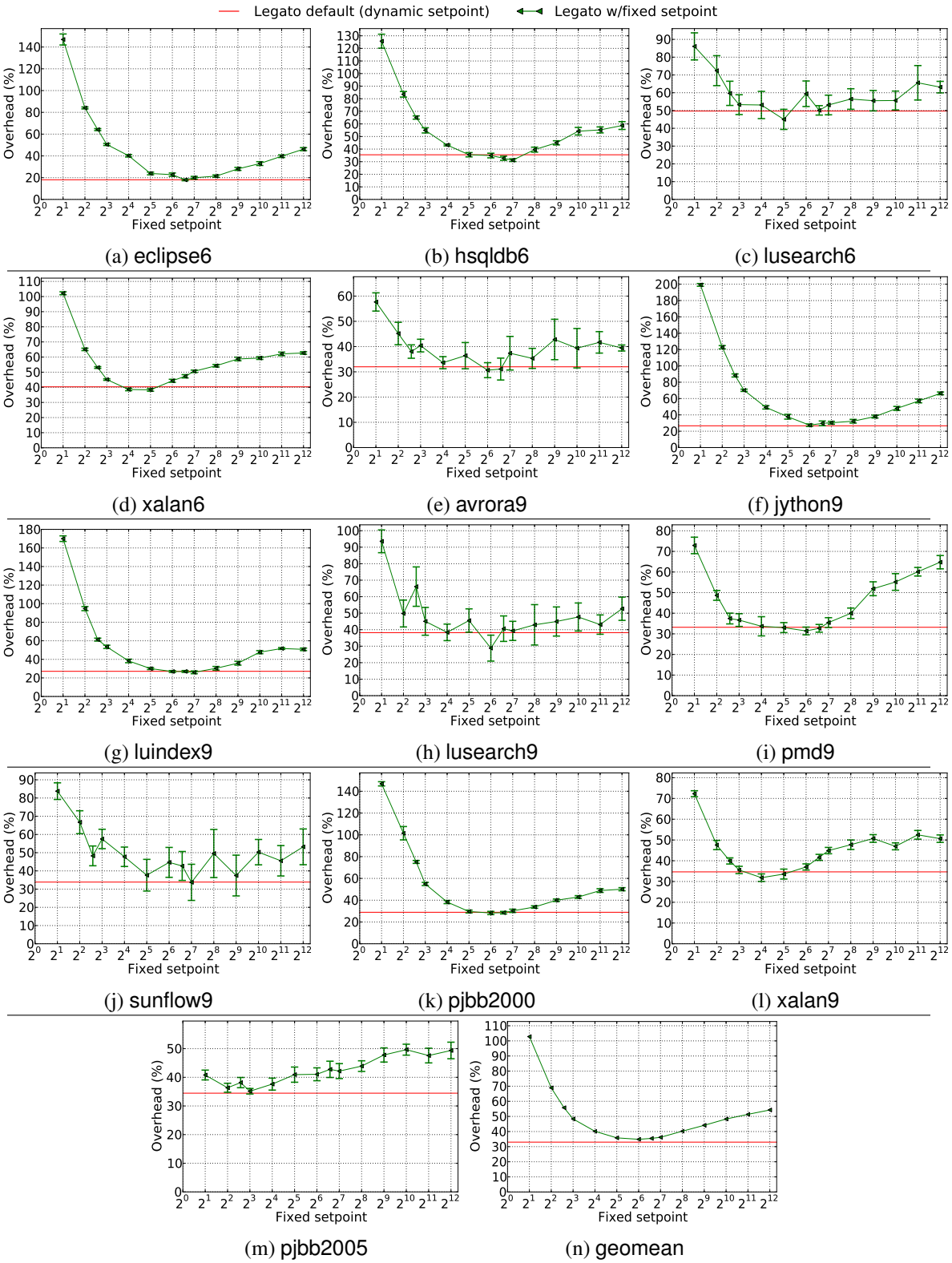


Figure 6.7: Performance of Legato's dynamic setpoint algorithm compared with Legato using various fixed setpoints, for each benchmark and across benchmarks. X-axis plots fixed set points.

compilation environment. Second, EnfoRSer achieves low overhead by employing a form of lightweight, *biased* reader–writer lock [BKC+13] that adds low overhead for programs with relatively few inter-thread dependences. For programs with even modest levels of communication, the biased locks slow programs substantially [SBZ+15, BKC+13]. In this chapter, we address these two keys issues by leveraging commodity HTM to enable cheaper conflict detection. The technique addresses the *high per-transaction cost* via merging multiple regions into a single transaction. It addresses *abort cost* by using a control-theoretic technique that dynamically adapts to *transient* and *phased* program behavior on the fly. It overcomes the limitations and complexity of existing work. Unlike prior approaches, Legato maintains consistent overhead even in the face of applications with many communicating threads, making it more suitable for applications with diverse communication patterns, and giving it more predictable performance overall.

6.8 Impact and Meaning

Legato is the first technique to use HTM efficiently and at scale—to resolve a deep-rooted memory consistency problem. Prior to this work, HTM has primarily been used to provide atomicity using *speculative lock elision* [RG01, ZWAT+08, YHLR13] or to provide scalable synchronization solutions in multiple areas [RB13, RUJ14, AOS15, ZLJ16, LGPS15, MKTD14, LXG+14], except one work that provides unbounded RS [OS15] (**Chapter 7**). Legato significantly changes the state of the art in bounded RS enforcement, both in the treatment of design and in the quality of performance. Legato simplifies *data-race-freedom* enforcement on commodity hardware—involving only minimal changes to

the runtime. We believe that this novel mechanism to utilize existing hardware for developing memory consistency models, will foster coordinated effort by researchers working on langauge memory models as well as new hardware.

Chapter 7: Related Work

Chapter 2 discusses the prior work and ideas directly related to memory models, reducing the overhead of dynamic analyses, and programming models. This chapter discusses work comparable to and related to our work that were not discussed in **Chapter 2**.

Checking for region conflicts. This dissertation proposes three distinct techniques to *enforce* atomicity of bounded regions based on the idea that harmful effects of programmer induced errors can be eliminated using compiler and runtime techniques. In contrast to this ideology, prior work treats data races as *exceptions* that *terminate* the program and hence *checks* for region conflict freedom (RCF) in order to detect sequential consistency (SC) violations and region serializability (RS) violations due to data races [EDLC+12, MSM+10, SMN+11, LCS+10, GG91, BZBL15]. These regions can be delimited only by synchronization operations [LCS+10, EDLC+12] or statically bounded to simplify the hardware [MSM+10, SMN+11] complexity and design. These approaches rely on custom hardware [MSM+10, SMN+11, LCS+10, GG91] or add high overhead [EDLC+12] while monitoring state in large unbounded regions. However, checking RCF can generate errors unexpectedly, even for executions for which serializability is not violated. In contrast, this dissertation *enforces* RCF and thus RS in commodity systems.

Another serializability-based model. O'Neill and Stone use TSX to provide speculation-based atomicity for *observationally cooperative multithreading* (OCM) [OS15]. In OCM, all program code is in atomic regions, but the regions are not dynamically bounded. The OCM implementation uses TSX's HLE implementation, which automatically falls back to a global lock for transactions that abort; the main performance challenge is serialized execution resulting from regions that abort when executed as TSX transactions. In contrast, we find in **Chapter 6**, Legato's regions generally fit in a TSX transaction; the main challenge is merging regions into transactions to balance competing costs.

Dynamic atomic regions. Prior work proposes customized hardware to execute code in atomic regions, enabling aggressive speculative optimizations that can be rolled back if necessary [DGB+03, MK12]. Mars and Kumar address one challenge in such settings: data conflicts can cause atomic optimization regions to roll back excessively [MK12]. They introduce a framework to mitigate the effect of shared-memory conflicts that lead to misspeculations or "squashes." Their technique, called *BlockChop*, uses a composition of retrials, delay, translation to finer-grained regions, and fallback to conflict-free interpretation mode to significantly reduce the number of misspeculations, exploiting the right tradeoff between the costs of these mechanisms. While BlockChop shows the cost of conflicting accesses to be a significant contributor to overheads, Legato's approach to DBRS enforcement, shows that in current commercial hardware, the cost of start and end of transactions is prohibitive. Essentially, because of their different objectives and constraints, BlockChop favors reducing region sizes to reduce conflicts while Legato favors increasing region sizes to reduce overhead.

Using whole-program static analysis. Whole-program static analysis can identify potential data races between static program memory accesses [NAW06, NA07]. However, sound (no missed races) static analysis suffers from two main problems. First, its precision scales poorly with program size and complexity. Second, whole-program static analysis relies on the so-called "closed-world hypothesis"—that all of the code is available at compile time—which fails to hold for dynamically loaded languages such as Java. However, prior work has used static analysis to identify accesses where instrumentation would be "redundant" due to preceding instrumentation for the same object [FF13, BKC+13]. In spite of the aforementioned limitations, EnfoRSer, presented in **Chapter 4** uses the results of sound static analysis to avoid instrumenting definitely data-race-free accesses, reducing average overhead from 36% to 27% [SBZ+15]. In follow-up work, presented in **Chapter 5**, we extend EnfoRSer to provide atomicity of low-contention regions with coarse-grained pessimistic locks, using static pairs of potentially racy accesses to identify regions that should use the same lock [SCBK15]. In contrast, Legato, presented in **Chapter 6** does not rely on whole-program static analysis or the closed-world hypothesis.

Leveraging Static locks. To provide deterministic replay for racy programs, *Chimera* records an execution by recording not only its synchronization operations but also the ordering between accesses involved in data races [LCFN12]. Chimera uses the same static lock for each pair of accesses that potentially race with each other, according to conservative static analysis, similar to EnfoRSer-S's selection of static locks based on the $RACE$ relation as discussed in **Chapter 5**. Using static locks slows programs by more than an order of magnitude. To reduce the overhead of acquiring a lock at each potentially racy access, Chimera coarsens lock granularity in cases where profiling predicts that two regions

using the same lock will contend infrequently. While lock coarsening is similar in spirit to EnfoRSer-S and EnfoRSer-H's use of the $SRSL$ relation to acquire a single lock for each region, Chimera's coarsening can potentially lead to high contention because its regions are in general neither statically nor dynamically bounded. Chimera relies on symbolic execution in order to associate address ranges with coarsened locks to minimize false conflicts. This approach is mostly beneficial to avoid repeated false conflicts inside loops. EnfoRSer-H avoids such optimizations since loop back edges are natural boundaries of dynamically bounded regions.

Aside from the fact that Chimera focuses on recording cross-thread dependencies, whereas our work presented in **Chapter 5** enforces atomicity, the two approaches differ in two important ways. First, Chimera uses only static locks associated with static code locations (sites or regions); it makes no use of what we call "dynamic locks": locks associated with shared memory (objects). Although Chimera can achieve reasonable performance using static locks, our results show that EnfoRSer-S experiences very high contention with static locks alone (Section 5.4.2), suggesting that Chimera's evaluated programs have significantly less contention than our evaluated programs. Second, since Chimera uses only static locks, it does not investigate hybridizing static and dynamic locks, which is the main contribution of EnfoRSer-D.

Hybridizing locking mechanisms. **Chapter 5** of this dissertation targets a major source of overhead of enforcing DBRS: the *overhead* incurred by EnfoRSer-D to perform a lock acquire operation at every potentially racy memory access. An orthogonal cost is the performance penalty incurred by biased reader–writer locks for conflicting lock acquires that

require *communication* between threads [BKC+13, SBZ+15]. In other work, we have targeted this separate problem; that work chooses between *biased* and *unbiased* reader–writer locks based on run-time profile information [CZB14]. EnfoRSer-S and EnfoRSer-H could make use of that complementary approach in order to use a combination of biased and unbiased reader–writer locks for both dynamic and static locks.

Other prior work has combined synchronization mechanisms adaptively. For example, Usui et al. combine lock-based mutual exclusion and software transactional memory (STM) [UBES09]. Abadi et al. present an STM that adaptively changes how it detects conflicts for non-transactional accesses, depending on whether transactions access the same objects as non-transactional code [AHM09].

Miscellaneous uses of commodity HTM. Recent work applies HTM to concurrent and parallel garbage collection (GC) to overcome the complex synchronization problems. For example, the runtime needs to ensure mutual atomicity of an object access by the *mutator* and the movement of the same object by a moving collector [AOS15, RUJ14]. Chihuahua utilizes HTM's strong atomicity property to preclude complex synchronization and enable transactional movement of objects. Ritson et al. use HTM in order to synchronize efficiently between GC threads in parallel collection and between GC and mutator threads in concurrent copying collection, attaining speed-up in the latter [RUJ14]. *TxIntro* uses TSX to detect malicious modification to VM data structures without stopping the VM— leveraging TSX's strong atomicity [LXG+14]. There is work on race detection using custom hardware and HTM [GSC+09]. Recent work uses TSX to detect data races and for deterministic replay. *TxRace* uses commodity HTM as a low-overhead filter to detect potential data races; it detects data races precisely by re-executing aborted transactional code with

instrumentation-based race detection [ZLJ16]. Matar et al. use TSX to ensure atomicity of a happens-before data race detector's [FF09] metadata checks and updates [MKTD14]. *TSXProf* records the commit order of TSX transactions and replays the order with debuggable software transactions [LGPS15]. In contrast, Legato, presented in this dissertation, enforces an end-to-end memory model, DBRS, by executing all code in transactions that are at least as large as DBRs.

Software transactional memory. **Chapter 2** discusses background on transactional memory (TM), specifically hardware transactional memory (HTM). Afters years of research in TM, HTM is part of commercial hardware [WGW+12, int12]. However, language level support for TM—*software transactional memory* (STM) is yet to appear in mainstream languages like C++ and Java. Typically, STMs, as the name suggests, are purely software-based [SMAT+07, DSS10b, ZHCB15, WMvP+09]. In contrast, HTM transactions might fail to complete and hence fallback on STM for completion [YHLR13, CHM16]. STMs usually use software-based reader-writer locks for conflict detection. An STM transaction also maintains read-write sets to track conflicts precisely. Therefore, a vast majority of designs suffer either from high single-threaded overhead or compromise on scalability while managing conflicts dynamically [SMAT+07, DSS10b, SATH+06, ZHCB15]. Section 4.4.3 provides evidence that even a stripped down version of STM can potentially incur high overheads for our use case—bounded region serializability. A significant literature exists that studies problems and solutions with respect to the interference of transactional and non-transactional code [SMAT+07, DS09, HLR10]. Our work, executes all the code in transactions, barring region boundaries; hence bypassing the issue. Several STMs thoroughly study progress guarantees with respect to its design [SMAT+07, DSS10b,

ZHCB15]. Our techniques in EnfoRSer and EnfoRSer-H could potentially show a lack of progress (e.g. livelock) which could be solved using established STM techniques that guarantee progress [ZHCB15]. However, for bounded regions, we didn't encounter the issue in practice.

Chapter 8: Future Work and Potential Improvements

In the words of Paul Graham [Gra04]: *"The difference between design and research seems to be a question of the good versus the new. Design doesn't have to be new, but it has to be good. Research doesn't have to be good, but it has to be new. I think these two paths converge at the top: the best design surpasses its predecessors by using new ideas, and the best research solves problems that are not only new, but worth solving. So ultimately design and research are aiming for the same destination, just approaching it from different directions."* I share the same opinion that research needs to be new than be perfect. However, good research should address a critical problem that has broad impacts. This dissertation proposes a new memory model (DBRS) for bounded region serializability to address a challenging and critical issue—enforcement of *data-race-freedom*. This work presents three novel and distinct ways to enforce DBRS. There can be several directions in which—DBRS as a memory model or the techniques used to practically enforce it—can be altered or augmented. This chapter describes ideas in that direction.

8.1 Strong Semantics for Transactional and Non-transactional Programs

Until now we have focused on providing atomicity of code regions that programmers expect to be atomic but the programmer did not explicitly use synchronization for the regions or used synchronization erroneously. In other words, several dynamically bounded regions that are expected to execute atomically are contained in synchronization-free regions. A critical section's atomicity is enforced through locks or *transactional memory* (TM) [HM93]. Transactional programs suffer from weak semantic behaviors especially in the interaction of transactional and non-transactional code. Guaranteeing strong semantics is expensive and the state of the art notion of strong semantics for transactional programs still suffers from weak guarantees on relaxed memory models. Future work could investigate ways to provide semantics stronger than the state of the art for transactional programs. Critical sections in real-world code are often over-synchronized [ZLH+15] and researchers have used *speculative lock-elision* (SLE) techniques related to TM to enhance the performance of suboptimally synchronized programs [RG01, ZWAT+08, RHH09]. Our notion of strong semantics should, therefore, extend to both locks and transactions in a single system. An interesting direction is to provide stronger semantics than the state of the art for transactional or hybrid programs (e.g. SLE systems [RG01, ZWAT+08, RHH09]). Adding transactions to legacy code is challenging. Therefore, supporting user-defined transactions in a system will enable incremental deployment of the programs. We also conjecture that for a critical-section, both traditional locks or TM might be inefficient, and a technique based on two-phase locking akin to our means to provide DBRS using EnfoRSer might be more efficient. Therefore, future work should support user-defined transactions for better programmability with automatically chosen atomicity enforcement if the user does not

specify the choice. The system can be designed to support multiple forms of atomicity enforcement while still enforcing strong semantics at reasonable overheads.

8.2 Using Hardware Transactional Memory more Efficiently for DBRS

In **Chapter 6** the limitations of adopting commodity HTM is addressed via Legato's merging technique. However, EnfoRSer'X controller is conservative with respect to the actions taken in case of an abort due to the absence of the abort location in the information exposed by the hardware. An alternate approach could measure the duration of time elapsed, instead of regions executed, as the control variable in the merging algorithm. The rationale behind developing a merging algorithm with the notion of time is to predict the length of potentially successful future transactions. Although the size of the transaction that executed right before the abort is unknown, the length of time from transaction start until the end of transactional abort (including abort time $\approx$ 150 clock cycles [RB13]) can be computed. This insight provides a way to predict the size of the next transaction potentially more precisely than our work described in Section 6.2. The key idea in time-based merging algorithm is to use the knowledge of elapsed duration in order to decide the length of transactions dynamically. The *RDTSC* instruction provides the elapsed time in cycles. At every region boundary, the time duration since the start of the transaction could decide if the transaction should be committed at that program point. Every transactional commit or abort could determine the length of time the next transaction would be allowed to run.

8.3 Providing Bounded Region Serializability with Record and Replay Support

In a system where DBRS is the default memory model, the debuggability of a program should be simpler as DBRS restricts program behavior more than other weaker memory

models. A future direction could be to provide record and replay support to concurrent programs executing under the DBRS memory model. The RDTSC instruction provides a guarantee that if $t1 < t2$, where $t1$ and $t2$ are invocations of the instruction on two different processors, then there is a *happens-before* order from the events preceding $t1$ to the events succeeding $t2$ [LGPS15]. This property can be capitalized to record a total ordering of Legato transactions along with transaction size, in terms of DBRs, in the record run. The replay run could enforce this schedule through wait or forced transactional aborts. This approach to record and replay the programs fits naturally with the DBRS memory model since the entire program is running as a sequence of transactions.

Chapter 9: Conclusion

In this chapter, we review the contribution of this dissertation in the overall context of making parallel programming with correctness easier for programmers to achieve. John Hennessy in an interview admits the challenge and according to him [O'H06], *"..when we start talking about parallelism and ease of use of truly parallel computers, we're talking about a problem that's as hard as any that computer science has faced."* David Patterson in the same interview concurs with him on the difficulty of parallel programming: *" ..Parallel programming has proven to be a really hard concept. Just because you need a solution doesn't mean you're going to find it."* It is well known that parallel programming is hard and with the incredible challenge of writing correct and scalable programs, inevitably, synchronization bugs are part of any parallel software. Under this assumption, we believe that the compilers and runtime need to provide the correct semantics with ideally low overhead—targeting production systems—even if the programmers write ill-synchronized programs in the face of the challenge of exploiting parallel hardware. In this context, we summarize how this dissertation contributes in building automatic compiler and runtime techniques to eliminate bugs in concurrent software at reasonable overheads. We also highlight how building further analyses might be easier under the execution behavior that the proposed memory model provides.

9.1 Summary

- **Chapter 3** proposes a new memory model called dynamically bounded region serializability (DBRS) and motivates its utility as a strong end-to-end memory model that could be implemented efficiently. It also discusses progress guarantees in comparison with other strong memory models like SC [Lam79] and full-SFR RS [OCFN13]. A memory model based on serializability not only helps reasoning about programs easier but enables optimizations within the region. Dynamic analysis and model checkers [LCFN12, LWV+10, HZD13, VLW+11, BKC+15, CLL+02, MQ07] that provide determinism, performance, and correctness of parallel programs essentially try to solve the following key issue: *soundly tracking inter-thread dependences in presence of data races.* DBRS can significantly simplify these analyses by reducing the set of all possible thread interleavings.

- **Chapter 4** proposes a novel hybrid static–dynamic analysis which empirically demonstrates that DBRS can be made practical in commodity systems. The analysis called EnfoRSer uses strong compiler transformations and dynamic analysis to enforce DBRS. We implement adversarial memory [FF10], an additional analysis from prior work to expose bugs on relaxed memory models and then establish that DBRS eliminates several of these concurrency errors. Low-overhead implementation of DBRS shows a new direction in enabling strong semantics in concurrent software. The key contribution of this work is the insight and empirical evidence—that atomic execution of bounded regions can eliminate several concurrency bugs. Bounding the size of regions enable novel techniques that can keep the overhead of the atomicity enforcement low. EnfoRSer pioneers enforcement of bounded region serializability on

commodity hardware—which otherwise, until this work, was only investigated by hardware researchers [AQN+09, LDSC08].

- **Chapter 5** explores a design space that uses profiling and a cost model to automatically predict an efficient synchronization mechanism for a region. Different parts of a program enforce DBRS differently. The analysis reduces the instrumentation overhead of enforcing DBRS by using a single static lock instead of per-access locks. This distinction is achieveable for regions where finer granularity of conflict detection is unnecessary. This work demonstrates that for applications with high thread locality and a large number of accesses, this hybrid mode of providing atomicity is several times less expensive. This technique uses heterogeneous forms of atomicity enforcement in a single system while maintaining correctness in providing DBRS. EnfoRSer-H highlights a new direction in DBRS enforcement where application characteristics are utilized to provide the low-overhead enforcement of DBRS. The hybridization of locks provides evidence that different program segments could enforce atomicity differently.

- **Chapter 6** takes a orthogonal approach to DBRS enforcement as compared to the previous techniques presented in the dissertation. The availability of hardware transactional memory (HTM) in commodity processors makes it a viable choice for atomicity enforcement. Legato, the approach presented in this chapter is inspired by the simple abstraction and intrinsic conflict detection capability of HTM. While conflict detection proved to be expensive in EnfoRSer, the reader-writer locks introduced additional single threaded overhead. The atomicity transformations introduced compiler complexity and register pressure. Legato attempts to address these challenges

using HTM. However, DBRS enforcement using HTM with one DBR per transaction incurs prohibitive overhead due to expensive TSX start and end instructions. Legato amortizes this overhead by merging multiple regions in a single transaction and solves the critical problem of balancing abort cost and per-transaction cost using a control theoretic approach. Its simple instrumentation at region boundaries and intrinsic conflict detection in the hardware eliminates the complexity of EnfoRSer. Legato attains stable overhead across programs of different characteristics outperforming EnfoRSer on an average. Legato is a step forward in unleashing the potential of commodity hardware in the enforcement of strong end-to-end memory consistency. More advanced hardware with improved conflict resolution (e.g. unlike the *requester wins* policy in TSX—the flexibility for the requester or the *responder* to win) and HTM friendly cache-coherence policies can simplify DBRS enforcement further.

9.2 Context and Meaning

The viewpoint of computing has changed due to constraints in technology. The designs moved focus from extracting performance out of instruction-level-parallelism in a single thread to designs that utilize thread-level parallelism. As the microarchitecture becomes more parallel with addition of cores on a die, the burden for writing correct and parallel code, is left to the software programmer. Understanding concurrency on existing memory models is a challenging task. Herb Sutter in his column for a monthly journal [Sut05] comments on this enormous challenge. According to him, *"Everybody who learns concurrency thinks they understand it, ends up finding mysterious races they thought weren't possible,*

and discovers that they didn't actually understand it yet after all." This reflects the complexity of semantics that a programmer needs to understand to avoid data races or debug data races while building a concurrent system. This dissertation attempts to address this acute problem through runtime support that provides strong semantics in programs with data races.

As discussed in **Chapter 2**, prior work proposes bounded region serializability with complex, custom hardware [CTMT07, AQN+09, MSM+10, SMN+11, LDSC08, LCS+10] for potential systems of the future. In contrast, this dissertation provides solutions that exploit commercial hardware to enable stronger semantic guarantees at reasonable runtime overheads. Commodity hardware makes the solutions readily usable.

A myriad of software rely on concurrency for performance and are afflicted by bugs which are extremely hard to detect and address. Our work strongly advocates that there is a definite need to design automatic language runtime solutions that guarantee strong semantics accepting the ubiquity of data races. The memory model that we proposed in this dissertation and the corroborating evidence—that atomically executing bounded regions could eliminate a large class of concurrency bugs—is a crucial step forward.

These findings could provide insights into designing new hardware. Adve and Boehm recommend software-hardware codesign and they believe [AB10]: *"We believe that hardware that takes advantage of the emerging disciplined software programming models is likely to be more efficient than a software oblivious approach"*. We believe, the insights and empericial evidence presented in this dissertation provide efficient solutions to enforce *data-race-freedom* on commodity hardware. While Adve and Boehm *"call upon software and hardware communities to develop languages and systems that enforce data-race-freedom, and co-designed hardware that exploits and supports such semantics.."* [AB10],

we believe this dissertation not only champions this cause by innovating commodity system solutions to data-race-freedom—but further provides insights that the hardware research community could leverage to build co-designed hardware for data-race-freedom.

Bibliography

[AAB+00] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, Susan Flynn Hummel, D. Lieber, V. Litvinov, M. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[AAB+05] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The Jikes Research Virtual Machine Project: Building an Open-Source Research Community. *IBM Systems Journal*, 44:399–417, 2005.

[AAK+05] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded Transactional Memory. In *HPCA*, pages 316–327, 2005.

[AB10] Sarita V. Adve and Hans-J. Boehm. Memory Models: A Case for Rethinking Parallel Languages and Hardware. *CACM*, 53:90–101, 2010.

[ACJL13] Joy Arulraj, Po-Chun Chang, Guoliang Jin, and Shan Lu. Production-Run Software Failure Diagnosis via Hardware Performance Counters. In *ASPLOS*, pages 101–112, 2013.

[AFG+00] Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *OOPSLA*, pages 47–65, 2000.

[AFI+08] Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. The Semantics of Power and ARM Multiprocessor Machine Code. In *DAMP*, pages 13–24, 2008.

[AG05] Matthew Arnold and David Grove. Collecting and Exploiting High-Accuracy Call Graph Profiles in Virtual Machines. In *CGO*, pages 51–62, 2005.

[AH90] Sarita V. Adve and Mark D. Hill. Weak Ordering—A New Definition. In *ISCA*, pages 2–14, 1990.

[AHM09] Martín Abadi, Tim Harris, and Mojtaba Mehrara. Transactional Memory with Strong Atomicity Using Off-the-Shelf Memory Protection Hardware. In *PPoPP*, pages 185–196, 2009.

[AOS15] Todd A. Anderson, Melissa O'Neill, and John Sarracino. Chihuahua: A Concurrent, Moving, Garbage Collector using Transactional Memory. In *TRANSACT*, 2015.

[AQN+09] W. Ahn, S. Qi, M. Nicolaides, J. Torrellas, J.-W. Lee, X. Fang, S. Midkiff, and David Wong. BulkCompiler: High-Performance Sequential Consistency through Cooperative Compiler and Hardware Support. In *MICRO*, pages 133–144, 2009.

[BA08] Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ Concurrency Memory Model. In *PLDI*, pages 68–78, 2008.

[BCM10] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. Pacer: Proportional Detection of Data Races. In *PLDI*, pages 255–268, 2010.

[BCZ+17] Swarnendu Biswas, Man Cao, Minjia Zhang, Michael D. Bond, and Benjamin P. Wood. Lightweight Data Race Detection for Production Runs. In *International Conference on Compiler Construction*, pages 11–21, 2017.

[BD14] Hans-J. Boehm and Brian Demsky. Outlawing Ghosts: Avoiding Out-of-Thin-Air Results. In *MSPC*, pages 7:1–7:6, 2014.

[BDLM07] Colin Blundell, Joe Devietti, E. Christopher Lewis, and Milo M. K. Martin. Making the Fast Case Common and the Uncommon Case Simple in Unbounded Transactional Memory. In *ISCA*, pages 24–34, New York, NY, USA, 2007.

[BGH+06] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA*, pages 169–190, 2006.

[BGH+08] Jayaram Bobba, Neelam Goyal, Mark D. Hill, Michael M. Swift, and David A. Wood. TokenTM: Efficient Execution of Large Transactions with Hardware Transactional Memory. In *ISCA*, pages 127–138, 2008.

[BHSB14] Swarnendu Biswas, Jipeng Huang, Aritra Sengupta, and Michael D. Bond. DoubleChecker: Efficient Sound and Precise Atomicity Checking. In *PLDI*, pages 28–39, 2014.

[BKC+13] Michael D. Bond, Milind Kulkarni, Man Cao, Minjia Zhang, Meisam Fathi Salmi, Swarnendu Biswas, Aritra Sengupta, and Jipeng Huang. Octet: Capturing and Controlling Cross-Thread Dependences Efficiently. In *OOPSLA*, pages 693–712, 2013.

[BKC+15] Michael D. Bond, Milind Kulkarni, Man Cao, Meisam Fathi Salmi, and Jipeng Huang. Efficient Deterministic Replay of Multithreaded Executions in a Managed Language Virtual Machine. In *PPPJ*, pages 90–101, 2015.

[BLR02] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *OOPSLA*, pages 211–230, 2002.

[BM08] Sebastian Burckhardt and Madanlal Musuvathi. Effective Program Verification for Relaxed Memory Models. In *CAV*, pages 107–120, 2008.

[Boe12] Hans-J. Boehm. Position paper: Nondeterminism is Unavoidable, but Data Races are Pure Evil. In *RACES*, pages 9–14, 2012.

[BZBL15] Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Brandon Lucia. Valor: Efficient, Software-Only Region Conflict Exceptions. In *OOPSLA*, pages 241–259, 2015.

[CCC+07] Hassan Chafi, Jared Casper, Brian D. Carlstrom, Austen McDonald, Chi Cao Minh, Woongki Baek, Christos Kozyrakis, and Kunle Olukotun. A Scalable,

Non-blocking Approach to Transactional Memory. In *HPCA*, pages 97–108, 2007.

[CCD+10] Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick Marlier, and Etienne Rivière. Evaluation of AMD's Advanced Synchronization Facility Within a Complete Transactional Memory Stack. In *EuroSys*, pages 27–40, 2010.

[CDLQ09] Luis Ceze, Joseph Devietti, Brandon Lucia, and Shaz Qadeer. A Case for System Support for Concurrency Exceptions. In *HotPar*, 2009.

[CHM16] Keith Chapman, Antony L. Hosking, and J. Eliot B. Moss. Hybrid STM/HTM for Nested Transactions on OpenJDK. In *OOPSLA*, pages 660–676, 2016.

[CLL+02] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. In *PLDI*, pages 258–269, 2002.

[CRSB16] Man Cao, Jake Roemer, Aritra Sengupta, and Michael D. Bond. Prescient Memory: Exposing Weak Memory Model Behavior by Looking into the Future. In *ISMM*, pages 99–110, 2016.

[CTMT07] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. BulkSC: Bulk Enforcement of Sequential Consistency. In *ISCA*, pages 278–289, 2007.

[CVJL08] Ravi Chugh, Jan W. Voung, Ranjit Jhala, and Sorin Lerner. Dataflow Analysis for Concurrent Programs using Datarace Detection. In *PLDI*, pages 316–326, 2008.

[CZB14] Man Cao, Minjia Zhang, and Michael D. Bond. Drinking from Both Glasses: Adaptively Combining Pessimistic and Optimistic Synchronization for Efficient Parallel Runtime Support. In *WoDet*, 2014.

[CZSB16] Man Cao, Minjia Zhang, Aritra Sengupta, and Michael D. Bond. Drinking from Both Glasses: Combining Pessimistic and Optimistic Tracking of Cross-Thread Dependences. In *PPoPP*, pages 20:1–20:13, 2016.

[DGB+03] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The Transmeta Code Morphing™ Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges. In *CGO*, pages 15–24, 2003.

[DKL+14] Dave Dice, Alex Kogan, Yossi Lev, Timothy Merrifield, and Mark Moir. Adaptive Integration of Hardware and Software Lock Elision Techniques. In *SPAA*, pages 188–197, 2014.

[DS09] Luke Dalessandro and Michael L. Scott. Strong Isolation is a Weak Idea. In *TRANSACT*, 2009.

[DSS10a] Luke Dalessandro, Michael L. Scott, and Michael F. Spear. Transactions as the Foundation of a Memory Consistency Model. In *DISC*, pages 20–34, 2010.

[DSS10b] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. NOrec: Streamlining STM by Abolishing Ownership Records. In *PPoPP*, pages 67–78, 2010.

[EDLC+12] Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-J. Boehm. IFRit: Interference-Free Regions for Dynamic Data-Race Detection. In *OOPSLA*, pages 467–484, 2012.

[EQT07] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: A Race and Transaction-Aware Java Runtime. In *PLDI*, pages 245–255, 2007.

[FF09] Cormac Flanagan and Stephen N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *PLDI*, pages 121–133, 2009.

[FF10] Cormac Flanagan and Stephen N. Freund. Adversarial Memory for Detecting Destructive Races. In *PLDI*, pages 244–254, 2010.

[FF13] Cormac Flanagan and Stephen N. Freund. RedCard: Redundant Check Elimination for Dynamic Race Detectors. In *ECOOP*, pages 255–280, 2013.

[FFY08] Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. In *PLDI*, pages 293–303, 2008.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and Its Use in Optimization. *TOPLAS*, 9(3):319–349, 1987.

[GG91] Kourosh Gharachorloo and Phillip B. Gibbons. Detecting Violations of Sequential Consistency. In *SPAA*, pages 316–326, 1991.

[GN08] P. Godefroid and N. Nagappan. Concurrency at Microsoft – An Exploratory Survey. In EC^2, 2008.

[Gra04] Paul Graham. *Hackers and Painters*. O'Reilly Media, 2004.

[GSC+09] Shantanu Gupta, Florin Sultan, Srihari Cadambi, Franjo Ivančić, and Martin Rötteler. Using Hardware Transactional Memory for Data Race Detection. In *IPDPS*, pages 1–11, 2009.

[HF03] Tim Harris and Keir Fraser. Language Support for Lightweight Transactions. In *OOPSLA*, pages 388–402, 2003.

[HG06] Benjamin Hindman and Dan Grossman. Atomicity via Source-to-Source Translation. In *MSPC*, pages 82–91, 2006.

[HLR10] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory*. Morgan and Claypool Publishers, 2nd edition, 2010.

[HLZ10] Jeff Huang, Peng Liu, and Charles Zhang. LEAP: Lightweight Deterministic Multi-processor Replay of Concurrent Java Programs. In *FSE*, pages 207–216, 2010.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ISCA*, pages 289–300, 1993.

[HRB88] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. In *PLDI*, pages 35–46, 1988.

[HWC+04] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos

Kozyrakis, and Kunle Olukotun. Transactional Memory Coherence and Consistency. In *ISCA*, pages 102–113, 2004.

[HZD13] Jeff Huang, Charles Zhang, and Julian Dolby. CLAP: Recording Local Executions to Reproduce Concurrency Failures. In *PLDI*, pages 141–152, 2013.

[int12] *Intel Architecture Instruction Set Extensions Programming Reference*, chapter 8: Intel transactional synchronization extensions. Intel Corporation, 2012.

[KKO02] Kiyokuni Kawachiya, Akira Koseki, and Tamiya Onodera. Lock Reservation: Java Locks Can Mostly Do Without Atomic Operations. In *OOPSLA*, pages 130–141, 2002.

[Lam78] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *CACM*, 21(7):558–565, 1978.

[Lam79] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Computer*, 28:690–691, 1979.

[LCFN12] Dongyoon Lee, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Chimera: Hybrid Program Analysis for Determinism. In *PLDI*, pages 463–474, 2012.

[LCS+10] Brandon Lucia, Luis Ceze, Karin Strauss, Shaz Qadeer, and Hans-J. Boehm. Conflict Exceptions: Simplifying Concurrent Language Semantics with Precise Hardware Exceptions for Data-Races. In *ISCA*, pages 210–221, 2010.

[LDSC08] Brandon Lucia, Joseph Devietti, Karin Strauss, and Luis Ceze. Atom-Aid: Detecting and Surviving Atomicity Violations. In *ISCA*, pages 277–288, 2008.

[LGPS15] Yujie Liu, Justin Gottschlich, Gilles Pokam, and Michael Spear. TSXProf: Profiling Hardware Transactions. In *PACT*, pages 75–86, 2015.

[LNG10] Changhui Lin, Vijay Nagarajan, and Rajiv Gupta. Efficient Sequential Consistency Using Conditional Fences. In *PACT*, pages 295–306, 2010.

[LNGR12] Changhui Lin, Vijay Nagarajan, Rajiv Gupta, and Bharghava Rajaram. Efficient Sequential Consistency via Conflict Ordering. In *ASPLOS*, pages 273–286, 2012.

[LPSZ08] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *ASPLOS*, pages 329–339, 2008.

[LTQZ06] Shan Lu, Joe Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: Detecting Atomicity Violations via Access-Interleaving Invariants. In *ASPLOS*, pages 37–48, 2006.

[LWV+10] Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter M. Chen, and Jason Flinn. Respec: Efficient Online Multiprocessor Replay via Speculation and External Determinism. In *ASPLOS*, pages 77–90, 2010.

[LXG+14] Y. Liu, Y. Xia, H. Guan, B. Zang, and H. Chen. Concurrent and Consistent Virtual Machine Introspection with Hardware Transactional Memory. In *HPCA*, pages 416–427, Feb 2014.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Prentice Hall PTR, 2nd edition, 1999.

[MBM+06] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based Transactional Memory. In *HPCA*, pages 254–265, 2006.

[MK12] Jason Mars and Naveen Kumar. BlockChop: Dynamic Squash Elimination for Hybrid Processor Architecture. In *ISCA*, pages 536–547, 2012.

[MKTD14] Hassan Salehe Matar, Ismail Kuru, Serdar Tasiran, and Roman Dementiev. Accelerating Precise Race Detection Using Commercially-Available Hardware Transactional Memory Support. In *WoDet*, 2014.

[MMN09] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. LiteRace: Effective Sampling for Lightweight Data-Race Detection. In *PLDI*, pages 134–143, 2009.

[MPA05] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java Memory Model. In *POPL*, pages 378–391, 2005.

[MQ07] Madanlal Musuvathi and Shaz Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In *PLDI*, pages 446–455, 2007.

[MSM+10] Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. DRFx: A Simple and Efficient Memory Model for Concurrent Programming Languages. In *PLDI*, pages 351–362, 2010.

[MSM+11] Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. A Case for an SC-Preserving Compiler. In *PLDI*, pages 199–210, 2011.

[NA07] Mayur Naik and Alex Aiken. Conditional Must Not Aliasing for Static Race Detection. In *POPL*, pages 327–338, 2007.

[NAW06] Mayur Naik, Alex Aiken, and John Whaley. Effective Static Race Detection for Java. In *PLDI*, pages 308–319, 2006.

[OAA09] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. In *ASPLOS*, pages 97–108, 2009.

[OCFN13] Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. ...and region serializability for all. In *HotPar*, 2013.

[O'H06] Charlene O'Hanlon. A Conversation with John Hennessy and David Patterson. *Queue*, 4(10):14–22, December 2006.

[OS15] Melissa E. O'Neill and Christopher A. Stone. Making Impractical Implementations Practical: Observationally Cooperative Multithreading Using HLE. In *TRANSACT*, 2015.

[PAM+08] Seth H. Pugsley, Manu Awasthi, Niti Madan, Naveen Muralimanohar, and Rajeev Balasubramonian. Scalable and Reliable Communication for Hardware Transactional Memory. In *PACT*, pages 144–154, 2008.

[PZX+09] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. In *SOSP*, pages 177–192, 2009.

[RB13] Carl G. Ritson and Frederick R.M. Barnes. An Evaluation of Intel's Restricted Transactional Memory for CPAs. In *CPA*, pages 271–292, 2013.

[RD06] Kenneth Russell and David Detlefs. Eliminating Synchronization-Related Atomic Operations with Biased Locking and Bulk Rebiasing. In *OOPSLA*, pages 263–272, 2006.

[RDB99] Michiel Ronsse and Koen De Bosschere. RecPlay: A Fully Integrated Practical Record/Replay System. *TOCS*, 17:133–152, 1999.

[RG01] Ravi Rajwar and James R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *MICRO*, pages 294–305, 2001.

[RHH09] Amitabha Roy, Steven Hand, and Tim Harris. A Runtime System for Software Lock Elision. In *EuroSys*, pages 261–274, 2009.

[RPA97] Parthasarathy Ranganathan, Vijay Pai, and Sarita Adve. Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap between Memory Consistency Models. In *SPAA*, pages 199–210, 1997.

[RUJ14] Carl G. Ritson, Tomoharu Ugawa, and Richard E. Jones. Exploring Garbage Collection with Haswell Hardware Transactional Memory. In *ISMM*, pages 105–115, 2014.

[SATH+06] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: A High Performance Software Transactional Memory System for a Multi-Core Runtime. In *PPoPP*, pages 187–197, 2006.

[SBO01] L. A. Smith, J. M. Bull, and J. Obdrzálek. A Parallel Java Grande Benchmark Suite. In *SC*, pages 8–8, 2001.

[SBZ+15] Aritra Sengupta, Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Milind Kulkarni. Hybrid Static–Dynamic Analysis for Statically Bounded Region Serializability. In *ASPLOS*, pages 561–575, 2015.

[SCBK15] Aritra Sengupta, Man Cao, Michael D. Bond, and Milind Kulkarni. Toward Efficient Strong Memory Model Support for the Java Platform via Hybrid Synchronization. In *PPPJ*, pages 65–75, 2015.

[A08] Jaroslav Ševčík and David Aspinall. On Validity of Program Transformations in the Java Memory Model. In *ECOOP*, pages 27–51, 2008.

[SFW+05] Zehra Sura, Xing Fang, Chi-Leung Wong, Samuel P. Midkiff, Jaejin Lee, and David Padua. Compiler Techniques for High Performance Sequentially Consistent Java Programs. In *PPoPP*, pages 2–13, 2005.

[SGT96] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *ASPLOS*, pages 174–185, 1996.

[SI94] CORPORATE SPARC International, Inc. *The SPARC Architecture Manual: Version 9*. 1994.

[SMAT+07] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing Isolation and Ordering in STM. In *PLDI*, pages 78–88, 2007.

[SMN+11] Abhayendra Singh, Daniel Marino, Satish Narayanasamy, Todd Millstein, and Madan Musuvathi. Efficient Processor Support for DRFx, a Memory Model with Exceptions. In *ASPLOS*, pages 53–66, 2011.

[SNM+12] Abhayendra Singh, Satish Narayanasamy, Daniel Marino, Todd Millstein, and Madanlal Musuvathi. End-to-End Sequential Consistency. In *ISCA*, pages 524–535, 2012.

[SS88] Dennis Shasha and Marc Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. *TOPLAS*, 10(2):282–312, 1988.

[SSA+11] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER Multiprocessors. In *PLDI*, pages 175–186, 2011.

[Sut05] Herb Sutter. The Free Lunch is Over. *Dr. Dobb's Journal*, 30, March 2005.

[TNGT08] Chen Tian, Vijay Nagarajan, Rajiv Gupta, and Sriraman Tallam. Dynamic Recognition of Synchronization Operations for Improved Data Race Detection. In *ISSTA*, pages 143–154, 2008.

[UBES09] Takayuki Usui, Reimer Behrends, Jacob Evans, and Yannis Smaragdakis. Adaptive Locks: Combining Transactions and Locks for Efficient Concurrency. In *PACT*, pages 3–14, 2009.

[U.S04] U.S.–Canada Power System Outage Task Force. Final Report on the August 14th Blackout in the United States and Canada. Technical report, Department of Energy, 2004.

[VLW+11] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. DoublePlay: Parallelizing Sequential Logging and Replay. In *ASPLOS*, pages 15–26, 2011.

[vPG01] Christoph von Praun and Thomas R. Gross. Object Race Detection. In *OOPSLA*, pages 70–82, 2001.

[vPG03] Christoph von Praun and Thomas R. Gross. Static Conflict Analysis for Multi-Threaded Object-Oriented Programs. In *PLDI*, pages 115–128, 2003.

[WGW+12] Amy Wang, Matthew Gaudet, Peng Wu, José Nelson Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael. Evaluation of Blue Gene/Q Hardware Support for Transactional Memories. In *PACT*, pages 127–136, 2012.

[WMvP+09] Peng Wu, Maged M. Michael, Christoph von Praun, Takuya Nakaike, Rajesh Bordawekar, Harold W. Cain, Calin Cascaval, Siddhartha Chatterjee, Stefanie Chiras, Rui Hou, Mark Mergen, Xiaowei Shen, Michael F. Spear, Hua Yong Wang, and Kun Wang. Compiler and Runtime Techniques for Software Transactional Memory Optimization. *Concurrency and Computation: Practice and Experience*, 21(1):7–23, 2009.

[YBM+07] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In *HPCA*, pages 261–272, 2007.

[YHLR13] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. Performance Evaluation of Intel Transactional Synchronization Extensions for High-Performance Computing. In *SC*, pages 19:1–19:11, 2013.

[ZdKL+13] Wei Zhang, Marc de Kruijf, Ang Li, Shan Lu, and Karthikeyan Sankaralingam. ConAir: Featherweight Concurrency Bug Recovery via Single-threaded Idempotent Execution. In *ASPLOS*, pages 113–126, 2013.

[ZHCB15] Minjia Zhang, Jipeng Huang, Man Cao, and Michael D. Bond. Low-Overhead Software Transactional Memory with Progress Guarantees and Strong Semantics. In *PPoPP*, pages 97–108, 2015.

[ZLH+15] Long Zheng, Xiaofei Liao, Bingsheng He, Song Wu, and Hai Jin. On Performance Debugging of Unnecessary Lock Contentions on Multicore Processors: A Replay-based Approach. In *CGO*, pages 56–67, 2015.

[ZLJ16] Tong Zhang, Dongyoon Lee, and Changhee Jung. TxRace: Efficient Data Race Detection Using Commodity Hardware Transactional Memory. In *ASPLOS*, pages 159–173, 2016.

[ZWAT+08] Lukasz Ziarek, Adam Welc, Ali-Reza Adl-Tabatabai, Vijay Menon, Tatiana Shpeisman, and Suresh Jagannathan. A Uniform Transactional Execution Environment for Java. In *ECOOP*, pages 129–154, 2008.