2018

Strong Memory Consistency For Parallel Programming

Christian Delozier
University of Pennsylvania, crdelozier@gmail.com

Strong Memory Consistency For Parallel Programming

Abstract
Correctly synchronizing multithreaded programs is challenging, and errors can lead to program failures (e.g., atomicity violations). Existing memory consistency models rule out some possible failures, but are limited by depending on subtle programmer-defined locking code and by providing unintuitive semantics for incorrectly synchronized code. Stronger memory consistency models assist programmers by providing them with easier-to-understand semantics with regard to memory access interleavings in parallel code. This dissertation proposes a new strong memory consistency model based on ordering-free regions (OFRs), which are spans of dynamic instructions between consecutive ordering constructs (e.g. barriers). Atomicity over ordering-free

regions provides stronger atomicity than existing strong memory consistency models with competitive performance. Ordering-free regions also simplify programmer reasoning by limiting the potential for atomicity violations to fewer points in the program's execution. This dissertation explores both software-only and hardware-supported systems that provide OFR serializability.

Degree Type
Dissertation

Degree Name
Doctor of Philosophy (PhD)

Graduate Group
Computer and Information Science

First Advisor
Joseph Devietti

Second Advisor
Steve Zdancewic

Keywords
Atomicity, Memory Consistency, Parallel Programming

Subject Categories
Computer Engineering | Computer Sciences

STRONG MEMORY CONSISTENCY FOR PARALLEL PROGRAMMING

Christian DeLozier

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2018

Supervisor of Dissertation

_____

Joseph Devietti

Assistant Professor of Computer and Information Science

Graduate Group Chairperson

_____

Lyle Ungar

Professor of Computer and Information Science

Dissertation Committee

Steve Zdancewic, Professor of Computer and Information Science

Boon Thau Loo, Professor of Computer and Information Science

Jonathan Smith, Olga and Alberico Pompa Professor of Engineering and Applied Science

Brandon Lucia, Assistant Professor of Computer Engineering, Carnegie Mellon University

STRONG MEMORY CONSISTENCY FOR PARALLEL PROGRAMMING

To my wife Allyson and my working buddies Lily, Guen, and Tony

ACKNOWLEDGMENTS

I would like to thank my advisor, Joe Devietti, for his enthusiasm and support over the past five years. Joe has been an excellent mentor, and I would not have been able to finish this thesis without his help. Joe has constantly provided me with interesting problems to work on and helpful insights on how to solve those problems. His enthusiasm for research in Computer Science is unparalleled and has inspired me to continue performing research in my career. He has also been understanding and supportive of my desire to teach and has allowed me to siphon a significant amount of time from research in pursuit of teaching opportunities. I am proud to have been Joe's first PhD student.

I would also like to thank Milo Martin for advising me through the first three years of my PhD. Milo mentored me through my first research project, and his advice has stuck with me throughout the years and will continue to influence how I conduct research in the future.

I thank my dissertation committee chaired by Steve Zdancewic, and with Boon Thau Loo, Jonathan Smith, and Brandon Lucia as members, for their comments and guidance on this dissertation. I would especially like to thank Brandon Lucia for collaborating on this work for many years and suffering through the countless rejections and rewrites. I would also like to thank Steve Zdancewic for his help on my WPE II committee, for collaborating on my first research project, and for his advice on teaching.

I owe a huge amount of thanks and gratitude to my parents for helping me to achieve this goal. Without their love, support, and encouragement, I would not have been able to pursue a PhD. They provided me with every opportunity to further my education and succeed. They have both been patient in listening to me when I struggled and offering thoughtful advice on how to proceed throughout the years. Thank you as well to my sister Laura for her love and for helping me with my teaching materials and techniques over the past few years. I also need to thank all the Heslops for being a second family to me.

I was lucky to collaborate with a fantastic group of people during my time at Penn. Thank you to Peter-Michael Osera, Richard Eisenberg, Santosh Nagarakatte, Yuanfeng Peng, Ariel Eizenberg, Kavya Lakshminarayanan, Shiliang Hu, and Gilles Pokam for their ideas, insights, advice, and work over the many years I have been at Penn. The work in this thesis and outside of it would not have been possible without such great collaborators.

In my time at Penn, I have also made a number of friends who have made campus a friendly and fun environment. I would like to thank Abhishek Udupa, Arun Raghavan, Santosh Nagarakatte, Laurel Emurian, Salar Moareff, Mukund Raghothaman, Ben Karel, Nikos Vasilakis, Yuanfeng Peng, Nimit Sighania, Peter-Michael Osera, Jennifer Paykin, Katie Gibson, Omar Navarro Leija, Kelly Shiptoski, Akshitha Sriraman, and Liang Luo for their friendship and advice. I apologize if I missed anyone, and I hope to keep in touch in years to come.

Outside of Penn, I would also like to thank my extended family for their love and support. I would like to thank my friends from Pitt and elsewhere for helping me to have fun and escape from school. Thanks especially to Josh Picozzi, Dave Hynek, Emily DeLeo, and Holey Boley.

I would not have been interested in graduate school if not for the early research experiences I had at the University of Pittsburgh. I would like to thank Brian Primack and Bruce Childers for getting me started in research and encouraging me to continue. I would also like to thank John Ramirez and John Aronis for helping me prepare for and apply to graduate school.

Last but certainly not least, I would like to thank my wife Allyson for her love, support, encouragement, and patience over the past eight years. Allyson has tirelessly listened to my out-loud thinking about research, paper reviews, rebuttals, rejections, and occasionally acceptances over the time I have worked on my PhD. Without her emotional support, I would not have earned this degree. She has also preserved my sanity by forcing me to occasionally step away from my research to explore the world, both near and far. Allyson, I love you, and you probably deserve to have your name on this work as much as I do.

The work in this dissertation was supported by NSF grant #XPS-1337174.

ABSTRACT

STRONG MEMORY CONSISTENCY FOR PARALLEL PROGRAMMING

Christian DeLozier

Joseph Devietti

Correctly synchronizing multithreaded programs is challenging, and errors can lead to program failures (e.g., atomicity violations). Existing memory consistency models rule out some possible failures, but are limited by depending on subtle programmer-defined locking code and by providing unintuitive semantics for incorrectly synchronized code. Stronger memory consistency models assist programmers by providing them with easier-to-understand semantics with regard to memory access interleavings in parallel code. This dissertation proposes a new strong memory consistency model based on ordering-free regions (OFRs), which are spans of dynamic instructions between consecutive ordering constructs (e.g. barriers). Atomicity over ordering-free regions provides stronger atomicity than existing strong memory consistency models with competitive performance. Ordering-free regions also simplify programmer reasoning by limiting the potential for atomicity violations to fewer points in the program's execution. This dissertation explores both software-only and hardware-supported systems that provide OFR serializability.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

LIST OF ALGORITHMS

1 INTRODUCTION

Despite decades of research progress, writing correct and efficient multi-threaded programs remains an open challenge. Multi-threaded applications are becoming increasingly common on all execution platforms, including the cloud, mobile devices, and even embedded systems [30]. Given the pervasiveness of multi-threaded code and increasing complexity of systems, programmers must be able to manage the complexity of developing parallel applications.

Writing a parallel program can be divided into two subtasks: identifying the parallelism in a problem and properly expressing that parallelism in an implementation. Programmers are creative and generally good at identifying parallelism in problems, but they frequently make mistakes while trying to express that parallelism. This dissertation focuses on helping the programmer properly express parallelism. Properly expressing parallelism requires both writing code that implements a parallel task and coordinating the execution of that code so that the parallel execution produces a correct result. Without proper coordination, parallel applications can suffer from bugs such as data-races and atomicity violations that can lead to program crashes or silent data corruption.

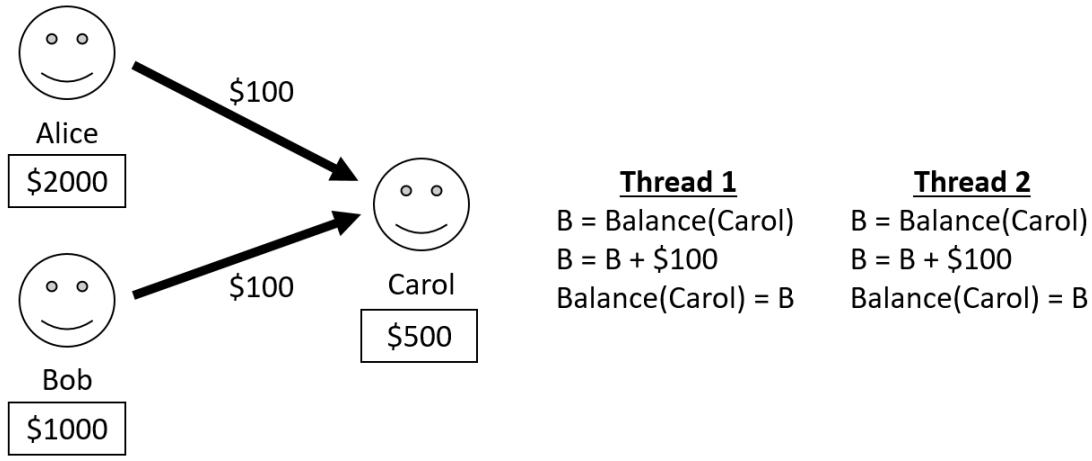

Figure 1.1: An atomicity violation that may occur when processing money transfers

As an example, consider the problem of processing transfers of money from one bank account to another. Processing one transfer at a time is safe but does not take advantage of multi-core hardware. To improve performance, transfers can be processed concurrently in multiple threads, but a multi-threaded implementation must ensure that transfers to or from the same accounts are not processed concurrently. Figure 1.1 demonstrates how transferring money in parallel may lead to an error. If Alice and Bob each transfer $100 to Carol and those transfers are processed concurrently, Carol could end up with less money than she expects. To process each transfer, the machine would read Carol's account balance, add $100 to that balance, and then save the new balance. If the balance reads happen at the same time, Carol may end up with $600 instead of $700. Parallel programming models must help the programmer reason about concurrent executions in order to avoid bugs.

A system's memory consistency model is a key factor in helping the programmer understand how a parallel execution executes on a multi-core system. The memory models for languages like Java [52], and C++ [11], and for various hardware architectures [51, 66, 68] permit aggressive optimization, but tend to be complex and inaccessible to most programmers due to the fact that they allow instructions to be interleaved in ways that a programmer may not expect. Systems with a Sequentially Consistent model [10, 16, 53, 76] give parallel executions sequential interleaving semantics, but do so at instruction granularity, which remains complex. Yet stronger models have semantics that execute coarse-grained, parallel code regions atomically and as a sequential interleaving [8, 28, 48, 67]. Reasoning about a system with region interleaving semantics is much simpler than reasoning about instruction interleaving [61].

Region-based systems for enforcing strong memory consistency must decide how to define the coarse-grained code regions that are interleaved during an execution. Some systems define regions arbitrarily [53] or implicitly [67] according to program sub-structures. The size of such regions is limited by architectural parameters and features of the program unrelated to parallelism. Other systems [8, 48] define regions in terms of programmer-provided

synchronization operations, such as synchronization-free regions [48, 61] (SFRs), release-free regions [8] (RFRs), and interference-free regions [28]. The size of regions determines how safe the execution will be and how easy it is for a programmer to reason about the parallel execution.

This work proposes a new region-based memory consistency model. The key insight is to provide coarse regions by default that are not as dependent on the correctness of the program's synchronization as prior models. Thus, the need to trust programmer-defined locking code, which can often be incorrect even in well-tested applications, is largely eliminated from the process of writing a parallel application. The programmer defines a thread's work and defines the points when threads' operations must explicitly form an order – at thread creation and completion, condition wait, and barrier wait. The sequence of dynamically executed instructions between ordering primitives is referred to as an ordering-free region (OFR). A program is OFR serializable if a program's behavior is equivalent to a serialization of atomically-executed ordering-free regions. Although OFR serializability is defined as the serialization of regions, the implementation does not necessarily need to serialize all ordering-free regions and can instead execute them partially or completely in parallel, as long as the illusion of serialization is preserved.

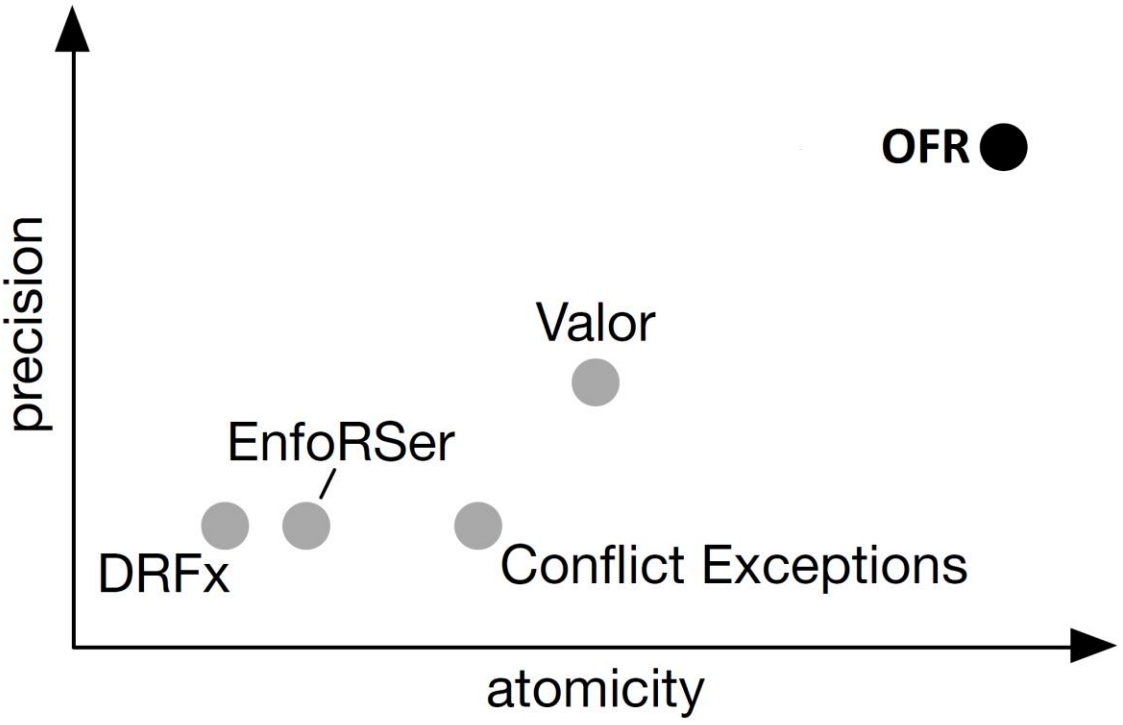

Figure 1.2: Comparison between OFR serializability and prior models

Figure 1.2 provides a visual comparison of the atomicity provided by various region-based memory consistency models. EnfoRSer [67], DRFx [53], and Conflict Exceptions [48] are all synchronization-free region (SFR) models, which enforce regions based on the acquisition and release of locks. Valor [8] is based on release-free regions (RFRs), which enforce regions only at lock release boundaries. OFRs provide more atomicity and precision than both SFR and RFR models. Later sections provide both quantitative and qualitative evaluations to support this claim.

Ordering-free regions provide a number of benefits to the programmability of parallel applications. The main benefit is that OFRs provide more atomicity than existing strong memory consistency models by enforcing atomicity across larger regions of code and more memory locations. Within an OFR, every memory location accessed by a thread is accessed atomically (without conflicting interference from another thread). Runtime systems based on ordering-free regions can help inform the necessary locking synchronization in a parallel application, and an initial user study indicated that novice programmers are better at applying correct synchronization

with the assistance of reports from an ordering-free region system than with the assistance of reports from data-race detection systems.

The subsequent chapters of this dissertation are organized as follows. Chapter 2 discusses background and related work that informed the design and implementation of ordering-free regions. Chapter 3 introduces ordering-free regions and formalizes the guarantees provided by ordering-free regions. Chapter 4 describes and compares multiple implementations or ordering-free regions, including the benefits and drawbacks of each implementation. Chapter 5 introduces hardware support for ordering-free regions. Chapter 6 provides examples and experiences in applying ordering-free regions to existing parallel applications through multiple case studies and experiments. Chapter 7 compares the performance of various implementations of software-only and hardware-supported implementations of ordering-free regions. Chapter 8 describes possible avenues for future work, and chapter 9 concludes the dissertation.

This dissertation draws on multiple published works. The Java implementation of OFR serializability, MAMA (Mostly Automatic Management of Atomicity), described in section 4 and evaluated in sections 6 and 7, was originally presented at WoDet 2014 [20]. Hardware support for OFR serializability included in sections 4 and 5 was originally proposed in the ORCA (Ordering-Free Regions for Consistency and Atomicity) technical report [23]. The technical report on ORCA also included an evaluation of the usability and performance of ordering-free region serializability with hardware support, as discussed in sections 6 and 7. SOFRITAS (Serializable Ordering-Free Regions for Increasing Thread Atomicity Scalably) was presented at ASPLOS 2018 [22]. Section 3 defines ordering-free regions and related terms as described by SOFRITAS, and section 4 provides details on the implementation of SOFRITAS. Sections 6 and 7 include the evaluation of SOFRITAS. The memory allocator described in section 4 was first introduced by TMI [21], which was presented at MICRO 2017. The remaining content is original work.

2 BACKGROUND

Strong memory consistency based on ordering-free regions is motivated by several areas of prior work on multithreaded programmability. This section provides a brief overview of each of these related areas of work and discusses recent works on each topic.

2.1 MEMORY CONSISTENCY

The **memory consistency model** of a system specifies how memory accesses in a parallel system will behave, as observed by the programmer [2]. Strong memory consistency models require memory accesses to behave as if they had been executed serially. Weak memory consistency models permit memory accesses to be reordered with respect to the serial order. Memory accesses can be reordered by compiler and hardware optimizations that can improve the performance of both serial and parallel executions. However, reordering memory accesses can be confusing to the programmer because an application may produce results that do not seem like they should be possible.

One of the strongest memory consistency models is **sequential consistency**. Sequential consistency requires that "the result of an execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program" [42]. In general, sequential consistency prevents loads and stores from being reordered with other loads and stores. Weaker consistency models permit more reorderings of loads and stores because reordering loads and stores can improve the overall performance of the system. x86 architectures implement total-store order, which allows loads to be reordered, but not stores. More relaxed consistency models, such as those implemented by ARM processors, permit reordering both loads and stores.

Many approaches to providing strong atomicity focus on enforcing sequential consistency. Several schemes have been proposed for detecting sequential-consistency violations with custom hardware support [27, 55, 62]. These systems detect violations of sequential consistency and halt the program. Enforcing sequential consistency forms a baseline for other strong memory consistency models. Other systems provide sequential consistency for a parallel execution [10, 16, 53, 76]. These systems require hardware modifications to existing systems because current processors do not support sequential consistency.

2.2 DATA-RACES AND ATOMICITY VIOLATIONS

Although sequential consistency is a useful property, it only prevents incorrect results due to memory reorderings. Parallel programs must still use synchronization to avoid data-races and atomicity violations. A ***data-race*** occurs when two memory operations on different threads access the same location, one of the accesses is a write, and there is no happens-before ordering between the two memory operations. A ***happens-before ordering*** is a chronological ordering between two threads due to some operation, such as a lock acquire or a barrier.

An ***atomicity violation*** occurs when a set of memory accesses on one thread does not have a happens-before ordering with a set of memory accesses performed by another thread or threads and at least one of the memory accesses in both sets is a write to the same location. Data-races can be considered atomicity violations, but not all atomicity violations are data-races. Figure 2.1 demonstrates an atomicity violation where each memory location (*str* and *length)* is properly protected by a lock, but the set of memory accesses performed by both threads exhibit an atomicity violation due to the lack of a happens-before ordering between the sets of accesses. The reads of *str* and *length* by Thread 1 need to occur within the same critical section to be ordered with the writes performed by Thread 2. No data-race will be detected in the code because each access has a happens-before ordering with all other accesses due the acquisition of lock L.

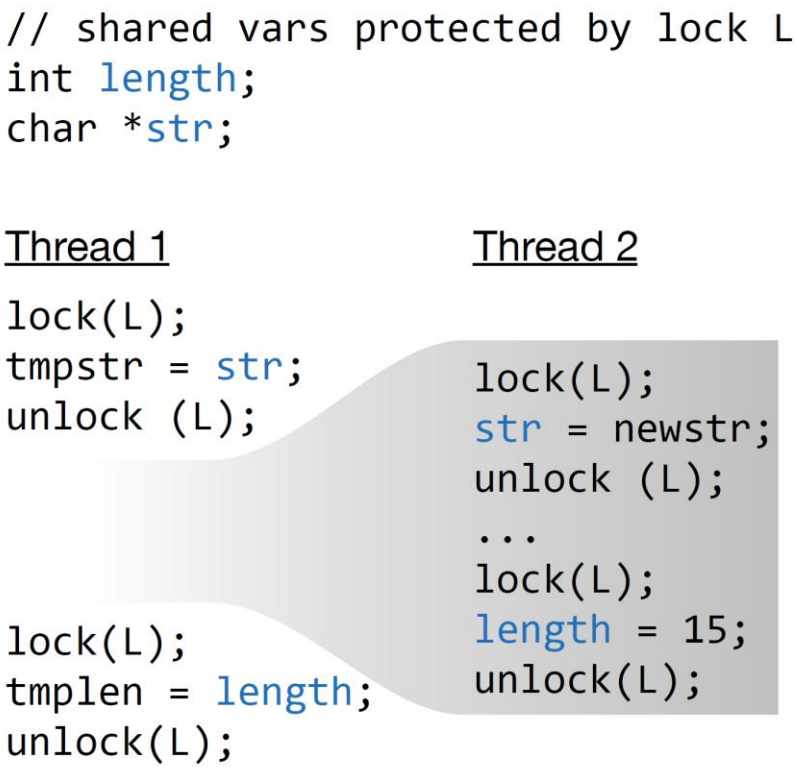

Figure 2.1: An atomicity violation found in Firefox

Many prior works have developed systems for detecting data-races and atomicity violations [17, 18, 31, 33, 44–46, 49, 62, 77]. Data-race detectors generally use vector clock algorithms to determine when a happens-before ordering does not exist between pairs of memory accesses. Atomicity violation detectors use heuristics to decide where atomic regions should start and end, allowing atomicity violations to be detected when the application's synchronization provides less atomicity than might be necessary.

2.3 STRONG MEMORY CONSISTENCY MODELS

There have been several proposals of strong memory consistency models that help catch bugs and simplify program reasoning. Although sequential consistency can help the programmer to understand a parallel execution, it still forces the programmer to reason about how individual instructions can be interleaved. Region-based memory models group instructions into regions of

code and allow the programmer to reason about the interleaving of regions rather than the interleaving of individual instructions. As Figure 1.2 shows, strong memory consistency models can be characterized along two dimensions: the granularity of the code regions at which serializability is guaranteed, and the precision with which serializability violations are detected. Ordering-free regions improve upon prior work along both dimensions.

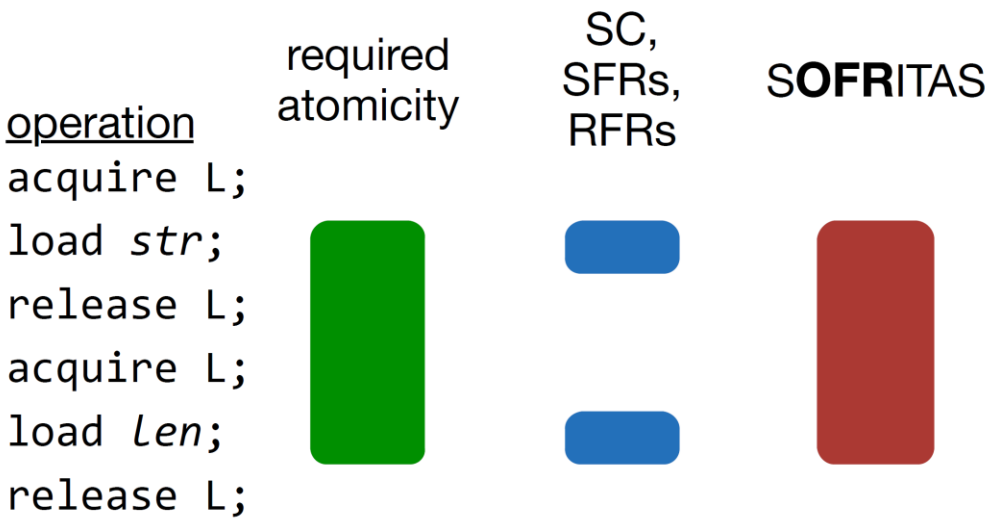

Figure 2.2: Code that demonstrates that programmer-defined acquires and releases may not match the required atomicity of an application

Synchronization-free regions [48, 61] (SFRs) span a region of instructions from one dynamic synchronization operation to the next. For example, a critical-section (between a lock and unlock operation) forms a synchronization-free region. Synchronization operations considered by synchronization-free regions include lock operations, barrier waits, condition variable waits, thread joins, thread creates, and other such operations. Release-free regions [8] (RFRs) strengthen the atomicity provided by synchronization-free regions by ending regions on release operations only. Only a subset of synchronization operations are considered to be release operations, including lock releases, barrier waits, condition variable waits, and thread creates. More generally, a release operation forms a happens-before order with other synchronization operations that occur later than itself in a trace of a program's execution [28]. All

operations that are not considered release operations are acquire operations. Interference-free regions [28] (IFRs) provide stronger atomicity than both synchronization-free regions and release-free regions. The interference-free region for a memory location extends from the acquire operation prior to the memory access to the first release operation that occurs after that access. The IFR for a memory location can be extended in both directions (of the execution trace) if it can be proven that the synchronization that formed a region should not affect the memory location. For example, in Figure 2.2, the IFRs for str and length match the critical regions that contain them. If the program used another variable x before, between, and after these two critical sections, the IFR for x would extend through the entire listed program because x should not be affected by the synchronization present in the program. Although IFRs offer stronger atomicity than both SFRs and RFRs, using IFRs in practice can be difficult because determining a memory location's IFR requires a trace of the application's execution, which is not available at runtime.

2.3.1 LENGTH AND WIDTH OF REGIONS

Region-based parallel programming models like synchronization-free regions, release-free regions, and ordering-free regions all provide atomicity over both instructions and memory locations. The ***length*** of the region is the number of dynamic instructions between the start and end of the region. In general, longer regions provide more atomicity but can lead to more conflicts between regions. Ordering-free regions (OFRs), which will be defined in Section 3, tend to be longer than SFRs and RFRs because ordering constructs tend to be used less frequently in existing parallel applications than lock acquires and releases. The ***width*** of a region is the number of memory locations that are atomic within the region, meaning that no other concurrent region can access those memory locations in a way that conflicts with the atomicity required by the region. For SFR, RFR, and OFR models, the width of the region is related to the length in that all memory location accessed by the dynamic instructions will be accessed atomically. Other

models, such as data-centric synchronization, may not protect all memory locations within each region and have regions that are less wide.

2.4 SERIALIZABILITY

A parallel execution is **serializable** If it is equivalent to some serial execution. Serializability is a desirable safety property for parallel applications because it is easier for programmers to reason about serial code than to reason about parallel code. Strong memory consistency models enforce region serializability. An execution is **region serializable** if the parallel execution of its regions, as defined by the region-based memory consistency model, is the same as some serial execution of its regions.

The concepts of atomicity and serializability are not only found in parallel programming. Databases and distributed systems have similar notions of atomicity and serializability, and one common implementation of serializability in these systems is **conflict-serializability**. Conflict-serializability requires that the parallel schedule of transactions is equivalent to a serial schedule with the same transactions such that all conflicting operations in the serial and parallel schedules occur in the same chronological order.

Two-phase locking (2PL) enforces conflict-serializability [6]. In two-phase locking, locks can only be acquired in a growing phase and released in a shrinking phase. With this approach, locks cannot be acquired after any lock has been released. While there are more refined notions of serializability than 2PL, they are expensive to maintain and do not offer much additional flexibility [6]. Ordering-free regions employ an algorithm based on 2PL.

Conflicts that violate the serializability of a parallel execution can either be detected or avoided. Prior work on strong memory consistency models [8,47,60] detects violations of region serializability to report data races to the programmer. With this fail-on-conflict approach, the application exits and reports the violation of serializability to the programmer when one occurs. Conflict-serializability avoids conflicts rather than detecting them by serializing regions before a

conflict occurs. Serializing the execution to avoid a conflict can automatically prevent a bug, but too much serialization can decrease performance.

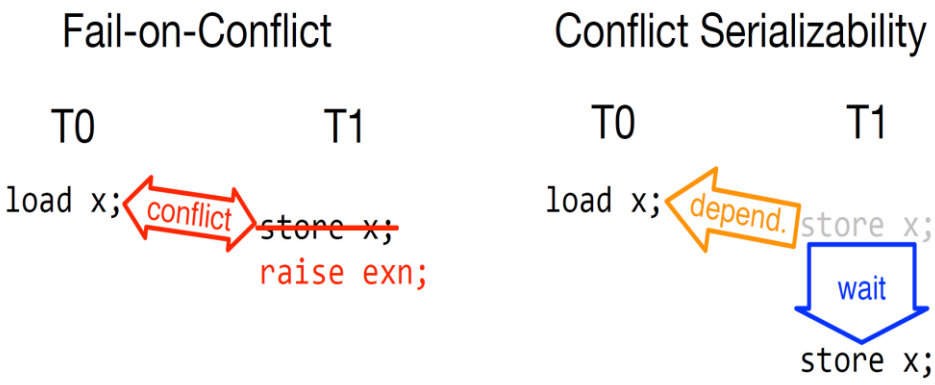

Figure 2.3: Demonstration of fail-on-conflict semantics and conflict serializability

Figure 2.3 demonstrates both a fail-on-conflict approach and conflict serializability. In the fail-on-conflict approach, an exception is raised when one region conflicts with another. The fail-on-conflict approach brings the programmer into the loop by indicating that a conflict exists. With conflict serializability, regions that would otherwise conflict can be serialized instead, preventing the conflict at runtime. Conflict serializability requires knowing how long the threads must be serialized to prevent the conflict from affecting the program's required semantics. Ordering-free regions rely on an algorithm based on 2PL to implement conflict serializability when possible. In the event that conflict serializability fails, ordering-free regions fall back to the fail-on-conflict approach and ask the programmer for help.

3 ORDERING-FREE REGIONS

This dissertation introduces a new strong memory model based on ordering-free regions (OFRs). Ordering-free regions function similarly to other strong memory consistency models such as SFRs, RFRs, and IFRs. Ordering-free regions rely on conflict serializability as much as possible and only fail-on-conflict when necessary. Unlike these prior models, ordering-free regions do not rely on programmer-defined lock acquires and releases to define region boundaries.

DEFINITION 3.0.1: Ordering-Free Region

The **ordering-free region** for a memory access to a location extends from the previous ordering construct to the next ordering construct, chronologically, in the dynamic execution of the program.

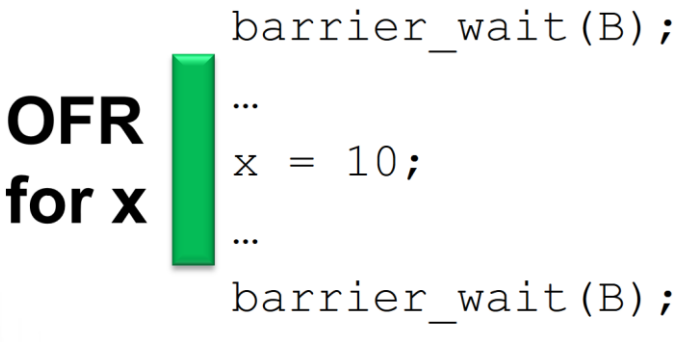

Figure 3.1: The ordering-free region for a variable

In the example above, the ordering-free region for x extends from the barrier wait before the write to x until the barrier wait after the write to x.

DEFINITION 3.0.2: Ordering Constructs

Ordering constructs are defined as barrier waits, condition variable waits, thread exits, and thread joins. These constructs all indicate that the waiting thread needs one or more other threads to make progress before it continues its execution.

Ordering-free regions conservatively approximate the atomicity that a parallel program might require. Although it is theoretically possible that a program could require atomicity that spans an ordering construct, it is difficult in practice to find an application with this requirement.

An initial study of 780K lines of parallel code from the PARSEC benchmark suite [7], memcached, apache, and pbzip2 found that none of the applications studied required atomicity over an ordering construct on any of the tested inputs. More generally, requiring atomicity across an ordering construct may lead to a deadlock in the case that thread T_1 is waiting for thread T_2 to perform some action, but thread T_1 also holds a lock that thread T_2 requires. Thus, ordering constructs seem to be a reasonable delimiter for the atomicity required by most parallel programs.

DEFINITION 3.0.3: OFR Serializability

An execution is OFR serializable if the dynamic, parallel execution of ordering-free regions is conflict serializable.

To enforce OFR serializability on a parallel execution, the accesses to all memory locations must occur atomically between ordering constructs. To this end, each memory location x is associated with a lock L_x. Before each memory access to x by a thread T, T acquires L_x if T does not already hold L_x. At the end of an OFR, when T encounters an ordering construct – a fork, join, wait, or barrier – T releases all the locks it holds. If T is ever unable to acquire a lock, L_x, then some other thread U must have accessed x in U's current OFR. T's inability to acquire L_x indicates a memory conflict between T and U. Prior consistency models raise an exception on T's access to x because of the memory conflict, but OFR serializability instead tracks a dependence from T to U and waits until U releases L_x, avoiding unnecessary exceptions on conflicts that do not compromise serializability.

This algorithm constructs the ordering-free region for each variable as shown in Figure 3.1. If the thread T is able to acquire the lock L_x within an OFR, no other thread has made a conflicting access within that OFR. Otherwise, the lock acquire of L_x by T would fail. After T acquires L_x, no other thread can acquire the lock until T releases it at the barrier_wait(). Therefore, T has atomicity over x between the two calls to barrier_wait(), which is what is required by ordering-free region serializability.

This simple algorithm does not account for read-sharing, which is common in parallel applications. In order to support read-sharing, the lock L_x that protects each memory location x is a reader-writer lock. A reader-writer lock permits multiple threads to acquire a read lock concurrently but only a single thread can hold an exclusive write lock. A reader-writer lock only permits an exclusive writer if there are no concurrent readers. With reader-writer locks, a dependence between threads only exists when the accesses by the threads to the location x conflict, meaning that one of the accesses must be a write. This use of reader-writer locks (instead of mutex locks) increases parallelism by allowing read-sharing of data, which is crucial for performance and scalability.

DEFINITION 3.0.4: OFR Exceptions

An OFR exception occurs when a cyclic dependence exists between two threads within concurrent OFRs. An OFR exception indicates that the two OFRs are not OFR serializable.

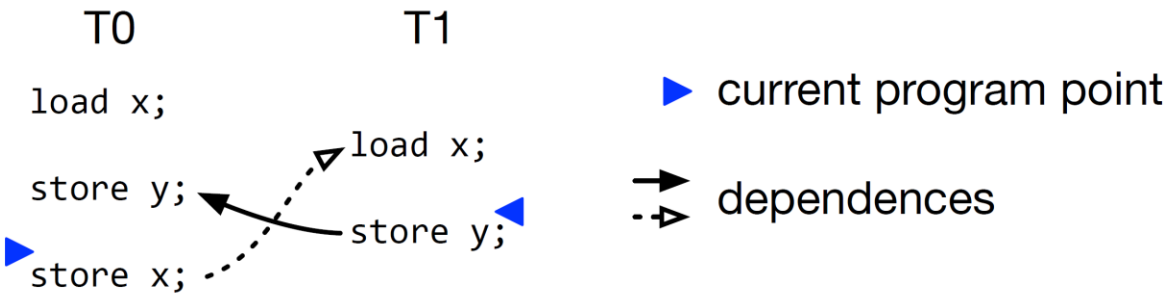

Figure 3.2: A scenario in which an OFR exception will be raised

OFR serializability triggers an OFR exception when executing OFRs have at least two conflicts and the conflicts form a cycle in the conflict graph [6]. An OFR exception indicates that the program permitted an unserializable execution of its regions and that its OFRs must be divided into smaller atomic regions to permit additional region interleavings. An OFR exception suggests how to avoid the same exception in future executions. Figure 3.2 demonstrates two OFRs that are not OFR serializable and raise on OFR exception when executed concurrently. In this example, T0 acquires a read lock on L_x when it loads the value of x. T1 then acquires a read

lock on L_x when it loads the value of x, causing L_x to be in a shared read state. T0 then acquires a write lock on L_y and stores a new value to y. T1 attempts to acquire a write lock on L_y but is unable to because T0 already holds a write lock on L_y. Finally, T0 attempts to acquire a write lock on L_x but is unable to because L_x is in a shared read state, which conflicts with T0's attempt to acquire a write lock.

An OFR exception indicates that the atomicity enforced by OFR serializability is too strong for the program. The atomicity must be relaxed to prevent the exception in future executions. Examining the code in Figure 3.2, the exception can be prevented by either (1) releasing the lock on y in T1, (2) releasing the lock on x in T0, (3) releasing both locks, or (4) ensuring that x and y are updated together by changing x to use mutex locking or altering the order of stores in T0. The choice of how to prevent the exception in future executions is made by the programmer using a small annotation API described in the next section. The programmer must examine the code that caused the exception and determine how much atomicity is required for each of the memory locations involved.

3.1 ORDERING-FREE REGIONS API

Programming with ordering-free regions requires a small API that allows programmers to refine a program's region specification and optimize performance.

A Release() annotation refines a program's region specification, sub-dividing a region into smaller regions, e.g., to eliminate an exception. The basic Release() annotation explicitly releases a specified location's lock and we include "syntactic sugar" API calls that batch release locks on objects and arrays. ReleaseObject() releases the locks on all fields of an object, and ReleaseArray() releases the locks on all elements of an array.

A RequireMutex() annotation associates a mutex lock with a memory location, rather than a reader-writer lock to avoid upgrade cycles. A post-dominator compiler analysis can often identify cases that might require a mutex and avoid the need for the programmer to manually add

RequireMutex() annotations. However, due to the need to be conservative, the programmer does occasionally need to add these annotations manually.

An EndOFR() annotation ends an ordering-free region before execution reaches an ordering operation. The EndOFR() annotation can be used when the atomicity requirements of thread allow the bulk release of all locks held by that thread. For example, pipeline parallel applications tend to perform operations on an object and then pass that object to the next stage of the pipeline for further processing. When this hand-off between pipeline stages occurs, ownership of the object transfers completely from the earlier pipeline stage to the later pipeline stage. Therefore, the earlier pipeline stage can release all of its locks on the object to allow those locks to be acquired by the later stages of the pipeline. These locks could be individually released using Release() annotations, but the EndOFR() annotation can instead be used to batch release all of the locks instead.

A ContinueOFR() annotation specifies that its containing region should not end at the next ordering operation executed. ContinueOFR() would be useful when a program requires atomicity coarser than an OFR (although we never encountered such a situation). ContinueOFR() can also be useful to improve performance by avoiding frequent lock releases at region boundaries. For example, in canneal, not releasing locks at a barrier does not affect correctness because Release() annotations release all locks that cause OFR exceptions.

3.2 PROOF OF CORRECTNESS

This section shows formally that OFR serializability enforces conflict serializability. The following definitions are used to support the arguments and can be assumed from this implementation of ordering-free regions.

DEFINITION 3.2.1: OFR Locking

Ordering-free regions associate a single reader-writer lock L_x with each memory location x.

DEFINITION 3.2.2: OFR Acquire

No access to a memory location x by a thread T proceeds without first holding the location's lock in the correct mode for the access (i.e., read vs. write mode).

DEFINITION 3.2.3: OFR Release

No lock L_x is released by a thread until the end of the thread's current ordering-free region.

DEFINITION 3.2.4: OFR Deadlock Detection

OFR serializability performs precise cyclic lock waiting (i.e. deadlock) detection.

THEOREM 3.2.5: Exception-Free Serializability

If an execution of OFRs is free of OFR exceptions, then the execution is conflict serializable.

Proof by contradiction. Assume that an OFR exception-free execution was not conflict serializable. This proof assumes two ordering-free regions, but the argument generalizes to arbitrary-length conflict cycles.

If an execution is not conflict serializable, then the definition of conflict serializability for OFRs implies that there is a set of conflicts between OFRs that form a cycle in the conflict graph. Consider O_i and O_j, two OFRs from different threads that both access a location x leading to a conflict. By (3.2.1) and (3.2.2), OFR serializability ensures O_i and O_j acquire the lock for x in the correct mode before each access. By the definition of a conflict, one (or both) of O_i or O_j is writing x and by (3.2.2) the writer(s) must hold the lock in write mode before the write. By (3.2.3),

whichever region successfully acquired x's lock continues executing, holding the lock until its region ends. The region that did not acquire x's lock waits until the lock is released.

By the assumption that the execution is not conflict serializable, there is another conflict between O_i and O_j on another arbitrary location y. As with x, one region acquires y's lock and one waits. If the same region acquires y's lock as acquired x's lock, then that region completes and releases both locks; in the absence of other conflicts, the regions serialize, violating the assumption that the execution is not conflict serializable. If, instead, the region that acquires y's lock was not the one that acquired x's lock, the regions deadlock, each waiting for the other to release its lock. By (3.2.4), OFR serializability precisely detects this deadlock and reports an exception, violating the assumption that the execution was exception-free. Thus, the assumption leads to a contradiction, proving that an exception-free execution is conflict serializable.

THEOREM 3.2.6: Unserializability of OFR Exceptions

If an execution triggers an OFR exception, then the execution is not conflict serializable.

Proof. As above, this proof covers the two OFR case, but the argument generalizes to arbitrary conflicts. Assume an OFR exception has been generated. By (3.2.4), an OFR exception corresponds to OFR serializability detecting that two regions are mutually waiting for one another to release locks: region O_i waits for O_j to release a lock on location x and O_j waits for O_i to release a lock on location y. By (3.2.2), if a region proceeds it will next immediately access the location protected by the lock it waits for. At least one region's imminent access to each variable is a write, because pairs of reads would be allowed to execute concurrently, by (3.2.1). Consequently, the regions' impending accesses form two conflicts, one on x and one on y. Furthermore, by (3.2.1) and (3.2.4), because the regions cyclically wait to acquire locks, the corresponding access conflicts are also cyclic. A case-based analysis shows that an OFR exception indicates a violation of conflict serializability. By the definition of conflict serializability [6], a conflict graph cycle indicates a violation of conflict serializability.

Case 1. In the single variable case, an OFR exception may be triggered by O_i and O_j acquiring a lock on x in read-mode and then attempting to upgrade to write-mode. This violates conflict serializability because both threads are attempting to both read and write to the same location x.

Case 2. With two variables, x and y, there are multiple, similar cases that may result in an OFR exception. In all of these cases, O_i writes to either x or y, and O_j writes to the opposite location. If both O_i and O_j write to the same location and do not incur an upgrade cycle, as in Case 1, the OFRs will serialize on one of the writes. Therefore, the OFRs must write to opposite locations. By the same logic, each OFR must read from the location that it does not write to because writing to both locations would also cause the OFRs to serialize. These conflicting read-write pairs violate conflict serializability.

Note that the correctness proof is sound and complete – an execution free of exceptions is conflict serializable and an exception indicates that an execution is not conflict serializable. By contrast, prior work [8, 47, 66] provided a weaker correctness argument – an exception-free execution is serializable, but an exception corresponds only to a conflict, indicating a data-race, but not a violation of conflict serializability.

3.3 QUANTIFYING ATOMICITY

While OFRs have intuitive benefits over finer-grained atomic regions, it may be the case that these advantages do not materialize due to the structure of real programs. To quantify atomicity, it is necessary to consider both the length and width of regions as described in Section 2.3.1.

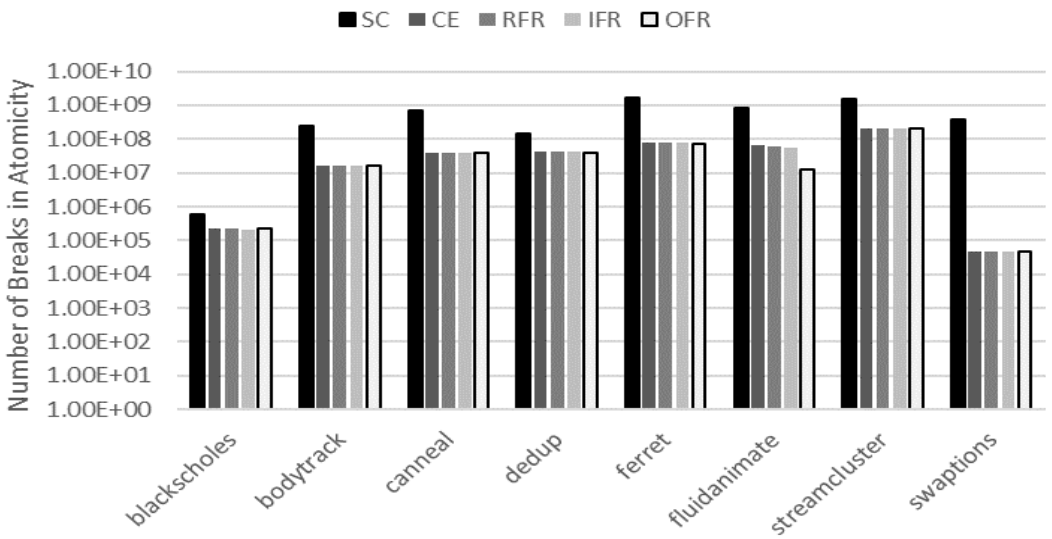

Figure 3.3: Number of breaks in atomicity for each model

Figure 3.3 shows the number of times that atomicity is broken in each programming model. Sequential consistency (SC) logically breaks atomicity after each memory access. Conflict Exceptions (CE, representative of SFRs) breaks atomicity at each lock acquire and release. Release-free regions (RFR) break atomicity at each lock release. IFRs extend CE and RFR to not break atomicity if the memory location will be accessed in the next region. OFRs break atomicity at each ordering construct. As shown, CE, RFR, IFR, and OFR break atomicity a similar number of times when considering only the length of regions. Thus, it is necessary to consider the width of regions to properly quantify the amount of atomicity provided by each of these programming models.

By considering both the length and width of regions, the atomicity of each model can be quantified by examining how many variables (width) are protected by a region of a given size

(length). The following CDF plots quantify the atomicity of three parallel applications: blackscholes, ferret, and fluidanimate. The length and width of all regions R were recorded. The length of the region R is the number of dynamic instructions included in that region. The width of the region R is the number of memory locations accessed within that region. These plots detail how many regions have a width of w as a fraction of all regions as a cumulative distribution function. The width metric captures the ability of a consistency model to enforce atomicity across memory locations, reducing the probability of multi-variable atomicity violations. In these CDFs, curves that rise more gently indicate greater atomicity, as there are a substantial proportion of wide regions and a small proportion of low-width (narrow) regions. Curves that rise steeply indicate that most regions are narrow.

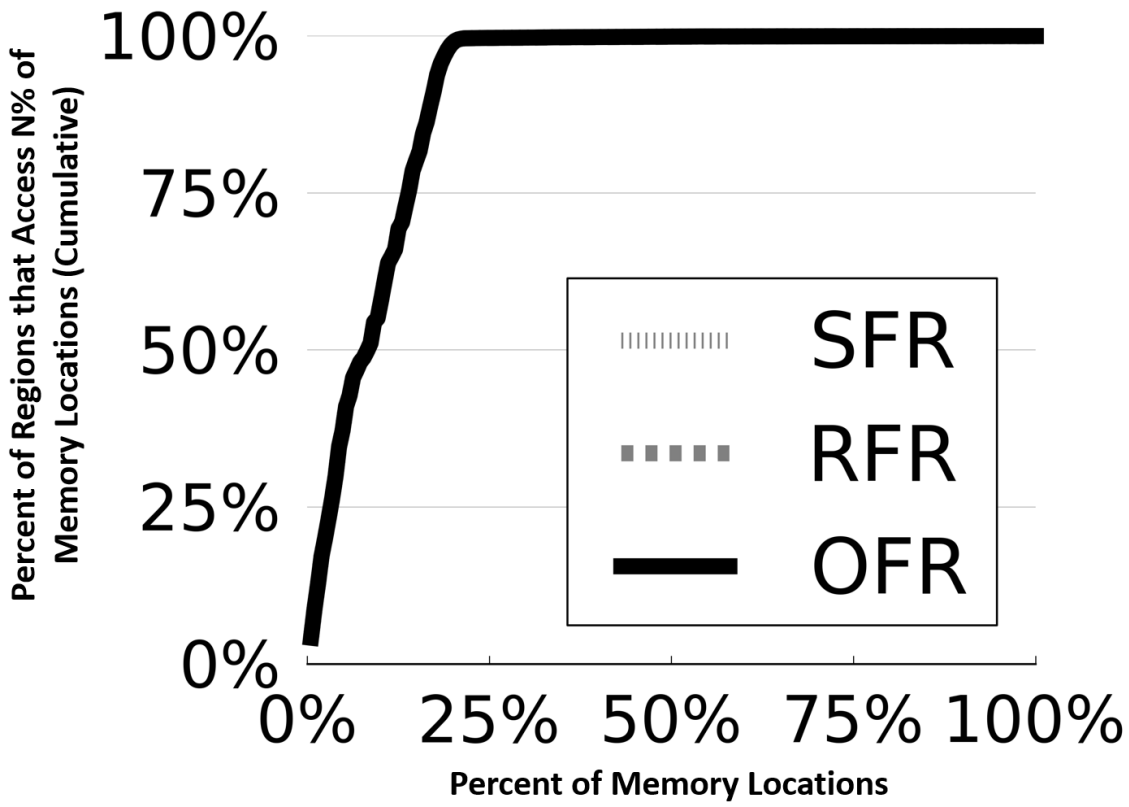

Figure 3.4: CDF plot showing the length and width of regions in blackscholes

Figure 3.4 quantifies the amount of atomicity provided by SFRs, RFRs, and OFRs on blackscholes. On this application, the amount of atomicity provided is the same for all three models because blackscholes is an embarrassingly parallel application, meaning that it requires no lock acquires or releases. Therefore, the three models use the same regions.

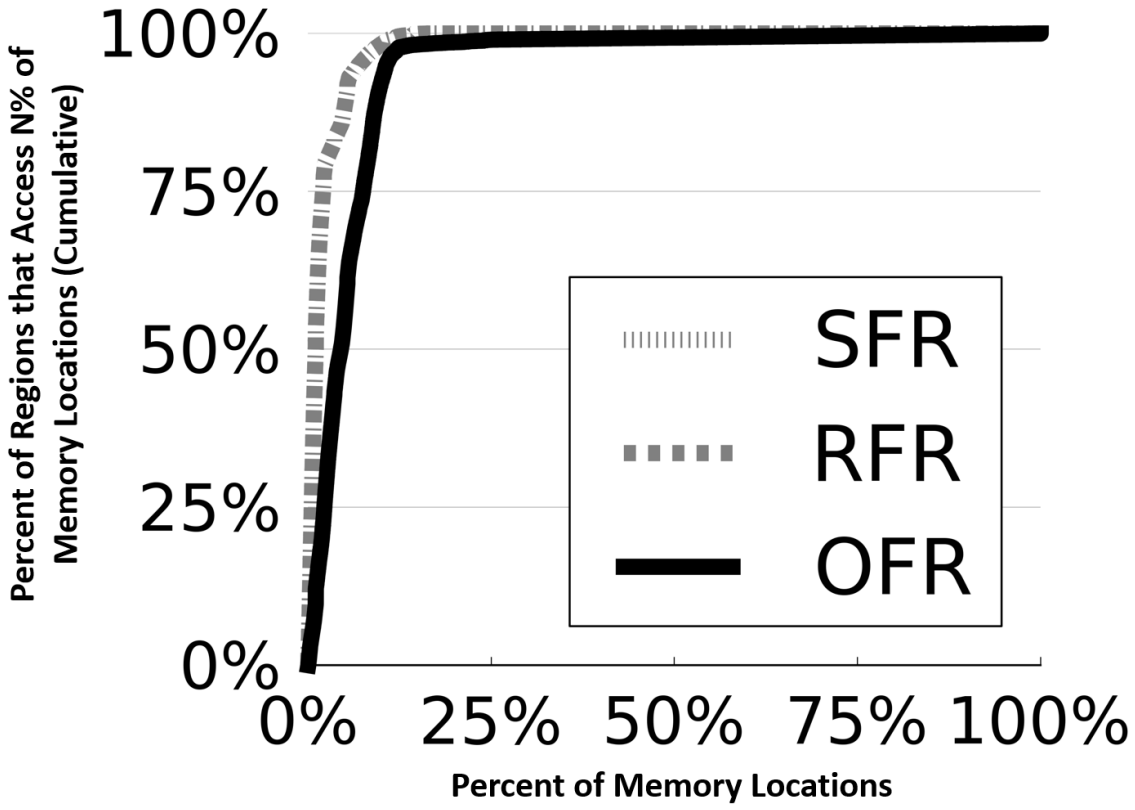

Figure 3.5: CDF plot showing the length and width of regions in ferret

Figure 3.5 quantifies the amount of atomicity provided by SFRs, RFRs, and OFRs on ferret. ferret is a pipeline parallel application that passes objects from one stage of the pipeline to the next. Each stage of the pipeline executes in parallel and performs disjoint work from all other pipeline stages. Objects are passed from one stage to the next using a queue that employs ordering constructs to prevent threads from dequeuing from an empty queue or enqueuing into a full queue. Most regions applied by SFRs and RFRs are quite small in ferret. The regions used by OFRs tend to be slightly larger, both in length and in width.

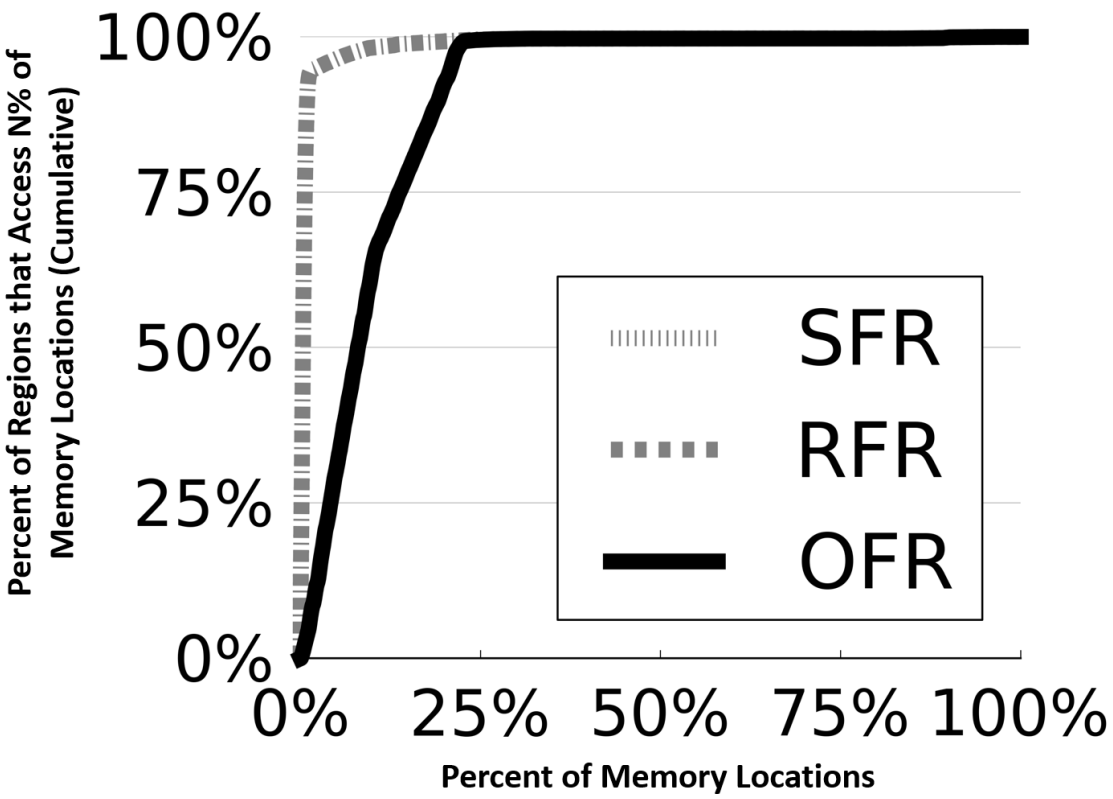

Figure 3.6: CDF plot showing the length and width of regions in fluidanimate

Figure 3.6 quantifies the amount of atomicity provided by SFRs, RFRs, and OFRs in fluidanimate. fluidanimate uses complex, fine-grained locking that causes frequent region boundaries for SFRs and RFRs while OFRs are considerably wider due to infrequent barrier synchronization.

Overall, the theoretical benefits of OFRs manifest more clearly in programs with more complicated parallel structure, which are arguably the programs likeliest to suffer from concurrency bugs. From this analysis, there appears to be no significant difference in atomicity between SFRs and RFRs, suggesting that the practical benefits of moving from SFRs to coarser-grained RFRs are limited. However, OFRs clearly provide longer and wider regions than both SFRs and RFRs, indicating that they provide increased atomicity for parallel programs.

4 IMPLEMENTING ORDERING-FREE REGIONS

The following section discusses the tradeoffs and decisions in implementing a system that enforces atomicity based on ordering-free regions. These implementations feature a few common components that are required to implement OFR serializability. Each implementation requires a lock design that permits high-performance execution and a shadowspace design that allows efficient retrieval of locks as needed. Similarly, each implementation requires the common API of runtime functions for managing OFR serializability. A distributed deadlock detector is used to detect and report OFR exceptions, and some amount of allocator and compiler support may be required to identify allocations, loads, stores, and other functions of interest.

4.1 Overview

This section examines and discusses three runtime systems that implement OFR serializability. MAMA [20] was implemented as part of our initial investigation into the applicability of OFR serializability to parallel applications. This software-only runtime system leveraged the RoadRunner [32] framework to associate locks with data and provided insights into how future implementations could be made more efficient. From our experiences with the MAMA prototype, we believed that hardware support would be necessary for a reasonably efficient implementation of OFR serializability. We also decided to implement future systems in C and C++ instead of Java due to our desire to use LLVM for compiler optimizations. ORCA [23] introduced hardware support for OFR serializability. We used the Pin [50] dynamic binary instrumentation tool to apply OFR serializability to C and C++ applications in a simulated environment. Although ORCA enforces OFR serializability at low performance overheads, the need for new hardware support was undesirable. SOFRITAS [22] provides OFR serializability using a software-only runtime system and simple compiler instrumentation. SOFRITAS builds upon both MAMA and ORCA to provide OFR serializability efficiently in a software system.

4.2 Locks

Implementing ordering-free regions using locks requires a highly efficient lock design. As described in Section 3, the system that enforces OFR serializability requires that a thread holds a lock in the correct mode on each access to a memory location protected by that lock. This functionality does not necessarily have to be implemented using locks, but the systems designed for this dissertation all rely on locks in their implementations. These locks must be efficient enough to limit the overheads of associating a lock with every memory location and acquiring those locks on every memory access.

Table 4.1: Characterization of memory accesses and lock acquires

Benchmark	Memory Accesses	Reads (%)	Writes (%)	Acquires (%)
blackscholes	7.1 Billion	85.13%	14.87%	2.81%
bodytrack	95.7 Billion	93.23%	6.77%	3.39%
canneal	21.6 Billion	96.07%	3.93%	23.49%
dedup	3.1 Billion	98.97%	1.03%	28.79%
ferret	187.9 Billion	89.43%	10.57%	5.91%
fluidanimate	228.7 Billion	88.00%	12.00%	20.17%
streamcluster	428.3 Billion	99.57%	0.43%	51.46%
swaptions	196.0 Billion	76.58%	23.42%	0.00%
gups	500 Million	40.00%	60.00%	80.00%
kmeans	3.2 Billion	91.73%	8.27%	15.12%
pagerank	1.2 Billion	93.61%	6.39%	25.30%
histogram	3.8 Billion	62.50%	37.50%	0.00%
kmeans	14.8 Billion	99.80%	0.20%	1.13%
linear_regression	4.9 Thousand	17.57%	82.43%	2.46%
matrix_multiply	2.0 Billion	99.95%	0.05%	0.05%
pca	16.1 Billion	99.80%	0.20%	0.42%
reverse_index	2.1 Billion	99.56%	0.44%	49.26%
string_match	1.4 Billion	34.55%	65.45%	0.00%
word_count	740.5 Million	97.46%	2.54%	0.49%

Table 4.1 characterizes the memory accesses performed by 19 parallel applications. These access characteristics heavily influenced the lock designs used to implement OFR serializability. The *Memory Accesses* column lists the total number of memory accesses

performed by each application. The number of accesses is abnormally low for linear_regression because most of the memory accesses performed by this application are to a read-only mapped memory region holding file input, which can be safely ignored by the OFR algorithm considering that all accesses must be reads. The accesses shown for gups are round numbers due to the synthetic nature of the benchmark. The second two columns show the percentage of *Reads* and *Writes* over all memory accesses. Reads tend to be more common, but some application like gups and string_match do exhibit a larger number of writes than reads. For this reason, the locks used to implement OFR serializability should be biased toward reads but still allow for efficient writes. The final column (*Acquires*) shows the percentage of memory accesses that cause a lock acquire under OFR serializability. The non-acquire memory accesses simply check to see if the lock is currently held in the correct permission. Aside from a few outliers, the percentage of acquires tends to be low compared to checks. Thus, the lock design should be biased toward fast checks but still permit relatively fast acquires when possible.

Table 4.2: Characterization of lock acquires as reads or writes

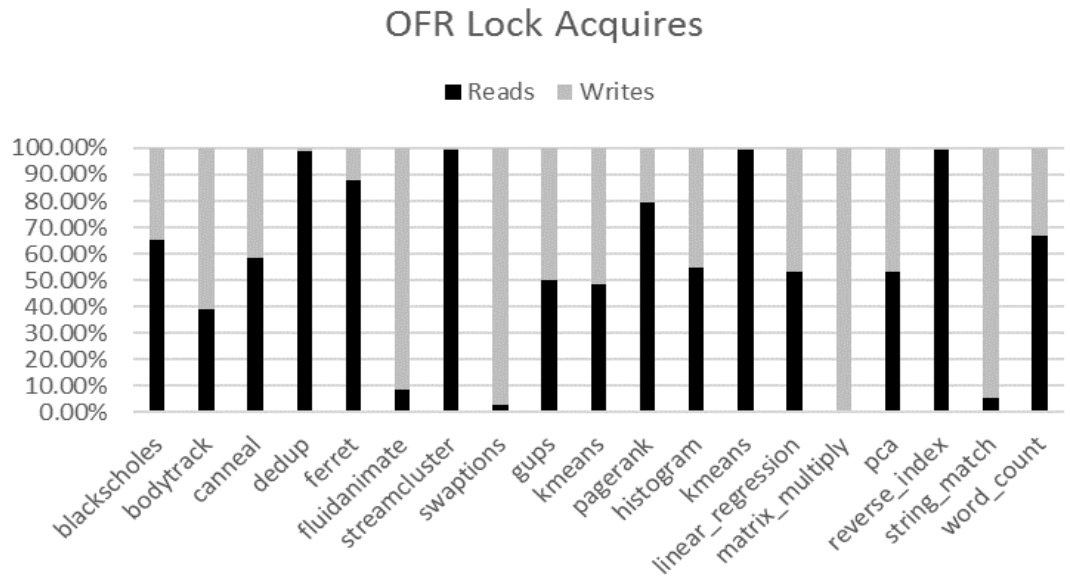

Table 4.2 characterizes the modes that locks are acquired in to enforce OFR serializability. Although the applications are more likely to favor read acquires over write

acquires, there are many applications that instead favor write acquires over read acquires. Therefore, the lock designs should not heavily bias to one type of acquire over the other. In general, the locks used to implement OFR serializability attempt to optimize for common case transitions between lock states. For example, threads may use the same variables in successive ordering-free regions. In this case, the lock should be designed to allow the thread to efficiently reacquire the locks that it held in the previous OFR.

Aside from being efficient, the locks used to implement OFR serializability must provide the features necessary to interact with the rest of the OFR runtime system. First, both reader-writer and mutex locks must be available, whether the same lock implements both or two separate lock types are used. Neither reader-writer nor mutex locks avoid strictly more deadlocks, so both types must be available to the runtime system. Second, the locks must indicate the current owners and state to allow the system to detect cycles, which in turn are forwarded to the user as exceptions. Unlike conventional locks used in multithreaded systems, OFR locks cannot simply indicate whether the lock is currently held or not. OFR locks are similar to re-entrant locks that track which thread currently holds the lock and allows a simple check rather than an acquire if the current owner attempts to acquire the lock a second time.

4.2.1 Eager or Lazy Releases

The OFR execution model discussed thus far **eagerly** releases all locks at every ordering construct, which is sufficient to guarantee 2PL serializability of OFRs but comes with a large performance tax. Eagerly releasing locks at all ordering constructs can be accomplished in a number of ways. In any implementation, logically, a list of locks held by each thread must be maintained in some way, or the lock structures must be designed to make such a list unnecessary. A naïve solution might simply maintain a per-thread list of every lock the thread has acquired. This solution would require additional overhead on every lock acquire, and eagerly releasing locks would require a list traversal to release every lock held by the thread. A per-

thread version number could also be added to each lock to identify which locks have been acquired in the current ordering-free region, but this solution would increase the size and complexity of the locks substantially (similarly to maintaining a vector clock for data-race detection.

Alternatively, locks can be released using a **lazy** policy that only releases locks when another thread attempts to acquire them. This lazy release policy holds locks across ordering constructs. Locks can still be released by threads waiting at an ordering construct: if a thread T_0 holds a lock L_x and is blocked at a barrier, another thread T_1 can **steal** L_x. T_0's OFR serializability is preserved because the OFR in which T_0 acquired L_x must have ended, as T_0 is at an ordering construct. Lazy releases do not compromise OFR atomicity, and in fact strengthen it – in the absence of steals, a variable's atomicity is preserved across multiple OFRs. A subtlety of lazy releases is that a dependence cycle may not violate 2PL serializability. Consider the case in which a thread T_0 holds a lock L_x on some location x and does not release that lock at the end of its ordering-free region O_1. If another thread T_1 similarly holds a lock L_y on location y and does not release that lock at the end of its ordering free region O'_1, T_0 and T_1 may deadlock if T_0 attempts to acquire L_y in its new OFR O_2 and T_1 attempts to acquire L_x in its new OFR O'_2. Both threads have moved on to new ordering-free regions O_2 and O'_2, but the lazy release of locks has caused a dependence cycle that would not have existed if locks had been eagerly released.

Precisely supporting lazy releases would require an implementation similar to a vector-clock, which would be unlikely to provide improved performance. In practice, it is possible to use a more targeted form of lazy releases that does not require a full vector clock. Both pipeline parallelism and barrier-based synchronization can be supported by a scalar **stage** that enables lazy releases of locks between ordering-free regions.

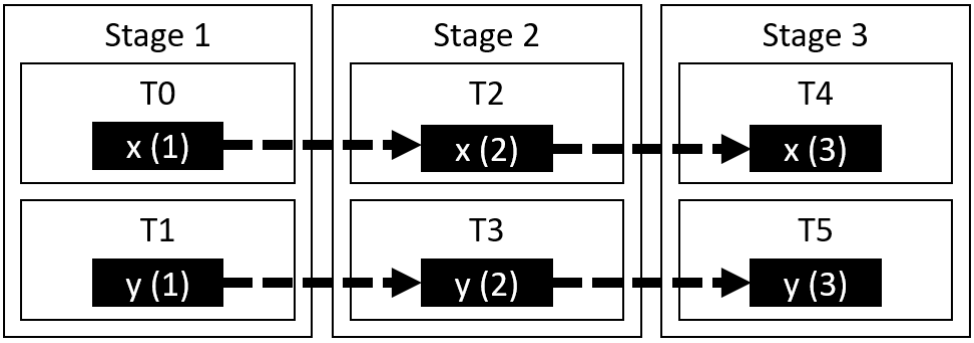

Figure 4.1: Pipeline stages using staged locking to lazily pass locks from one stage of the pipeline to the next. Stage is shown in parentheses

Figure 4.1 demonstrates how stages can be used to lazily transfer locks from one stage of a pipeline to the next. This pattern is found in applications like dedup and ferret which pass objects through a parallel pipeline. Within the same stage, locks cannot be stolen. For example, T_0 and T_1 would not be able to steal each other's locks. If T_0 and T_1 produce an OFR exception, it is a true exception and not generated due to the structure of the parallelism. T_2 and T_3 can steal locks from T_0 and T_1 because they are part of a later stage of the pipeline and often receive data that T_0 and T_1 are no longer using. Supporting pipeline parallelism with a scalar stage requires a small stage due to the limited number of pipeline stages that will practically be used in most parallel applications. The programmer must indicate via an annotation which threads belong to which pipeline stages. Staging annotations are untrusted and are verified at runtime: an incorrect staging annotation that contradicts the program's sharing patterns will trigger an OFR exception to support straightforward debugging. However, pipeline parallelism is not the only type of parallelism that can be aptly supported by a scalar stage.

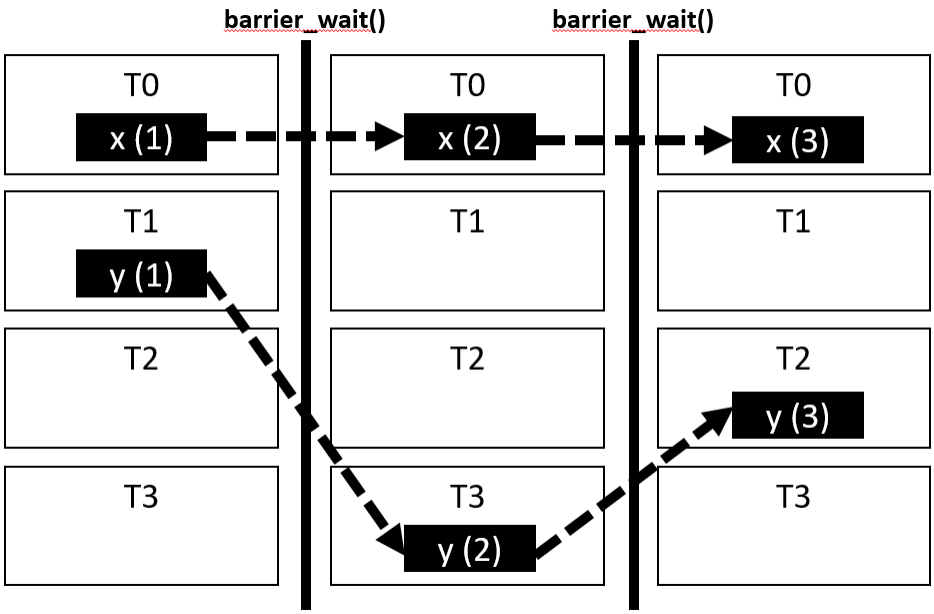

Figure 4.2: Stages of a barrier-based application

Parallel applications often use barriers to delimit phases of the parallel algorithm. Such applications also lend themselves to the use of a scalar stage for lazy lock releases. Figure 4.2 demonstrates how locks can be passed from one stage of a barrier-based application to the next. As shown, some memory locations (*e.g. x*) may be locked by the same thread in subsequent stages. In these cases, the stage is updated when the lock is tested for proper permissions. In a barrier-based application, a thread may steal a lock from another thread if the current stage of execution is greater than the scalar stage indicated by the lock. A thread can steal locks held by a thread in a previous stage, but not from its own stage or future stages. This use of stages assumes that all threads that are sharing data participate in the same barriers and therefore have matching stages. Unlike pipeline stages, barrier stages can be automatically identified at runtime (i.e. as each barrier_wait() occurs).

4.2.2 Debugging Metadata and Annotation Suggestions

To apply OFR serializability to a parallel application, the programmer must properly apply annotations from the OFR API to the code. A runtime system can assist the programmer with this task by providing suggestions based on the behavior of locks. However, the runtime system must use additional metadata to provide useful suggestions, and therefore this functionality is generally used for debugging and not during production.

When a dependence cycle occurs, the execution of the two or more threads involved will stop precisely at the lines of code that complete the cycle. Although the information found on those lines of code may be useful to the programmer, it may not identify the root cause of the dependence cycle. To locate the root cause, the programmer needs to know where the data associated with each lock was last accessed by that lock's owner. In debugging mode, locks track the last read and last write source-code location for each memory location. This information can be represented as an integer that maps to a source-code file and line number. This information is updated on each read or write to a memory location. When a dependence cycle is detected, the runtime system will suggest that the programmer place a Release() annotation at either the last read or last write line of source-code, depending on structure of the dependence cycle. The runtime system can also suggest RequireMutex() annotations in the case of read-to-write upgrade dependence cycles.

The behavior of the runtime system can also provide useful suggestions to the programmer to improve the performance of the application under OFR serializability. By examining lock contention, the runtime system can identify instances in which the application serializes on accesses to some memory location and suggest that the programmer add a Release() annotation to avoid unnecessary serialization. The runtime system can also identify frequent lock reacquires, which may indicate that a ContinueOFR() annotation can be applied. These performance suggestions should be carefully considered by the programmer so to not compromise safety for performance.

4.3 Shadowspace

Efficiently checking and acquiring locks in a system that provides OFR serializability requires an efficient mapping from each memory location to its associated lock. Locks are stored in a **shadowspace** that corresponds to the memory allocated by a parallel application. Multiple design choices can influence the runtime and space overheads of retrieving locks, including the granularity of mapping, the flexibility of mapping, and the use of a monolithic or distributed mapping.

4.3.1 Mapping Granularity

The mapping granularity of the shadowspace affects both the efficiency and the usability of ordering-free regions. A coarse mapping may provide better performance, but a coarse mapping may also cause false OFR exceptions to be generated.

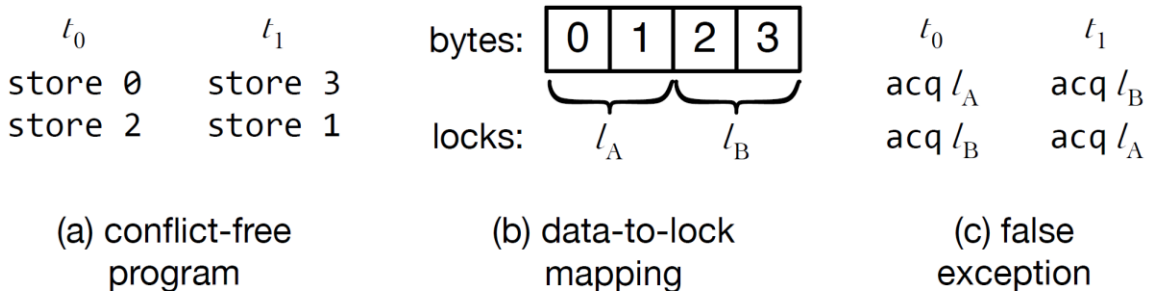

Figure 4.3: Example demonstrating how a coarse mapping leads to a false OFR exception

Figure 4.3 shows how a coarse mapping can lead to a false exception. The example assumes a mapping from each set of 2 bytes to a lock. T_0 accesses byte 0 and then byte 2 while T_1 accesses byte 3 and then byte 1. Due to the lock mapping, this access pattern causes T_0 to acquire L_A and then attempt to acquire L_B while T_1 acquires L_B and then attempts to acquire L_A, leading to a cycle. With a byte-to-lock mapping, there would be no cycle in the lock acquires because the bytes accessed by T_0 and T_1 do not overlap.

In C and C++, there is no mapping provided from bytes of data to their structure. By default, the runtime system does not know the structure of objects in memory. Thus, a straightforward way to map data to locks in C and C++ is to choose a fixed mapping from some number of bytes to a lock. Many parallel applications do not access data at byte granularity, but some do. For the applications that only access data at word (4-byte) granularity, a shadowspace mapping from words to locks would be appropriate to save space. However, for the applications that access data at byte granularity, a byte to lock mapping is necessary to avoid false exceptions. The evaluation of the systems used to enforce OFR serializability quantifies the tradeoff between byte and word mappings.

Languages like Java provide details on the structure of objects in memory that are available to a runtime system [32]. By default, Java maintains mappings of bytes and words to the fields and elements that comprise objects and arrays. Therefore, Java permits a more coarse-grained mapping than C and C++ because the runtime system is aware of the granularity at which the application will access data. The system can still choose a coarse or fine-grained mapping for locks in Java. For objects, the system can choose to either map from fields to locks or from objects to locks, and for arrays, the system can choose to either map from array elements to locks or from arrays to locks. These mappings have the same tradeoffs as byte and word granularity mappings for C and C++.

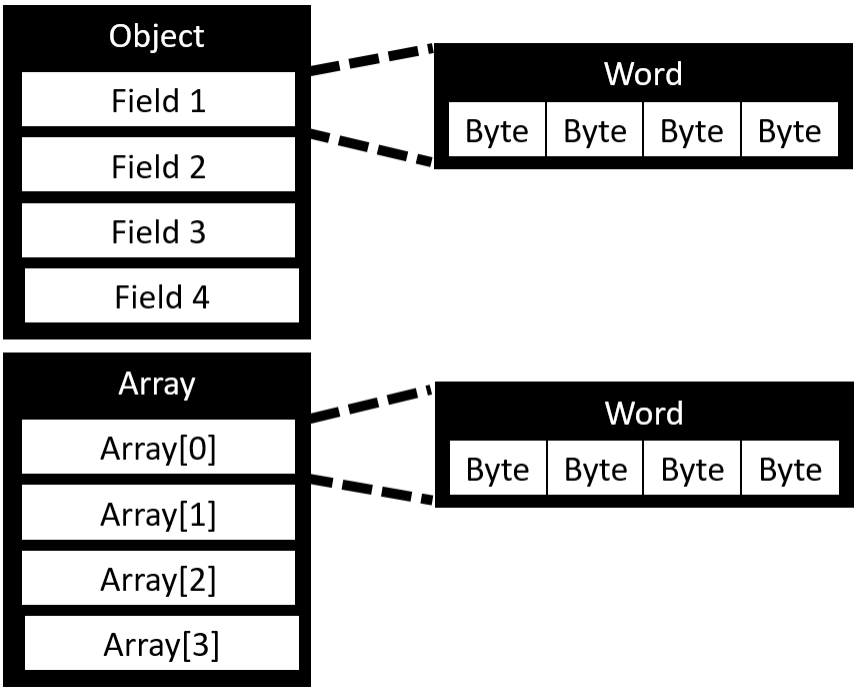

Figure 4.4: The lock shadowspace can be coarse or fine-grained

Figure 4.4 shows the granularity of mappings available in both C++ and Java. In Java, object fields and array elements can be mapped to locks because the runtime system is aware of the structure of data. In C++, only groups of bytes can be mapped to locks because the runtime system does not maintain mappings of bytes to structures. Java also permits an inline instrumentation that adds locks next to the fields or array elements that those locks protect. In C and C++, there is no guarantee that the code will not create a pointer and walk over the bytes of an object, so it is generally not safe to add locks to objects or array inline. Thus, a lock shadowspace for C and C++ applications should be disjoint from the memory being shadowed.

4.3.2 Rigid or Flexible Mapping

As mentioned in the previous section, associating locks with memory locations in C and C++ necessitates the use of a shadowspace that is disjoint from program memory. This disjoint shadowspace can be structured as either a rigid or flexible mapping from program memory to the lock shadowspace.

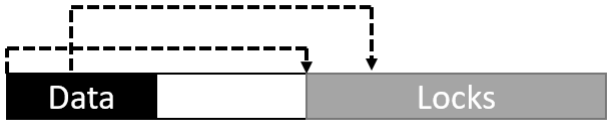

Figure 4.5: Rigid mapping from data to locks

A rigid mapping uses arithmetic to translate from each program memory address to the address of a corresponding lock. A rigid mapping requires a fixed lock format and a specific memory layout. Figure 4.5 demonstrates how a rigid mapping translates from data addresses to lock addresses. Using a fixed mapping requires few arithmetic instructions and limits the work necessary to check the state of a lock.

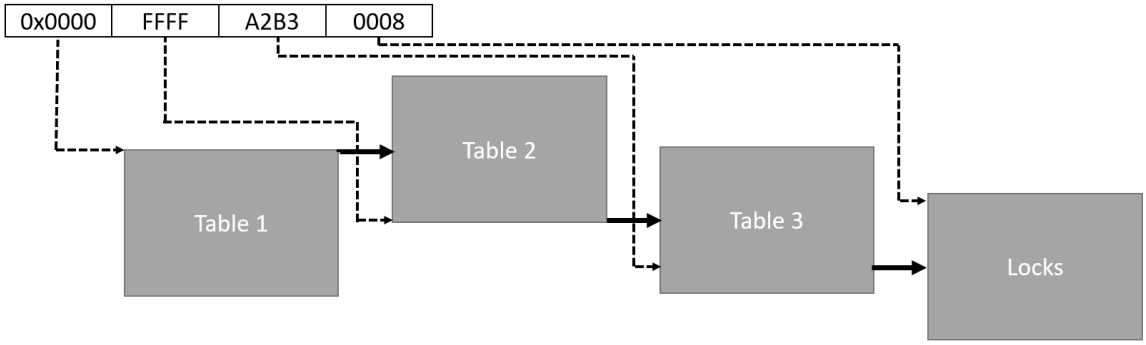

Figure 4.6: A lock trie maps data addresses to lock addresses

A more flexible translation process would admit different lock representations for different memory locations, e.g., more space-efficient mutex locks and contention-aware locks for frequently-accessed locations. To provide this flexibility, we explored an alternative shadowspace design that uses a four-level trie (like a page table) to map data to locks. Figure 4.6

demonstrates how a lock trie maps data addresses to lock addresses. The bytes of the data address are used to index tables that store pointers to lower level tables. The lowest level of tables stores the locks. This approach requires more instructions to access locks compared to the fixed mapping approach. However, the trie approach allows flexible lock formats and flexible mappings from data to locks. For example, this mapping would allow a single lock to protect a large set of data addresses. The performance evaluation of OFR serializability in later sections examines the performance tradeoffs associated with using a lock trie instead of a fixed mapping.

4.3.3 Lock and Shadowspace Design: MAMA

MAMA [20] provides OFR serializability for Java applications. As an initial investigation into OFR serializability, the locks and shadowspace used to implement MAMA were not heavily optimized but were rather meant to show that OFR serializability was a feasible model for parallel applications. MAMA's reader-writer locks consist of a single integer for the current writer's thread ID (or -1 if none exists) and a bitmap of current readers, which is implemented as an ArrayList<Integer>. MAMA relies on RoadRunner's fine-grained metadata to associate locks with data. RoadRunner associates a shadow variable with every field of an object and every element of an array. The naïve lock implementation used by MAMA suffers from high performance overheads and has been significantly improved upon by both ORCA and SOFRITAS.

4.3.4 Lock and Shadowspace Design: ORCA

ORCA [23] enforces OFR serializability efficiently using hardware support. The locks and shadowspace used for ORCA are designed to fulfill the requirements set forth in Section 4.2. However, the hardware support included with ORCA provides optimizations on top of the lock implementation in software.

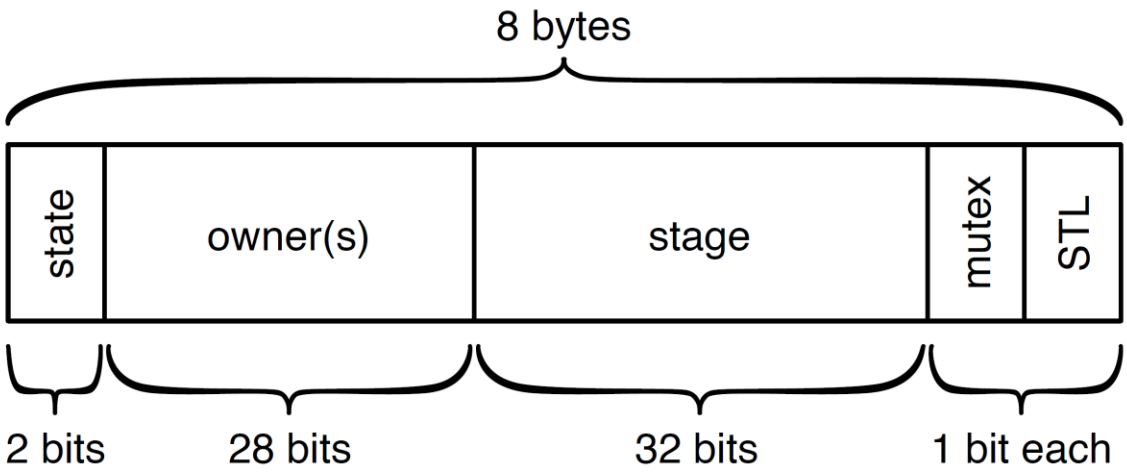

Figure 4.7: Design of ORCA's reader-writer locks

Figure 4.7 shows the lock design for ORCA's reader-writer locks. ORCA's locks are designed to allow fast ownership checks in hardware. 2 state bits list the current state of the lock, which can be unowned, exclusive reader, shared readers, or exclusive writer. This state field is loaded into a cache to facilitate fast lock checks, as will be discussed in Section 5. An additional 28 owner bits store a bitmap identifying the current owners of the lock. The 32-bit stage field is used to support lazy releases without suffering from additional OFR exceptions. The mutex bit is set when the corresponding lock must be used as a mutex instead of as a reader-writer lock. The STL bit is used to indicate that the lock was protecting an STL data structure and should be acquired and released on entry and exit from STL methods. All updates to the lock structure rely on atomic compare-and-swap operations to atomically update the metadata.

ORCA permits both fixed and flexible lock shadowspaces. The proposed hardware support relies on a fixed lock format for address translation. As an alternative, a software trie could be used to permit flexible lock mappings and formats. The flexible lock trie suffers from additional runtime performance overheads compared to using fixed hardware translation.

The ORCA lock design suffers from a few critical flaws. The hardware designed to support ORCA loads the state bits into a cache, but the state bits are distributed across multiple bytes of memory. To load 64 locks worth of state into the cache, the hardware must perform 64

loads of data. The owner field limits the total number of threads that can be used by an application to 28, which may not be enough threads. For example, ferret requires a 16-byte lock to allow for a larger owner field because it uses 65 total threads. Expanding the owner field to accommodate larger thread counts would quickly become infeasible. The lock design used by ORCA also necessitates a lazy release policy for high-performance because clearing all ORCA locks would require iterating over the lock shadow-space and zeroing out all owned locks. Although the lazy release policy yields performance benefits, it weakens runtime properties of an ORCA execution because an OFR exception can be thrown due to lazy releases rather than due to a violation of OFR serializability. Although lazy releases can provide some amount of runtime performance optimization, the stage field requires 4 bytes of memory and could suffer from wraparound issues due to integer overflow.

4.3.5 Lock and Shadowspace Design: SOFRITAS

To remedy the drawbacks of ORCA's locks, SOFRITAS [22] adopted a new lock design in order to avoid the need for a lazy release policy. The SOFRITAS lock design specifically targets an eager release policy that clears all locks currently owned by a thread at each ordering construct.

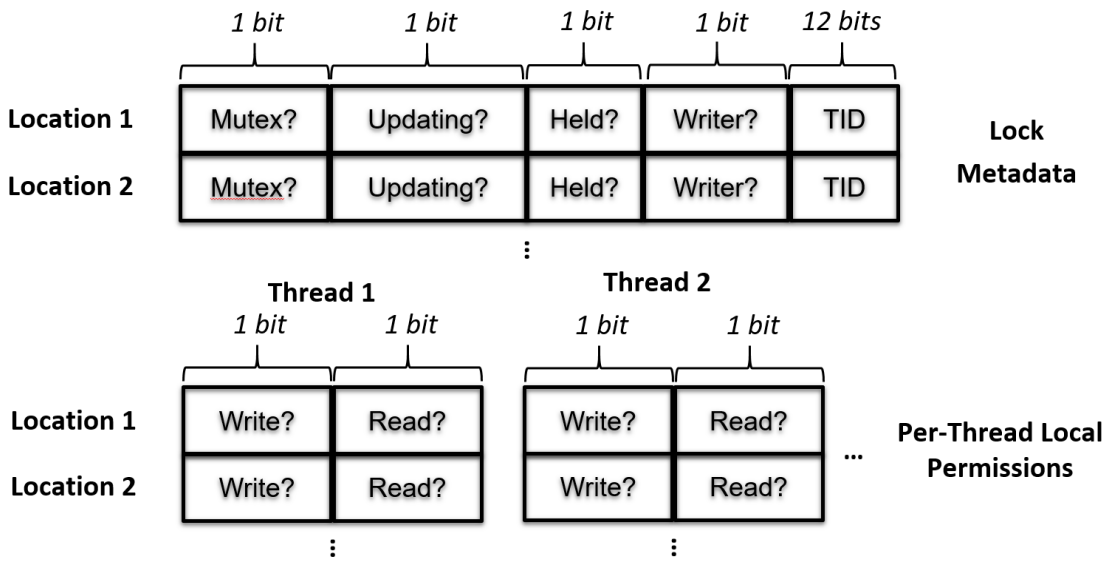

Figure 4.8: SOFRITAS distributed lock implementation

SOFRITAS's locks are designed to support efficient lock ownership checks, as these checks vastly outnumber lock acquires on most programs. Figure 4.8 shows the structure of the locks used by SOFRITAS to enforce OFR serializability. Each lock is split into disjoint structures: 16 bits of global metadata and 2 bits (per-thread) of thread-local permissions. Thread-local permissions indicate whether or not a thread can currently read from or write to a location. For each location, a thread can have read, read and write, or no permissions because a thread with exclusive write access also has exclusive read access. Local permissions are only ever updated by their corresponding thread, though they may be read by remote threads. A thread T's lock ownership checks need consult only T's local permissions. Thus, thread-local metadata can be read without synchronization. The locks for adjacent memory locations map to adjacent global metadata, and to adjacent local permissions for a given thread, ensuring that spatial locality among a thread's data accesses translates to good locality for its lock accesses as well.

The mutex bit is set by RequireMutex() and ensures that a lock is always acquired with write permissions. The updating bit acts as an internal lock over the lock's state, and is held

while updating any lock state, including thread-local permissions. The updating bit avoids writer starvation as once a writer is able to set the updating bit, no new readers can arrive.

To motivate the rest of the SOFRITAS lock design, we first discuss how to enable efficient lock releases. Resetting global metadata on each lock release would require maintaining a prohibitively expensive list of every lock acquired during an OFR. Instead, only local permissions are updated on a release. This admits an efficient implementation of bulk releases via the madvise system call, using the MADV_DONTNEED flag to zero a thread's entire local permissions space. madvise has been used in prior works to efficiently save and restore state from a memory-mapped file in parallel applications [28,53]. The madvise system call is available as part of the Linux operating system. Using the MADV_DONTNEED flag with the madvise system call causes the operating system to unmap the physical pages of memory allocated to a specified address range. Subsequent reads to an unmapped page returns 0 without allocating a new physical page, and subsequent writes to an unmapped page causes the operating system to allocate a new, zeroed page of physical memory. SOFRITAS allocates memory with mmap using the MAP_ANONYMOUS flag to enable this zero-fill-on-demand behavior.

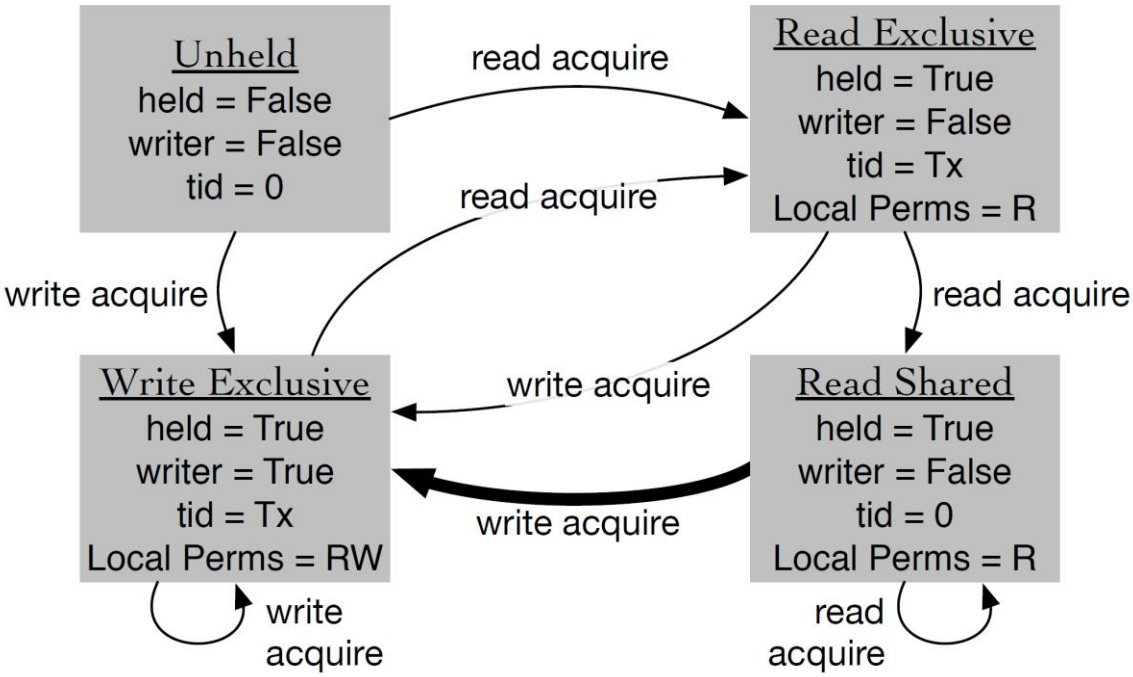

Figure 4.9: SOFRITAS lock state transitions

Global metadata can become stale in that it may reflect state that has changed due to a release operation. In fact, the global metadata will be stale unless the lock has been acquired by some thread in its current OFR. The definitive state of a lock is recorded in local permissions, and global metadata serves as a conservative summary of local permissions. Figure 4.9 details the state transitions performed by SOFRITAS locks. The held bit is set when a thread acquires the lock and remains set thereafter, allowing first-acquires to avoid checking any local permissions. The writer bit indicates that a lock is held with write permissions (otherwise it is in a read state), and the tid field identifies the exclusive writer, or reader, or identifies the lock as read-shared. Together, the writer and tid fields identify when a lock is (or was just) in an exclusive state, so an acquiring thread examines just one thread's local permissions during a state transition. Upon examining local permissions, an acquiring thread t can determine whether global metadata is stale, i.e., whether the lock is actually still held by its supposed owner. The only case where all local permissions must be consulted is for a read-shared to write-exclusive transition (heavy arrow in Figure 4.9), where the writer waits for all readers to release their locks.

4.4 Deadlock Detection

In order to detect dependencies between threads within an OFR, all implementations use a distributed deadlock detection algorithm [13] to detect conflict cycles. Only waiting threads run cycle detection, putting the work of deadlock detection off the execution's critical path. For completeness, a summary of the deadlock detection algorithm is included below.

ALGORITHM 4.3.1: Distributed Deadlock Detection

```
atomic nextK
atomic K

AcquireLock:
   Attempt to Acquire Lock
   If Lock Not Acquired:
      Snapshot(K)
      Init()
      Attempt to Acquire Lock

Init:
   If CompareAndSwap(nextK,nextK + 1):
      Detect(K)

Detect(K):
   Graph G
   For Each Thread T:
      S = ReadSnapshot(T,K)
      AddEdges(G,S)
   FindCycles(G)
   Finish(K)

Finish(K):
   Increment K
```

The distributed deadlock algorithm described in Algorithm 4.3.1 allows deadlock detection on threads without interfering with threads that are not deadlocked. For each K, one deadlocked thread will be able to perform the compare-and-swap operation on nextK and become the thread that performs deadlock detection. Each thread maintains a snapshot of its state at the K that it deadlocked at. This snapshot is valid for all K greater than the K at which the

thread snapshot was created. The use of a distributed deadlock detection algorithm prevents false OFR exceptions by ensuring that stale state is not used for deadlock detection.

Like data-races, atomicity violations, and other bugs in parallel programs, deadlocks are schedule dependent [58]. Even with many runs, it can be difficult to detect all bugs in a parallel application due to the exponential number of possible schedules. Despite best efforts, it may not be possible to detect all deadlocks (i.e. OFR exceptions) that may occur. To ameliorate this problem, we adopt a methodology for testing a wide variety of schedules. All parallel applications are tested with multiple inputs to provide greater code coverage.

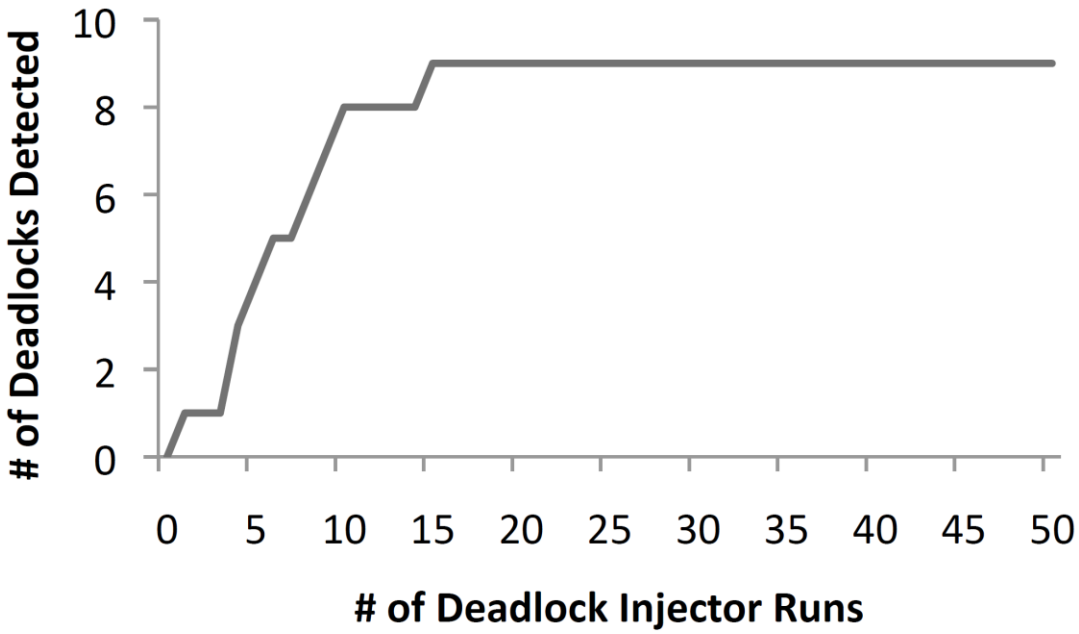

Figure 4.10: Number of OFR exceptions generated after N runs

We adopt the Lockout deadlock injection tool [41] which can increase the likelihood of deadlocks by orders of magnitude. We use Lockout to bias execution towards possible dependence cycles. Lockout represents the order in which threads acquire locks as a graph, searches for cycles, and inserts pauses in the execution whenever a lock along a cycle is acquired. These pauses increase the likelihood that a cycle will manifest. We apply Lockout to 50 executions on each application. This process did expose some exceptions, but after the fifth run

of bodytrack and the eighth run of dedup no further exceptions arose, suggesting that good schedule coverage had been achieved. Figure 4.10 shows the number of deadlocks detected after N runs for memcached. After 15 runs, no further deadlocks were detected. Although it is possible that further OFR exceptions could be lurking in these programs, these are vastly preferable to lingering data races or atomicity violations in a conventional programming model which can silently corrupt memory and cause the application to produce an incorrect result. To prioritize availability over correctness during deployment, an OFR program can be run with an OFR exception handler that logs exceptions and continues execution, similar to a conventional programming model but with the advantage of exception logs for post-mortem debugging.

4.5 Allocator Support

Implementing OFR serializability requires knowledge of the allocations performed by the parallel application - the shadowspace must mirror the program's allocations. Thus, the runtime system needs some way to hook into the memory allocations made by the application.

ORCA relied on hooks inserted by the Pin [50] dynamic binary translation tool to catch function calls to malloc, new, free, delete, and other functions used for memory allocation in C and C++. Although these hooks were sufficient to simulate the ORCA hardware support, dynamic binary translation is too slow to use in a software-only system that aims for low runtime performance overheads.

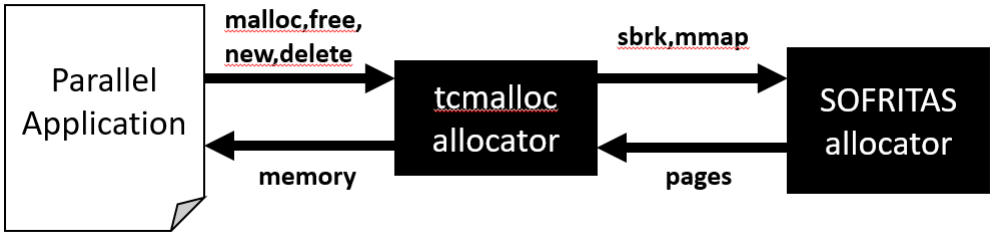

Figure 4.11: SOFRITAS allocates memory via tcmalloc

Figure 4.11 details how the SOFRITAS allocator hooks into application memory allocations. SOFRITAS uses a modified version of the tcmalloc allocator [34]. Calls to sbrk and mmap for large memory allocations are redirected to the SOFRITAS runtime from the tcmalloc allocator. The tcmalloc allocator manages free-lists and other structures required to efficiently allocate memory to a parallel application. The SOFRITAS allocator manages the larger memory allocations that the tcmalloc allocator would otherwise request from the operating system using sbrk and mmap. This design allows the SOFRITAS runtime to be made aware of all memory allocations performed by the parallel application without having to reimplement an efficient application-level memory allocator. A similar approach has been used in prior memory allocation systems for composing memory allocators [5].

4.6 Compiler Support

In order to implement OFR serializability, the runtime system needs to be able to perform a lock acquire before every load and store in the parallel application. ORCA implemented this functionality with dynamic binary instrumentation via Pin [50]. However, this functionality was not fast enough to implement a software-only system like SOFRITAS. SOFRITAS instead relied on a compiler instrumentation using LLVM to instrument loads and stores in the parallel applications. Using a compiler to instrument the code allows for optimizations that affect which loads or stores need to be instrumented. For example, simple optimizations can detect and eliminate redundant lock acquires within basic blocks. The following sections describe the SOFRITAS compiler implementation and the optimizations applied during instrumentation.

4.6.1 Basic Instrumentation

Immediately before each load or store instruction, the SOFRITAS compiler inserts calls to perform a read or write acquire, respectively. The call inserted is a simple function call that gets expanded into the code required to acquire a lock in the SOFRITAS runtime system. A small

portion of the acquire function is inlined, as ownership checks outnumber acquires for most programs. For non-aligned locations, checking lock ownership requires 9 assembly instructions; 4-byte aligned locations can be checked in 7 instructions because the needed thread-local permissions are always the low-order 2 bits and so masking is simple.

ALGORITHM 4.5.1: Inline ASM Lock Check

```
movq %r14, %rdi
subq _memoryStart(%rip), %rax
movq _locksStart(%rip), %rdx
movw (%rdx,%rax,2), %si
movzwl %si, %esi
cmpl $2, %esi
jne acquireFailed
```

Algorithm 4.5.1 shows the inline assembly for an aligned load check. If the check fails, a non-inlined call to acquire the lock is performed.

The SOFRITAS compiler also inserts hooks to identify the start and end of the main() function in the application. These hooks allow the SOFRITAS runtime to initialize its data structures before the application starts and clean them up with the application exits. The compiler replaces calls to pthread function with special version of those function calls (e.g. pthread_barrier_wait with SOFRITAS_barrier_wait) in order to identify the beginning and end of ordering-free regions in the application.

4.6.2 Optimizations

The SOFRITAS compiler applies multiple optimizations from prior work to the instrumentation required to enforce OFR serializability [4,19]. The SOFRITAS compiler elides instrumentation for locations that do not escape the stack. If a load or store has already been instrumented within a function, the compiler attempts to remove instrumentation on subsequent accesses to the same location. This optimization is conservative in a few ways. Alias analysis must determine that the

two locations must alias. Further, subsequent accesses must be instrumented if the associated lock may be released between the two accesses (e.g., by a call to pthread_condition_wait).

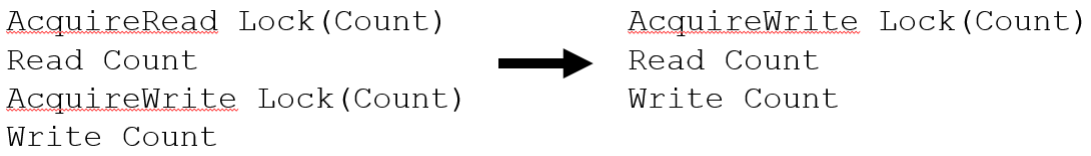

Figure 4.12: Subsequent access optimization for read-write upgrades

Many of the parallel applications studied required atomic updates on counters. With a naive instrumentation, a counter update is instrumented as both a load and a store. Using this instrumented will likely lead to an upgrade dependency cycle between multiple threads that successfully acquire a read lock on the counter and then attempt to acquire a write lock. To prevent this common scenario, any load that is post-dominated by a store is instrumented as a store instead. Figure 4.12 demonstrates how this subsequent access optimization removes a lock acquire and replaces the original AcquireRead with an AcquireWrite. This optimization often reduces the need for RequireMutex() annotations because the compiler can identify situations that may lead to an upgrade deadlock. However, the compiler analysis is conservative, and the programmer may be required to add RequireMutex() annotations when the analysis fails.

We briefly investigated the possibility of optimizing lock acquires on arrays using the SOFRITAS compiler. In many cases, a loop accesses sequential elements of an array and therefore sequentially acquires locks on that array. As a proof-of-concept, we identified “hot” code in the fluidanimate benchmark that acquired locks while looping over arrays and removed the acquires on the arrays. The single-threaded performance gains of manually removing all of the array accesses were not large, so we did not pursue this optimization further. We instead investigated the possibility of using read-only annotations as described in the next subsection.

4.6.3 Annotations

As a further optimization on one parallel application, we attempted to apply annotations for marking read-only memory locations using the SOFRITAS compiler [19]. These compiler annotations simply mark a variable as read-only in the source code and then attempt to propagate the read-only marking from the initial annotation across function calls and returns.

ALGORITHM 4.5.3: Read-Only Arrays in Streamcluter

```
float dist(Point p1, Point p2, int dim)
{
  int i;
  float result=0.0;
  for (i=0;i<dim;i++)
    result += (p1.coord[i] - p2.coord[i])*(p1.coord[i] - p2.coord[i]);
  return(result);
}
```

Algorithm 4.5.3 displays code from streamcluster that heavily benefits from a read-only annotation. In the dist function, the coord arrays are read-only. In fact, the coord arrays are only written as part of the application's initialization. With read-only annotations, the compiler was able to eliminate the instrumentation on the accesses to the coord array. We investigated why the automated read-only analysis failed to identify the coord array as read-only and found that alias analysis could not properly differentiate the coord array from another float member of the Point class. The getelementptr operations generated by the LLVM compiler were too similar to differentiate between the access to an array and the access to a scalar float. We tested this hypothesis by changing the data type of the cost variable to a double and found that LLVM's type-based alias analysis was then able to differentiate between the cost and coord members of Point.

4.7 Working with Libraries

Using a library with an application running with OFR serializability involves a few extra steps for the library writer. Library writers should identify library objects, so that a reader-writer lock can be associated with each one. Library API calls should be annotated as logical reads or writes of a library object. For example, inserting into a set counts as a write, while checking for a given set element is a read. This allows read-only operations to run in parallel. This approach to library integration allows legacy code to be reused safely with minimal effort. As a proof of concept, we have created the necessary annotations for C++ STL containers as many applications use these. Crucially, OFR serializability still provides coarse-grained atomicity for accesses to library objects: the lock on a set will be held until the end of the OFR. This provides natural atomicity across library API calls, making it straightforward to, e.g., atomically insert multiple elements into a set via individual insert calls. Internally, a library can use arbitrary synchronization idioms for correctness, including locks and atomic operations.

Libraries can also be rewritten and recompiled to use OFR serializability internally. Library developers should add annotations such that proper use of the library will not lead to a deadlock under the OFR serializability model. Library code using OFR serializability can be composed with application code using OFR serializability just like modules in a single application can be composed together. In a few of the applications we evaluated (e.g. bodytrack), library code is compiled and linked as part of the application.

4.8 Summary

This section of the dissertation introduced the implementation tradeoffs involved in designing a runtime system for enforcing OFR serializability. In practice, we implemented three such systems: MAMA, ORCA, and SOFRITAS. MAMA was our initial test case of using OFR serializability in Java. We then implemented the ORCA system that relied on hardware support to apply OFR serializability to C and C++ applications. Finally, we implemented a software-only system called SOFRITAS to demonstrate that OFR serializability was feasible on commodity hardware. Both ORCA and SOFRITAS relied on different implementation trade-offs to enforce OFR serializability with low performance overheads. The locks, shadowspace, and other implementation methods used to implement ORCA and SOFRITAS differed in many ways. In Sections 6 and 7, we will discuss and compare the usability and performance of the MAMA, ORCA, and SOFRITAS systems.

5 HARDWARE SUPPORT for ORDERING-FREE REGIONS

Under OFR serializability, when a thread accesses a memory location x, it must hold the lock for x (acquiring it if necessary). In a conventional software system, these lock operations could impose a prohibitive cost. However, they are cheap with targeted hardware support, just as in the case of other rich abstractions like virtual memory, memory safety [24, 59] or data-race freedom [24]. In this section we describe how hardware support to translate from memory locations to locks and a dedicated lock cache accelerate frequent OFR lock operations.

Sections 5.1, 5.2, and 5.3 discuss the hardware support designed for the ORCA system. Section 5.4 proposes extensions and modifications to the hardware design to better support the SOFRITAS lock and shadowspace designs.

5.1 Address Translation

On every memory access to a location x, ORCA needs to find the corresponding lock. To make this operation fast, ORCA restricts applications to a 60-bit virtual address space, stealing the high-order 4 bits of the address space to store locks. 2^{60} bytes of virtual memory are more than sufficient for the 48-bit physical addresses modern systems support. Each ORCA lock occupies 8B and a data address x translates to a lock address as follows.

$$L_x = (x << 3) \text{ OR } (1 << 63)$$

This simple calculation is performed by the ORCA hardware in fixed-function logic. If the OS or application runtime (e.g., a copying garbage collector) moves data in virtual memory, the locks must be moved as well to maintain the fixed data to lock mapping. Paging does not affect the data to lock mapping and both program data and locks can be paged transparently.

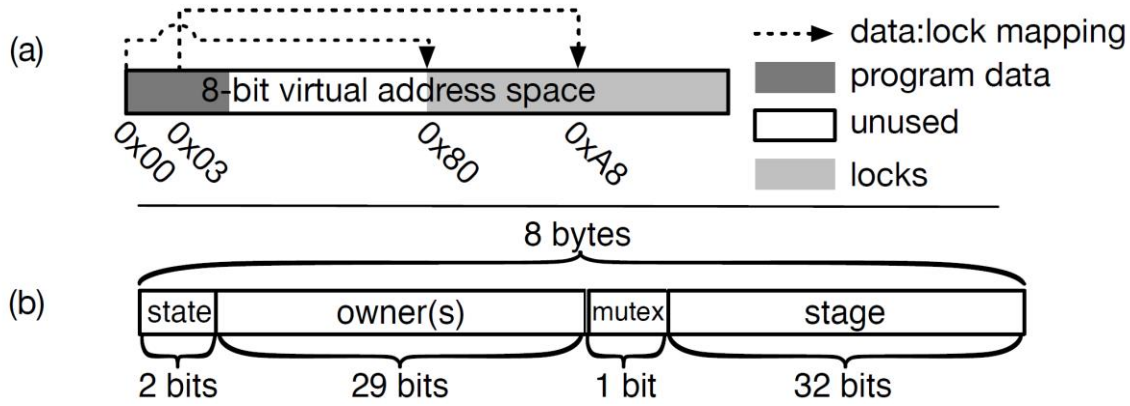

Figure 5.1: Translation from data to lock with hardware support

Figure 5.1(a) demonstrates how ORCA's hardware address translation computes lock addresses from program memory addresses. ORCA's locks require 8 bytes of lock memory for every byte of application memory. ORCA's fixed mapping requires a rigid lock format and address space setup. Performing this fixed translation in hardware requires that the starting addresses for the heap and lock shadowspace are at fixed locations for every application.

5.2 Caching

After translating a memory address to its lock's address, ORCA checks whether the executing thread has sufficient ownership to perform the memory access. ORCA maintains a reader-writer lock for every byte of program memory. These locks occupy 8 bytes, as shown in Figure 5.1(b). Each lock's state field records whether the lock is unheld, held in read-only mode, or held in read/write mode. If the lock is held, the owners field tracks the thread ID of the writer or a bitmap of readers. The mutex bit is used by mutex locks and the stage field is used for ORCA's staged locking optimization.

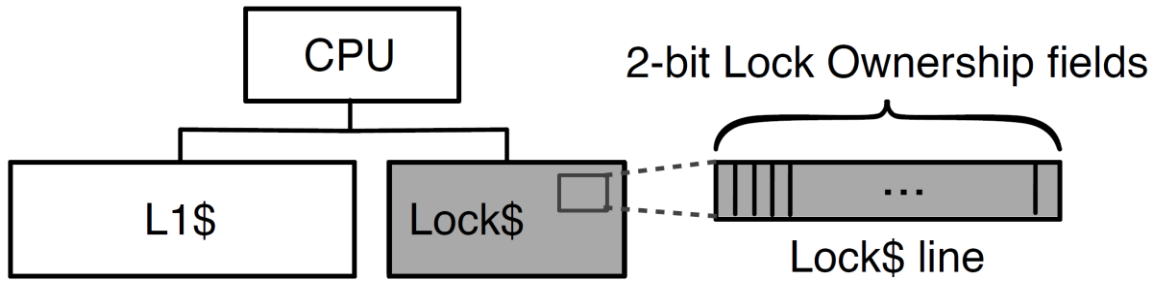

Figure 5.2: Lock cache design

The time and memory overhead of accessing ORCA's full lock representation on every memory access would be prohibitive. Similar to other hardware-enforced safety properties that maintain extensive metadata, program access patterns induce substantial spatial and temporal locality on lock accesses that can be exploited by standard caching techniques. We also find that lock ownership checks substantially outnumber lock acquires, so ORCA uses a dedicated hardware lock cache to accelerate these ownership checks. The lock cache compresses each 8B lock down to just 2 bits, for significant efficiency gains. The lock cache allows the majority of ownership checks to occur in parallel with the data cache access, hiding check latency. The lock cache also prevents lock words from polluting the data cache, and eliminates the dynamic instructions that would be required for software ownership checks.

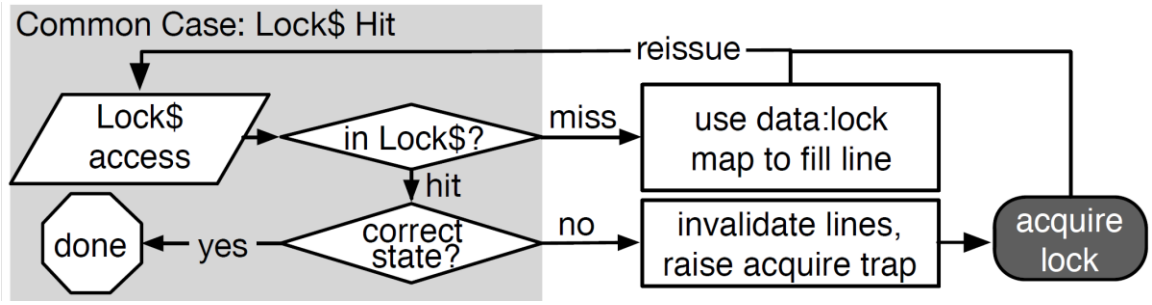

Figure 5.3: Flow-chart of lock cache operations

The lock cache maps a data address x to the ownership state of the corresponding lock lx with respect to the currently-running thread. A lock in the lock cache is represented as just 2 bits, encoding one of three possible lock states: unheld, held in read-only mode, or held in read/write mode. These 2-bit entries are packed together in a lock cache line to amortize tag

overhead (Figure 5.2): an n-byte lock cache line holds information for 4n locks which correspond to 4n contiguous bytes of program memory.

The operation of the lock cache is outlined in Figure 5.3. Memory accesses that hit in the lock cache with correct lock ownership (the common case) require no further work. Memory accesses that miss in the lock cache trigger a lock cache fill in hardware, which uses the data to lock mapping described in Section 5.3.1. Memory accesses that hit in the lock cache, but have incorrect ownership status, invalidate copies of the lock cache line in both local and remote lock caches (see below), and then raise a trap to invoke a software acquire routine. The acquire routine loads the lock into the data cache and manipulates the lock using standard atomic instructions. After the acquire, the lock's new state is cached in the lock cache.

Lock caches are read-only for simplicity. Thus, lock cache evictions do not require writebacks. To keep the lock cache state up-to-date, lock cache lines must be invalidated in several circumstances, all of which are dynamically rare. When a thread T releases a lock L, only T's ownership information changes so only T's local lock cache line containing L needs to be invalidated. When a thread T acquires L, it must invalidate its local lock cache line and, to support lock stealing, must also invalidate remote copies of the lock cache line. Updates to a thread's stage invalidate its lock cache entries to ensure that software will correctly update the scalar clock of any locks held by the thread. On context switches, a core's entire lock cache must be invalidated, as the lock cache contains ownership information for only the currently-scheduled thread. We model these costs in our simulations and find them to be tolerably low.

5.3 ISA Support

ORCA adds two new instructions to the ISA to support the lock cache and fast translation from data to lock addresses. The OFRInvalidate instruction invalidates the line in the local lock cache corresponding to a data address x (if such a line exists). An OFRInvalidate instruction is part of the software implementation of Release().

ORCA also adds an OFRLoadLock instruction that takes an address x and loads the corresponding lock l into the data cache. OFRLoadLock eliminates the extra instructions needed by software to compute lock addresses. OFRLoadLock uses the same hardware translation logic used for lock cache fills. ORCA does not require hardware implementations of lock acquire and release, deferring these infrequent operations to software to avoid the virtualization and fairness complexities of implementing reader-writer locks in hardware. ORCA does require a fixed lock format to allow hardware to fill lock cache lines, and communicates with the ORCA runtime via a user-level trap on memory accesses that hit in the lock cache with incorrect ownership.

5.4 Extending ORCA Hardware to SOFRITAS

The hardware support discussed thus far in this section pertains only to the ORCA hardware-support system for enforcing OFR serializability. The SOFRITAS software system makes multiple improvements over the ORCA system in terms of lock and shadowspace design. This section discusses how to modify the ORCA hardware for the SOFRITAS runtime system.

Given that SOFRITAS uses distributed locks instead of ORCA's monolithic locks, the hardware address translation needs to be modified to account for thread IDs. Rather than mapping addresses to a single lock shadowspace, hardware support for SOFRITAS needs to map data addresses to a different lock shadowspace for each thread. Supporting a variable number of threads also requires a more flexible hardware design than what was used for ORCA – lock shadowspace addresses cannot be hard-coded in hardware. As designed, the hardware does not need to be aware of the global metadata for each lock. The hardware would only need to directly interact with the thread-local metadata in order to perform fast lock checks.

A lock cache supporting SOFRITAS would be more efficient than the lock cache for ORCA. For ORCA, the lock cache needs to load 64 different bytes of memory to load each lock state into the lock cache. With SOFRITAS, the lock cache can simply load the correct amount of memory to cover a line in the lock cache directly from the thread-local shadowspace. The thread-

local shadowspace is already formatted in the same way that the lock cache expects data to be formatted, so the hardware does not need to parse the state of the lock from the shadowspace.

Hardware support for SOFRITAS would likely require an additional instruction to inform the hardware of the start address for each thread-local shadowspace. Otherwise, the ISA support for the lock cache would remain the same.

6 APPLYING ORDERING-FREE REGIONS TO APPLICATIONS

Using ordering-free regions with parallel applications requires some effort from the programmer (in the form of annotations), but ordering-free regions provide a number of benefits that make writing parallel applications easier. OFR serializability can detect and prevent data-races and atomicity violations that exist in parallel applications, some of which cannot be detected by a conventional data-race detector. OFR serializability is generally easy to apply to parallel applications, which is shown in a set of case studies of real applications. A small user study indicated that OFR exception are easier to use for novice programmers than reports from a data-race detector. In the following section, we evaluate the usability and programmability of ordering-free regions on parallel applications.

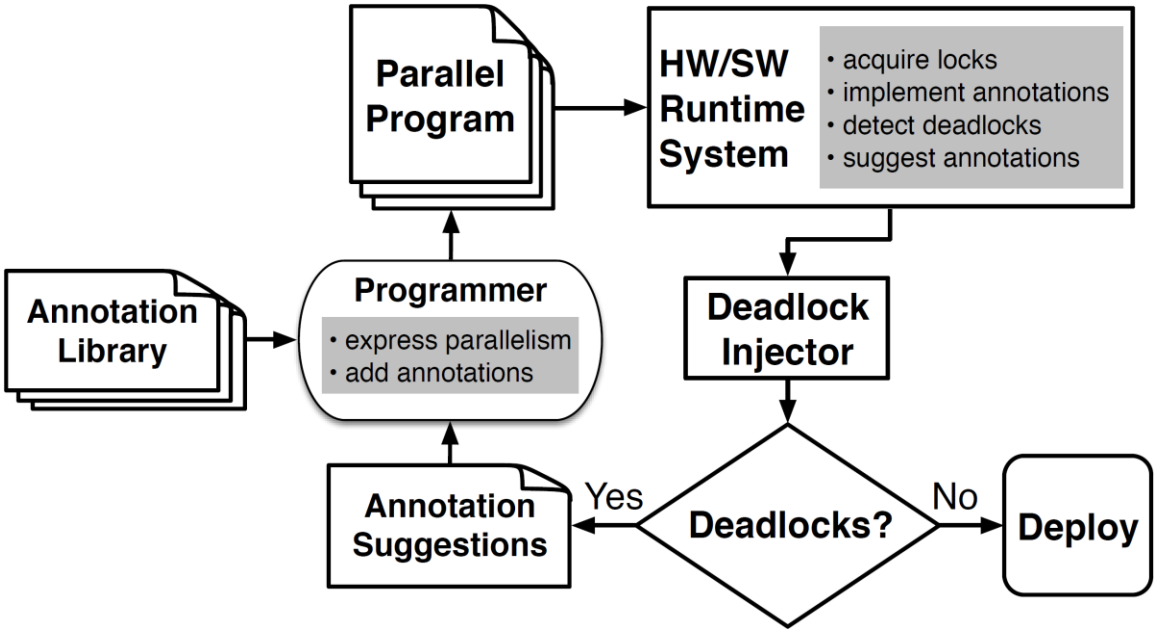

Figure 6.1: Workflow for applying ordering-free regions to an application

Figure 6.1 shows the programming model for using ordering-free regions with parallel applications. The programmer expresses the parallelism inherent in the problem they are trying to solve in parallel. With the help of a HW/SW runtime system, the programmer applies annotations to the code to prevent OFR exceptions. The HW/SW system assists the programmer

by acquiring locks (by default), detecting cycles, and suggesting annotations. A deadlock injector can be applied to attempt to discover additional OFR exceptions. Once the programmer feels that sufficient deadlocks have been discovered, the application can be deployed. We applied this approach to multiple benchmark suites across three implementations of OFR serializability: MAMA, ORCA, and SOFRITAS. The following section provide an evaluation of the usability of each of these systems.

6.1 Annotations

To evaluate the usability of OFR serializability against a standard parallel programming model, we measured the annotation burden of applying OFR serializability to multiple benchmark suites in Java, C, and C++. For the C and C++ applications, we compared the number of annotations against the number of pthreads annotations required for the same applications.

6.1.1 MAMA

Table 6.1: Lines of code and static synchronization in Java benchmarks

Benchmark	LoC	synchronized	volatile	wait()	notify()	run()	join()	Barrier
crypt	314	0	0	0	0	2	2	0
lufact	461	0	0	0	0	1	1	1
lusearch	124,105	440	21	18	27	1	1	0
matmult	187	0	0	0	0	1	1	0
moldyn	487	0	0	0	0	1	1	1
montecarlo	1,165	0	0	0	0	1	1	0
pmd	60,062	15	2	0	0	0	0	0
series	180	0	0	0	0	1	1	0
sor	186	0	0	0	0	1	1	1
sunflow	21,970	43	0	0	0	2	2	0
xalan	172,300	107	0	6	8	1	1	0

Table 6.2: Dynamic synchronization in Java benchmarks

Benchmark	synchronized	volatile	wait()	notify()	run()	join()	Barrier
crypt	0	0	0	0	14	14	0
lufact	0	0	0	0	7	7	29952
lusearch	1327134	1E+06	64	64	7	7	0
matmult	0	0	0	0	7	7	0
moldyn	0	0	0	0	7	7	2424
montecarlo	0	0	0	0	7	7	0
pmd	322	0	0	0	0	0	0
series	0	0	0	0	7	7	0
sor	0	0	0	0	7	7	1600
sunflow	770	0	0	0	14	14	0
xalan	4448917	0	8	1704	7	7	0

Tables 6.1 and 6.2 detail the number of lines of code and synchronization used in the original versions of 11 Java benchmarks from the Java Grande [70] and DaCapo [9] benchmark suites. Both statically and dynamically, atomicity synchronization (synchronized and volatile) are significantly more common than ordering synchronization (wait, notify, run, join, and Barrier). Within these two suites, there are a few benchmarks that exhibit point-to-point ordering synchronization, using wait and notify, such as lusearch and xalan. There are also applications that use barrier ordering, such as lufact, moldyn, and sor. These applications were part of an initial study of the feasibility of OFR serializability and offered a diverse set of synchronization characteristics.

Table 6.3: Dynamic deadlocks and lock releases for Java applications

	Deadlocks	Lock Releases	
Benchmark	Safe	Liveness	Performance
crypt	5,250,330	0	0
lufact	4,240,434	2,977	12,583,386
lusearch	250	0	43
matmult	700,405	0	0
moldyn	2,019,626	178	0
montecarlo	647,279	0	143,362
pmd	3,442	0	1,915,602
series	15	0	0
sor	4,508,422	4,058	0
sunflow	262,448	1	27,948
xalan	19,908	0	0

To evaluate the effectiveness of MAMA, we applied the algorithm to multiple parallel benchmarks and recorded where deadlocks occurred in the target program. The *Safe* column of Table 6.3 details the dynamic deadlocks that occurred during the execution of the benchmark suite under MAMA. As the first investigation into OFR serializability, MAMA did not batch release locks at ordering constructs but rather detected all deadlocks and released locks when it was safe to do so (e.g. when one thread was waiting on an ordering construct). The majority of deadlocks occurred while one of the threads was either joined, waiting on a condition variable, at a barrier, or exited. Thus, most deadlocks could be broken with confidence that MAMA was not breaking the atomicity required by the program. In later implementations of OFR serializability (ORCA and SOFRITAS), these deadlocks would be automatically broken using either lazy or eager lock releases at ordering constructs. The *Liveness* and *Performance* columns list how many locks are dynamically released to avoid deadlocks and serialization, respectively. In the Java applications, lock releases to avoid serialization were common because many of the applications use the main thread as a worker thread. This pattern requires releasing locks on the main thread after initialization to avoid serializing the other worker threads after the main thread's execution.

Table 6.4: Static annotations needed for Java benchmarks

Benchmark	Liveness	Performance
crypt	0	0
lufact	1	4
lusearch	0	4
matmult	0	0
moldyn	3	0
montecarlo	0	28
pmd	0	4
series	0	0
sor	1	0
sunflow	1	3
xalan	1	0

Table 6.4 details the number of static annotations required to support OFR serializability in the Java benchmarks. The annotations are partitioned into two categories: liveness and performance. Liveness annotations were required to break OFR exception that occurred during execution. Performance annotations were added to prevent unnecessary serialization. lufact, moldyn, sor, and sunflow required annotations for liveness. Despite the number of dynamic deadlock breaks that were required for these benchmarks, the number of static annotations to perform these deadlock breaks is just seven across all benchmarks. In lufact, sor, and moldyn, deadlocks occur despite these benchmarks not having any synchronized blocks in the original code because these benchmarks all use barriers for synchronization. In each of these benchmarks, it is safe to break the deadlocks that occur because the overlapping reads and writes are synchronized by the barrier.

In some cases, we explicitly broke the atomicity guarantees of MAMA in order to allow increased parallel execution. In lufact, lusearch, montecarlo, pmd, and sunflow, we identified shared counter variables that were updated atomically, requiring that the locks for these variables be released early to allow the threads to execute in parallel. In montecarlo and sunflow, we also

identified locks that were acquired for static initialization and could thereafter be downgraded to read-shared.

Our experiences in applying OFR serializability to Java applications with MAMA lead to many of the design choices made in implementing ORCA and SOFRITAS. We identified that deadlocks were extremely common at ordering constructs and realized that these deadlocks could be automatically broken. In ORCA and SOFRITAS, ordering constructs end the ordering-free region and release all locks, leading to better usability of the systems.

6.1.2 ORCA and SOFRITAS

Table 6.5: Ordering and atomicity annotation for pthreads and ORCA/SOFRITAS

		pthreads	SOFRITAS		
App	Ordering	Atomicity	Mutex	Release	End/ContinueOFR
blackscholes	2	-	-	-	-
bodytrack	17	34	3	20	-
canneal	3	13	1	7	1
dedup	9	13	5	18	1
ferret	8	7	2	7	1
fluidanimate	16	10	5	20	-
streamcluster	30	6	-	11	-
swaptions	2	-	-	-	-
gups	2	2	-	1	-
pagerank	2	10	-	7	-
histogram	2	-	-	-	-
kmeans	2	-	-	-	-
linear_regression	2	-	-	-	-
matrix_multiply	2	-	-	-	-
pca	2	4	-	3	-
reverse_index	2	4	1	2	-
string_match	2	-	-	-	-
word_count	2	-	-	-	-
pbzip2	34	103	7	10	-

Table 6.5 reports the annotation burden for enforcing atomicity with standard pthreads primitives and the annotation burden for refining coarse atomic regions with SOFRITAS. The annotations

required for ORCA closely match the annotations required for SOFRITAS, though a small number of additional annotations may be required due to false exceptions caused by ORCA's lazy release policy. The Ordering column gives the number of ordering constructs used in each application. bodytrack, canneal, fluidanimate, and streamcluster use barriers, and bodytrack, dedup, and ferret condition variable waits. The Atomicity column reports the number of atomicity constructs (lock and unlock calls) present in the pthreads version of each application. Systems that provide SFR and RFR consistency require the same atomicity and ordering constructs as pthreads.

The next three columns in Table 6.5 report the number of annotations required for refining the coarse atomicity provided by SOFRITAS. The Mutex column shows the number of RequireMutex() annotations required. In all cases, SOFRITAS correctly suggested that a RequireMutex() annotation is required by examining the lock state when an OFR exception occurs. If a lock has multiple shared readers and at least one thread is attempting to acquire write privileges, a RequireMutex() annotation is almost certainly required. The SOFRITAS compiler's post-dominator analysis avoids the need for 13 additional mutex annotations.

The Release column reports the number of Release() annotations required for each application. In most cases, the number of Release() annotations closely corresponds to the number of atomicity constructs required for the pthreads version of the application. The disparity between the number of necessary release annotations and pthreads locks can be explained by two major factors. First, the pthreads applications often use coarse-grained locking to protect data structures, whereas SOFRITAS automatically uses fine-grained locking for all memory locations. For example, dedup uses hash-table and memory-buffer structures that are protected by coarse-grained locking in the pthreads version. Second, atomicity violations exist in some of the PARSEC benchmarks that are not prevented by the existing pthreads synchronization. We discuss these atomicity violations more in Section 6.2.

The End/ContinueOFR column reports the number of EndOFR() or ContinueOFR() annotations that were added. dedup and ferret both exhibit pipeline parallelism such that each

stage of the pipeline performs some actions and then enqueues data for the next stage of the pipeline. Each enqueue operation represents the end of the thread's atomic actions on the enqueued data, so we use a single EndOFR() annotation in each benchmark to represent this. canneal represents a different case in which Release() annotations handle all of the necessary release for the benchmark, making the batch lock release operations at each barrier wait superfluous. To improve the performance of canneal, we add a single ContinueOFR() annotation to the barrier wait to prevent the batch lock release. This optimization yields a 4x speedup.

Table 6.6: Qualitative ease of adding OFR annotations

App	Easy	Hard
blackscholes	-	-
bodytrack	20	-
canneal	7	-
dedup	18	-
ferret	4	3
fluidanimate	20	-
streamcluster	11	-
swaptions	-	-
gups	1	-
pagerank	7	-
histogram	-	-
kmeans	-	-
linear_regression	-	-
matrix_multiply	-	-
pca	3	-
reverse_index	2	-
string_match	-	-
word_count	-	-
pbzip2	10	-

Table 6.6 characterizes the relative ease of adding OFR annotations to the parallel applications. When an OFR exception occurs, the SOFRITAS runtime system attempts to suggest the correct location and type of annotation that is required to properly refine the coarse atomicity provided by SOFRITAS. The Easy column reports the number of annotation

suggestions that we found to be easy to place using the suggestions provided by SOFRITAS. These annotations were either located at the exact line suggested by SOFRITAS or close to the suggested line.

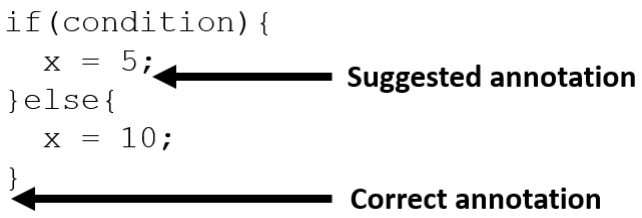

Figure 6.2: A "Close" annotation suggestion

Figure 6.2 demonstrates an annotation suggestion that was close to the suggested line. In the close cases, SOFRITAS suggested placing an annotation inside of control-flow, and we determined that the annotation should be placed after the control-flow structure to cover multiple paths. If we automatically placed the annotation suggested by the SOFRITAS runtime, similar deadlocks would occur on the opposite control-flow path, leading to additional deadlocks and annotations.

The Hard column reports the number of annotations that were difficult to place. These annotations were localized to the queue used by ferret. These difficult-to-place annotations arise due to interleavings caused by existing annotations. Internally, the queue relies on head and tail pointers that are protected by mutexes. Initially, SOFRITAS correctly suggests a release annotation on the tail pointer. Once this annotation has been added, one of the two suggestions provided by SOFRITAS on the next OFR exception may be incorrect due to interleavings caused by the existing annotation. For ferret, the programmer must understand that checking whether the queue is empty must be atomic with removing an item from the queue. Even though not all the suggestions provided by SOFRITAS are exactly correct, any incorrect suggestions still point to the correct source-code files and data-structures, providing the programmer with a reasonable starting point for resolving the OFR exception. Further, one of the two suggestions is correct, leaving the programmer with a multiple-choice question of how to resolve the OFR exception.

Beyond a comparable annotation burden, SOFRITAS precisely suggests a fix via a fail-stop exception for missing annotations. By contrast, missing locks in pthreads and other models [8, 48] are not fail-stop and do not help fix code. Missing locks in pthreads and other models can lead to memory corruption and application crashes with no usable debugging trace.

6.2 Bug Detection

By analyzing SOFRITAS's annotations and serialization of OFRs, we identified 7 existing atomicity violations in PARSEC benchmarks. Specifically, we found atomicity violations in the pthreads versions of bodytrack, ferret, fluidanimate, and streamcluster. We verified each of these violations by directly instrumenting the code to ensure that an atomicity violation did in fact exist. For 6 of these violations, SOFRITAS automatically prevents the bugs from manifesting as failures by correctly enforcing OFR serializability without any programmer involvement. The remaining violation (in fluidanimate) initially raised an OFR exception. SOFRITAS precisely reported the annotations needed to resolve the OFR exception with no need for manual reasoning.

6.2.1 bodytrack

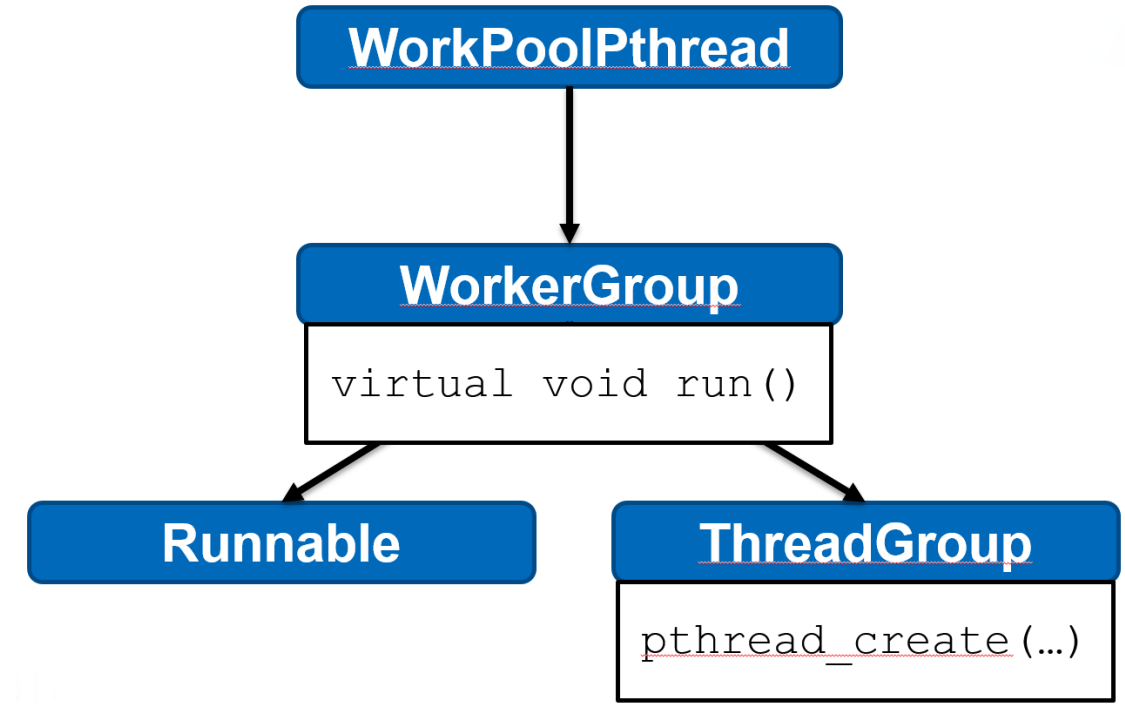

Figure 6.3: Inheritance diagram for construction bug in bodytrack

We give an illustrative example of SOFRITAS's ability to automatically prevent atomicity violations from bodytrack. Figure 6.3 demonstrates a chain of constructor calls that leads to a bug in bodytrack. In bodytrack, the WorkPoolPthread class inherits from the WorkerGroup class, which in turn inherits from ThreadGroup and Runnable. In its constructor, the WorkerGroup class passes its **this** pointer to ThreadGroup::CreateThreads, which spawns threads and calls the virtual run() method on the WorkerGroup object. In order to call the virtual method, each thread must read the vptr (virtual table pointer). The main thread simultaneously writes to the vptr as WorkPoolPthread finishes construction. Although this is defined by the C++ standard [38] for single-threaded code, this behavior constitutes an atomicity violation on accesses to vptr. SOFRITAS automatically prevents this atomicity violation with no annotations required by

preventing the child threads from accessing the virtual table pointer until the object has finished construction and the main thread has joined on the thread pool.

6.2.2 fluidanimate

```
int index = (ck*ny + cj) *nx + ci;

if(border[index])
  pthread_mutex_lock(&mutex[index][CELL_MUTEX_ID]);

int np = cnumPars[index];

if(np % PARTICLES_PER_CELL == 0) && cnumPars[index] != 0) ){
  cell->next = cellpool_getcell(&pools[tid]);
  cell = cell->next;
  last_cells[index] = cell;
}

++cnumPars[index];

if(border[index])
  pthread_mutex_unlock(&mutex[index][CELL_MUTEX_ID]);
```

Figure 6.4: Data-race found in fluidanimate

Figure 6.4 lists code that leads to a data-race in fluidanimate. In this code, the border array is meant to identify elements of the data arrays that may be accessed by more than one thread and therefore need to be protected by a lock. However, the border array is computed incorrectly, leading to a data-race. SOFRITAS correctly synchronizes all accesses to the cnumPars array and prevents the data-race from occurring. As future work, attempting to identify non-conflicting memory accesses between threads, as the border array in fluidanimate attempts to do, may prove to be a useful optimization for SOFRITAS.

6.2.3 memcached

```
Thread 1                                Thread 2

process_arithmetic_command(...)         process_arithmetic_command(...)
{                                       {
  ...                                     ...
  it = item_get(key, nkey)

                                          it = item_get(key, nkey)
                                          ...
                                          add_delta(...)
                                          {
                                            ...
                                            // operate on new item
                                            item_replace(it, new_it);
                                            ...
                                          }
  ...
  add_delta(...)
  {
    ...
    // operate on old item
    memcpy(..., it, res);
    ...
  }
```

Figure 6.5: Buggy interleaving in memcached-127. Image copied from https://github.com/jieyu/concurrency-bugs

To test SOFRITAS on a larger code base, we examined a known concurrency bug in memcached [72,80]. In the memcached-127 bug, a cached item is read and updated in separate critical sections. Both the read and update are protected by the same lock, which prevents existing

strong memory consistency models from detecting the bug. We ran SOFRITAS on the memcached-127 bug. With no additional annotations, SOFRITAS detects the concurrency bug via an OFR exception and pinpoints the item_replace() function call as the correct location for an annotation.

6.3 Case Studies

In the process of applying OFR serializability to parallel applications in C and C++, we examined a few applications in depth to understand more precisely how they interacted with ordering-free regions. The following subsections provide case studies of dedup, memcached, and pbzip2. The annotation and evaluation of both memcached and pbzip2 were performed by users other than the writer of this dissertation as a test of the usability of the system by additional programmers (though admittedly somewhat expert users due to their involvement in the project).

6.3.1 dedup

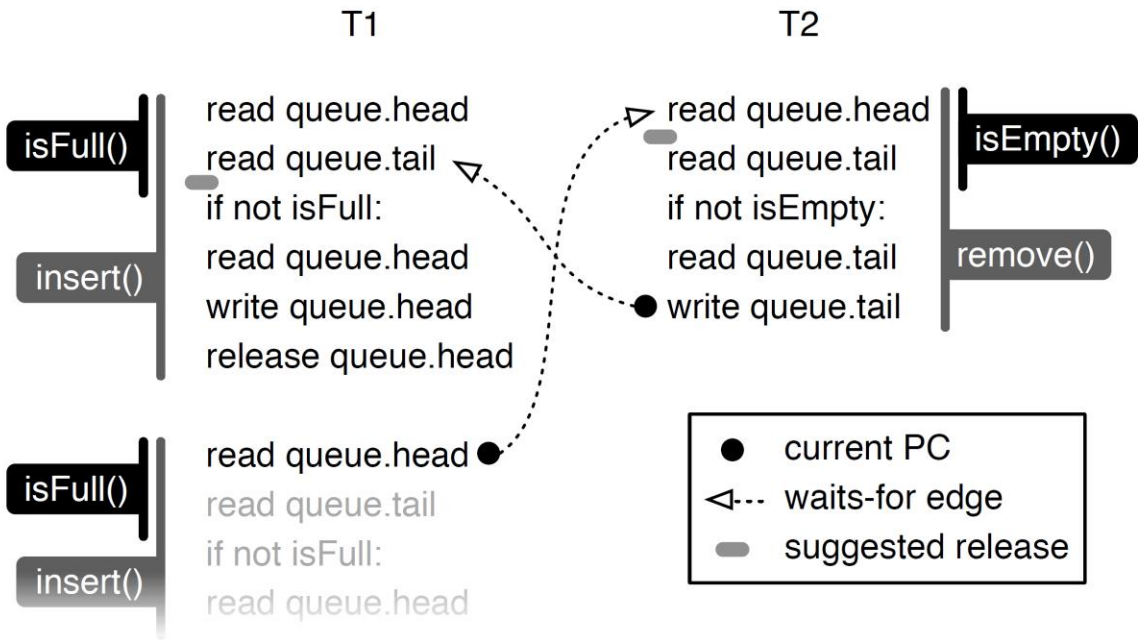

Figure 6.6: Example from dedup queue that lead to difficult to use OFR exception reports

Applying OFR serializability to the queue implementation in dedup presented a few unique challenges. Figure 6.6 illustrates a dependence cycle from dedup where OFR serializability suggested incorrect annotations. This particular example occurred after a release annotation had been correctly suggested and placed at the end of insert() and a mutex annotation had been correctly suggested and placed on queue.head. The presence of these two annotations leads to a new OFR exception for which OFR serializability does not suggest the correct location.

In the encountered dependence cycle, T_1 performs an insert which requires calling isFull() to ensure there is space in the queue. At the end of the insert, T_1 releases the mutex lock on queue.head while holding a read lock on queue.tail. Next, T_2 performs a remove, first checking that the queue is non-empty. At the end of the remove, T_2 gets blocked trying to acquire write permissions on queue.tail, which is not protected by a mutex lock (yet). Next, when T_1 attempts to perform another insert, T_1 becomes blocked trying to acquire the lock on queue.head, forming a dependence cycle. Based on the last accesses to queue.head and queue.tail, OFR serializability

suggests releases in isFull() and isEmpty(). These releases will violate the atomicity of insert() and remove(), respectively. The correct solution is to release both queue.head and queue.tail after performing either an insertion or removal. However, even in this subtle case, OFR serializability directs the programmer to all of the relevant parts of the correct solution.

dedup was also the only benchmark in which we added releases for performance rather than correctness. Profiling revealed that threads in the early stages of the pipeline were blocking on queue locks held by threads in later stages. Four unlock annotations were added to ensure that the queue locks are released along all control-flow paths. This optimization yielded a 1.24x speedup in dedup with four threads.

6.3.2 memcached

We ported memcached using OFR serializability to see how a real-world application works with OFR serializability. While OFR serializability requires 182 release annotations to be added to the code, the system gave correct suggestions for 155 annotations; the other 27 releases were inserted close to where an annotation was suggested. Besides these release annotations, 26 mutex annotations were added, all of which could be done mechanically with the suggestions given. After adding these annotations, we tested memcached using memslap to generate requests with 2 to 1024 concurrent users. We ran these tests up to 50,000 times, and memcached was able to respond to the requests correctly and in a timely fashion. The experience of porting memcached shows that OFR serializability can scale to non-trivial real-world applications like memcached, and that OFR exceptions can provide accurate annotation guidance in large code bases.

6.3.3 pbzip2

As a test of using OFR serializability on another real-world application, we ported pbzip2 v.1.1.13 to use SOFRITAS. Porting pbzip2 from a pthreads implementation to using SOFRITAS required

under 7 hours of real time, including time that was spent building and deploying SOFRITAS on a new system. Annotating pbzip2 required 7 mutex annotations and 10 release annotations, which compared favorably to the 103 atomicity annotations required in the pthreads implementation. One of the annotations was somewhat difficult to add because the SOFRITAS library had not marked pthread_cond_timed_wait() as a function that ends ordering-free regions. Once the runtime system was made aware of this function, the annotations added to pbzip2 were all easy to add. Porting pbzip2 from the original pthreads version to a SOFRITAS-compliant version required only 6 hours of work. In the future, we hope to compare the ported version of pbzip2 to a from-scratch parallel implementation under OFR serializability from the bzip2 serial baseline.

6.4 User Study

We empirically evaluated the difficulty of inserting Release() annotations through a survey of 45 computer science graduate students. Participants had to place lock acquires and releases in an unsynchronized program to correctly implement atomicity. The code given to participants is listed below. Participants were asked to ensure that the setter methods could execute in parallel. The survey had two variants of the synchronization task with identical program code. One variant was accompanied by OFR exception reports, the other by reports from a data race detector which are similar to the exceptions generated by previous memory consistency models [8, 29, 48]. We randomized the order of the variants to account for learning effects.

ALGORITHM 6.4.1: User Study Sample Code

```
class Vector3d{
  int x,y,z;

  void setX (int newX) {
    x = newX;
  }

  void setY (int newY) {
    y = newY;
  }

  void setZ (int newZ) {
    z = newZ;
  }

  void normalize() {
    int a = sqrt(x*x + y*y + z*z);
    x = x / a;
    y = y / a;
    z = z / a;
  }
}
```

To correctly synchronize the sample code, participants needed to acquire a lock associated with each variable in its setter method (e.g. lock(L_X) in setX) and release the lock at the end of the method. For the normalize method, the participants needed to acquire all three locks and release them in reverse order, as shown below.

ALGORITHM 6.4.2: User Study Solution

```
class Vector3d{
  int x,y,z;

  void setX (int newX) {
    Lock(Lx);
    x = newX;
    Unlock(Lx);
  }

  void setY (int newY) {
    Lock(Ly);
    y = newY;
    Unlock(Ly);
  }

  void setZ (int newZ) {
    Lock(Lz);
    z = newZ;
    Unlock(Lz);
  }

  void normalize() {
    Lock(LX);
    Lock(LY);
    Lock(LZ);
    int a = sqrt(x*x + y*y + z*z);
    x = x / a;
    y = y / a;
    z = z / a;
    Unlock(LZ);
    Unlock(Ly);
    Unlock(Lx);
  }
}
```

The survey also asked participants to answer 12 questions about parallel programming using both data-race detector reports and OFR exceptions. The survey questions are listed below. The scoring or scale for the question is listed in parentheses.

LISTING 6.4.3: Survey Questions

1. Please rate your expertise in parallel programming. (1-7)
2. Please define a mutex (lock) as used in parallel programming. (0-1)
3. Please define a data-race in a parallel program. (0-3)
4. Please rate how confident you are in your definition of a data-race. (1-7)
5. Please define a deadlock in a parallel program. (0-2)
6. Please rate how confident you are in your definition of a deadlock. (1-7)
7. Given the following code and multiple reports from a data-race detector, please insert locks and releases to ensure that the code executes correctly. The setX, setY, and setZ methods should all be able to execute in parallel. (0-1)
8. Please rate how confident you are that you have fixed all of the bugs in the buggy program from question 7. (1-7)
9. Please rate how easy it was to insert locks into the buggy program in question 7. (1-7)
10. Given the following code and multiple reports from a deadlock detector, please insert lock releases to ensure that the code executes correctly. The setX, setY, and setZ methods should all be able to execute in parallel. (0-1)
11. Please rate how confident you are that you have fixed all of the bugs in the buggy program from question 10. (1-7)
12. Please rate how easy it was to insert locks into the buggy program in question 10. (1-7)

Survey questions 7-9 and 10-12 were reordered on half of the surveys to account for learning effects. Questions 2, 3, and 5 were used to check the participant's self-rated expertise and were meant to identify students who would be more likely to perform well on one of the two variants. We summarize the scores from question 2,3, and 5 in the Sum column below.

Table 6.7: Summary statistics for survey questions

Question	1	2	3	4	5	6	Sum	7	8	9	10	11	12
AVG	3.19	0.91	1.36	4.34	1.55	5.39	3.82	0.66	4.30	4.68	0.80	4.48	5.05
STDEV	1.33	0.29	1.04	1.82	0.76	1.42	1.39	0.48	1.92	1.85	0.41	1.76	1.61

Table 6.7 presents summary statistics for each of the questions in the survey. On average, participants rated their own expertise in parallelism at a 3.19 out of 7. The questions about parallel programming (2,3,5) verified these scores with a sum of 3.82 out of 6 possible

points for all of the answers. Although participants seemed more confident about the definition of a deadlock as compared to the definition of a data-race, the results for those two questions were not statistically significant.

The summary statistics for the questions in which participants were asked to correctly synchronize a simple parallel program based on the reports from a data-race and deadlock detector seemed to indicate that users had an easier time of synchronizing code with the deadlock detector. However, given the sample size of 45 students, the summary statistics were not statistically significant. To identify whether or not OFR exception reports were useful to novice programmers, we dug further into the results.

The survey partitioned participants into three groups: (i) those who correctly synchronized both variants, (ii) those who correctly synchronized one variant but not the other, and (iii) those who incorrectly synchronized both variants. Group (i) was experienced enough at parallel programming that they were likely able to synchronize the code without assistance from either tool. Group (iii) was inexperienced enough at parallel programming that they did not understand the basic concepts of synchronization. Thus, we focused on the second group (ii) that got one variant correct, but not the other. We define the probability, p_{orca}, of getting the OFR exception variant correct, but the data race variant incorrect. We define the probability, p_{race}, of getting the data race variant correct, but the OFR exception variant incorrect. Our data support the fact that p_{orca} is significantly greater than p_{race}. To determine the statistical significance of that claim, we computed the 95% confidence interval of $p_{orca} - p_{race}$, which is [0.001, 0.271]. The relatively greater likelihood of correctly solving the OFR exception variant and not the data race variant suggests that using OFR exceptions to add synchronization is easier than using a data race detector. We further note that data race reports can encourage “narrowly” fixing a race on an individual access without providing sufficient atomicity across accesses, as with the normalize method in our survey. Multiple survey participants made this mistake, further highlighting the value of OFR serializability.

7 PERFORMANCE OF ORDERING-FREE REGIONS

We evaluated the performance overheads of three runtime systems for enforcing OFR serializability on parallel applications. For each of these systems, we evaluated the runtime and space performance overheads as compared to a baseline using standard synchronization.

7.1 MAMA

We performed an initial investigation into OFR serializability by implementing MAMA (Mostly Automatic Management of Atomicity) [20]. Using the Roadrunner [32] framework, we developed a runtime system to dynamically apply OFR serializability to Java applications. RoadRunner's dynamic instrumentation adds overhead but enables us to gather preliminary indications of the effectiveness of OFR serializability. MAMA associates a reader-writer lock with each program variable. On each variable access, MAMA requires that the accessing thread either already owns or acquires the lock for the given variable.

To gain confidence that the MAMA algorithm works correctly on programs without atomicity constructs, we removed the locking from our benchmarks. All synchronized blocks were automatically removed by modifying RoadRunner to not insert MONITOR_ENTER and MONITOR_EXIT bytecodes. We also manually removed uses of Java's ReentrantReadWriteLock, Java atomics, and concurrent data structures. For example, we replaced the PriorityBlockingQueue used by sunflow with a non-concurrent PriorityQueue. After this synchronization removal (but prior to applying MAMA), sunflow and xalan produced incorrect output, though the other benchmarks produced correct results on multiple trial runs.

We evaluated MAMA on benchmarks from the Java Grande [70] and DaCapo [9] benchmark suites. From the DaCapo suite, only avrora, lusearch, jython, pmd, sunflow, tomcat, and xalan run under RoadRunner's baseline instrumentation. We removed jython from our suite because it did not display significant parallelism. Under MAMA's instrumentation, avrora and

tomcat exhibited bugs that were unable to debug. We ran all of the benchmarks on a 32-core/64-thread machine with four Intel Xeon E7-4820 2.0 GHz sockets and 128 GB RAM. For the parallel experimental results, all benchmarks were run using 8 threads and pinned to a single socket to avoid the performance overheads of data-sharing across multiple sockets. Runtime performance overheads were measured using Java's currentTimeMillis(), and memory overheads were measured at the high-water mark using the jvisualvm tool provided by the JDK. We ran RoadRunner using fine-grained field and array tracking (one shadow variable per field and one shadow variable per array element). For crypt, lufact, sor, montecarlo, sunflow, and xalan, we used coarse-grained array tracking but chunked the arrays into 64 buckets to reduce the runtime and memory overheads of fine-grained tracking on large arrays. We validated that the benchmarks executed correctly using the built-in validation mechanisms of the Java Grande and DaCapo benchmarks. For the performance evaluation, we averaged five runs of each benchmark.

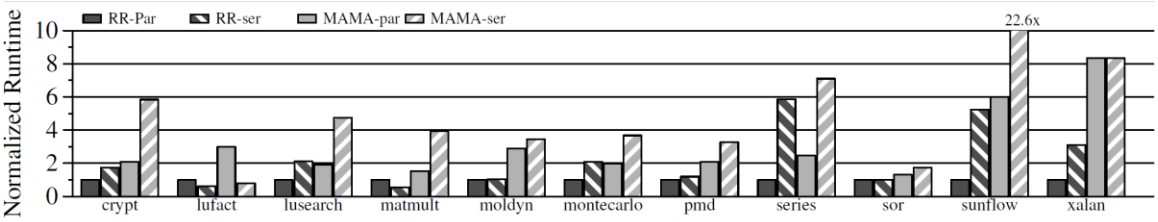

Figure 7.1: Runtime of parallel RoadRunner, serialized RoadRunner, parallel MAMA, and serial MAMA, normalized to parallel RoadRunner

We evaluated MAMA's parallel execution (MAMA-par) against a few different baselines: parallel RoadRunner execution (RR-par), serialized RoadRunner execution (RR-ser), and serialized MAMA execution (MAMA-ser). We compare MAMA to serialized baselines to verify whether MAMA can indeed exploit the parallelism in each workload, and whether MAMA exploits enough parallelism to overcome its locking overheads. By comparing the difference between RR-par and RR-ser with the difference between MAMA-par and MAMA-ser, we can determine whether or not MAMA preserves the potential parallel speedup in each benchmark.

The results of our evaluation are shown in Figure 7.1. On average, RoadRunner incurs approximately 6x overhead over the uninstrumented programs. Due to the overheads of locking on every variable access, MAMA is never faster than the RR-par baseline. Nevertheless, MAMA-par is capable of exploiting parallelism in many benchmarks. Compared to the RR-ser baseline, MAMA-par is competitive in many cases and performs better than RR-ser on lusearch, montecarlo, and series. In these cases, MAMA-par overcomes its locking overheads with parallelism.

Finally, we compared MAMA-par to MAMA-ser to measure the amount of parallelism in the execution of the benchmarks under MAMA. In almost all cases, MAMA-par handily outperforms MAMA-ser. There are two exceptions. lufact does not scale well with eight threads under RoadRunner's instrumentation (RR-par is slower than RR-ser). xalan does not exhibit parallelism under MAMA, even with early lock breaking, though there is clearly parallelism within the workload. More investigation is necessary to determine how to unlock xalan's parallelism.

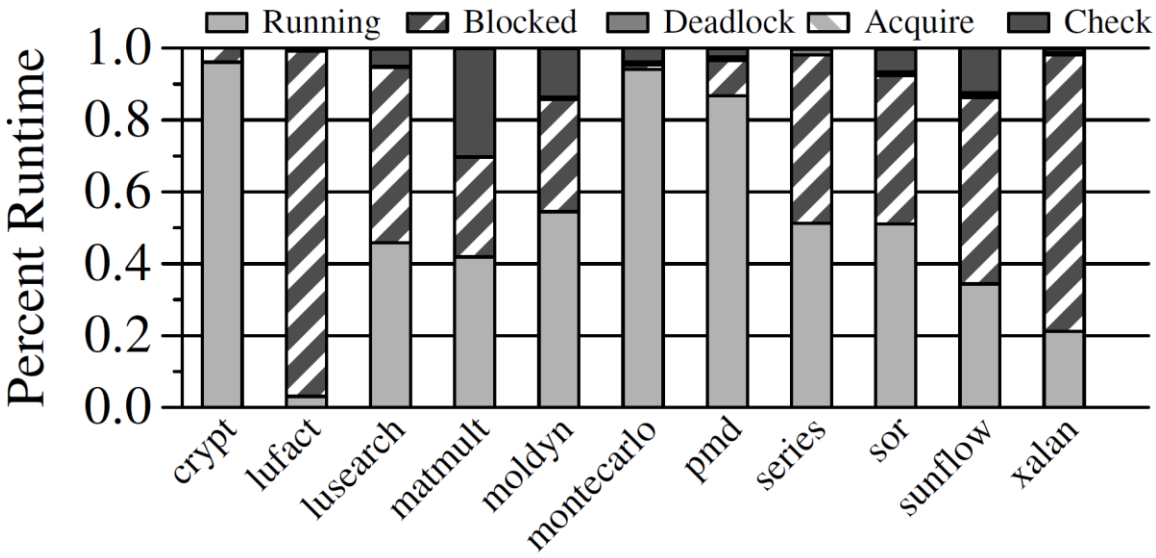

Figure 7.2: Percentage runtime for various routines and states in MAMA

Most of the performance overheads of MAMA stem from two sources: testing locks for ownership and serialization due to contested locks. Figure 7.2 shows the summed performance counters for all threads in each benchmark. In general, the deadlock detector is run infrequently

and only when threads are blocked. Thus, the overheads of deadlock detection are negligible. On every variable access, MAMA must check to see if the corresponding lock is already held. In the case of read-sharing, MAMA must check to ensure that the thread is one of the read owners of the lock. Recording the read owners of a lock is necessary to allow deadlocks to be broken at runtime. However, this overhead might be reduced by simply denoting that some thread had read ownership of a lock rather than explicitly recording which thread held ownership. For some benchmarks, such as xalan, contended locks cause the program's execution to be serialized. For these benchmarks, more investigation is necessary to find ways to allow multiple threads to execute under MAMA while still preserving the atomicity of the program as much as possible.

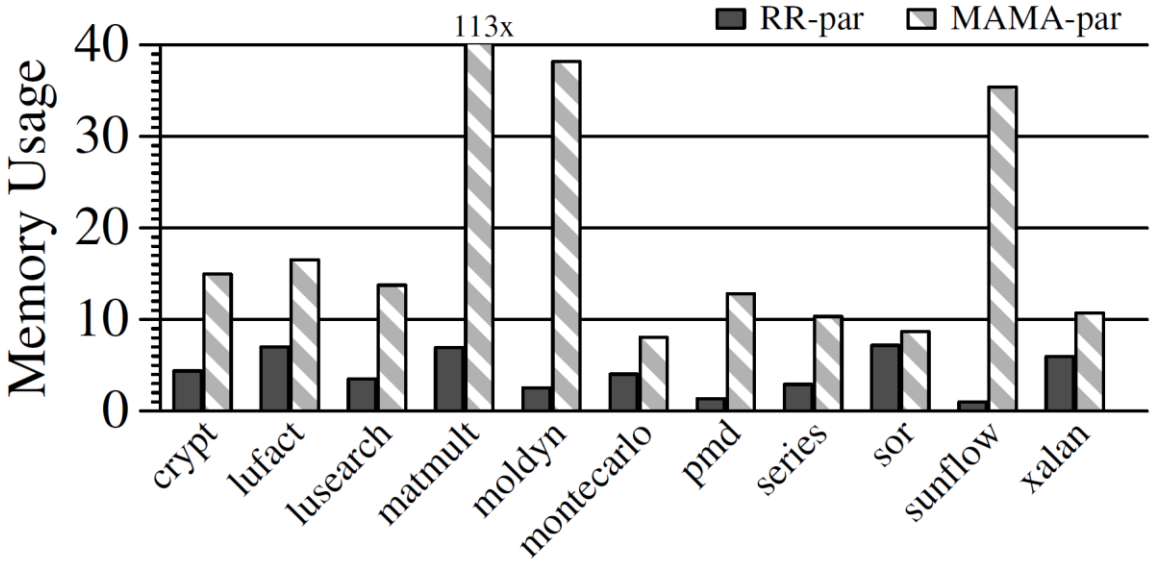

Figure 7.3: Normalized high-water mark memory usage parallel RoadRunner and parallel MAMA, normalized to JVM execution

We also evaluated the memory overheads of MAMA on our benchmark suite. MAMA requires a reader-writer lock for every shared variable in the program, which can lead to high memory overheads, as shown in Figure 7.3. The memory overheads for MAMA range from 8x on montecarlo to 113x on matmult, as compared to the uninstrumented Java baseline (without RoadRunner). Although the array chunking optimization reduces matmult's memory overheads to

just 9.8x, it also results in serialized execution for this workload. More adaptive array chunking could alleviate this time-space tradeoff. MAMA's memory overheads could possibly be further reduced by using a more compact reader-writer lock or by avoiding the need to record all of the current readers of the lock.

7.2 SOFRITAS

We evaluated SOFRITAS by running and annotating selected benchmarks from PARSEC [7], Phoenix [64], approximate computing benchmarks [3], and the real-world pbzip2 v1.1.13. We use the native inputs for all PARSEC benchmarks and the largest available input for Phoenix. We extend the execution of linear regression by 100 times to yield a reasonable baseline runtime of more than a second with 16 threads. We use custom inputs for the approximate computing benchmarks that yield a baseline runtime of a few seconds and scale with additional threads. For pbzip2 we compress a 200MB .iso file. Our experiments ran on dual 8-core Intel Xeon E5-2630v3 2.4 GHz CPUs with 128 GB RAM. We compiled all benchmarks using LLVM 3.5.1 with -O3 optimizations.

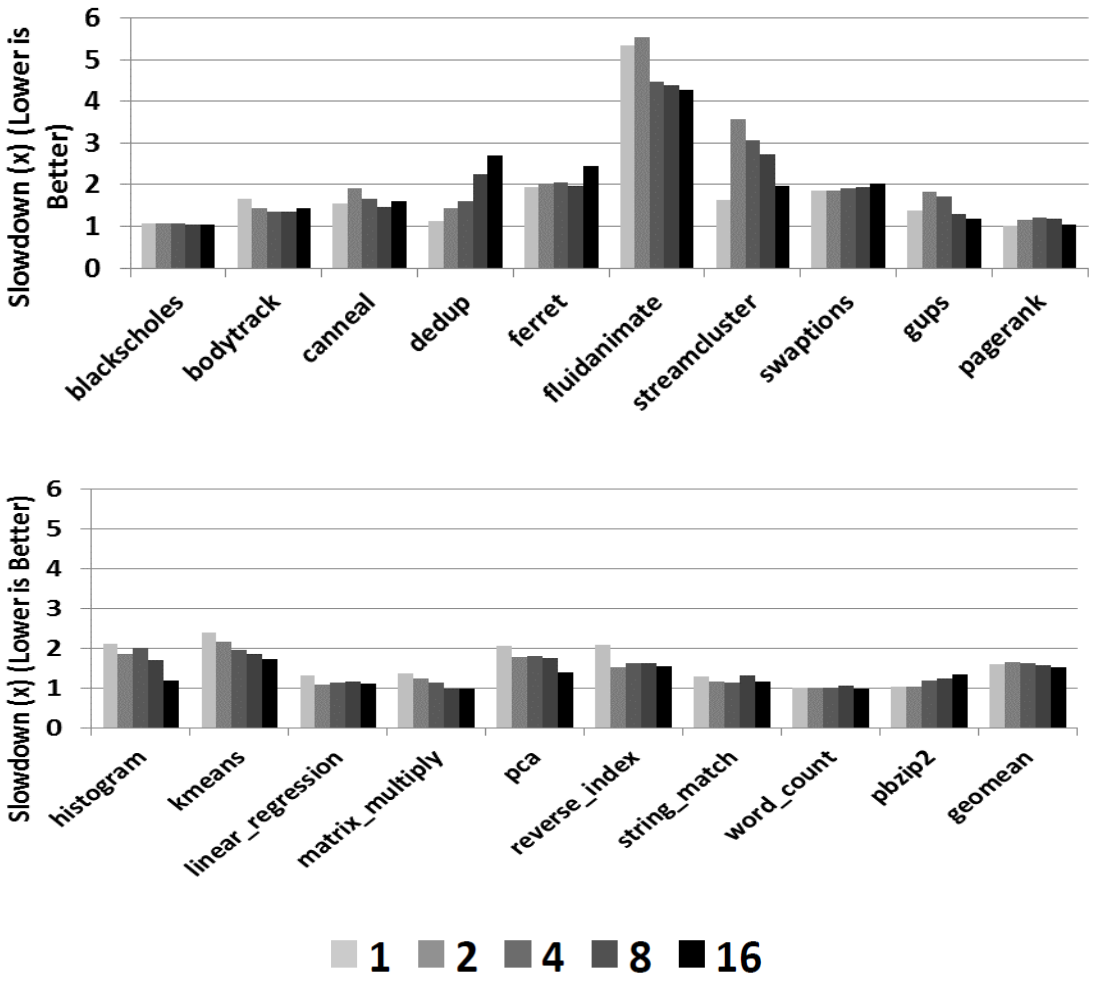

Figure 7.4: Runtime performance results for SOFRITAS across 1-16 thread counts

Figure 7.4 presents the runtime slowdown of SOFRITAS over pthreads. For each thread count, we normalize to the pthreads execution for the same thread count. Over all thread counts, the average runtime slowdown is 1.59x. Larger runtime slowdowns can generally be attributed to frequent ordering, such as barriers or condition waits – fluidanimate, and streamcluster both perform a considerable number of batch releases at the end of OFRs. Although SOFRITAS has highly-optimized batch releases, clearing the thread-local shadow spaces too frequently can be detrimental to performance. For many benchmarks, SOFRITAS provides strong atomicity guarantees at less than 2x slowdown.

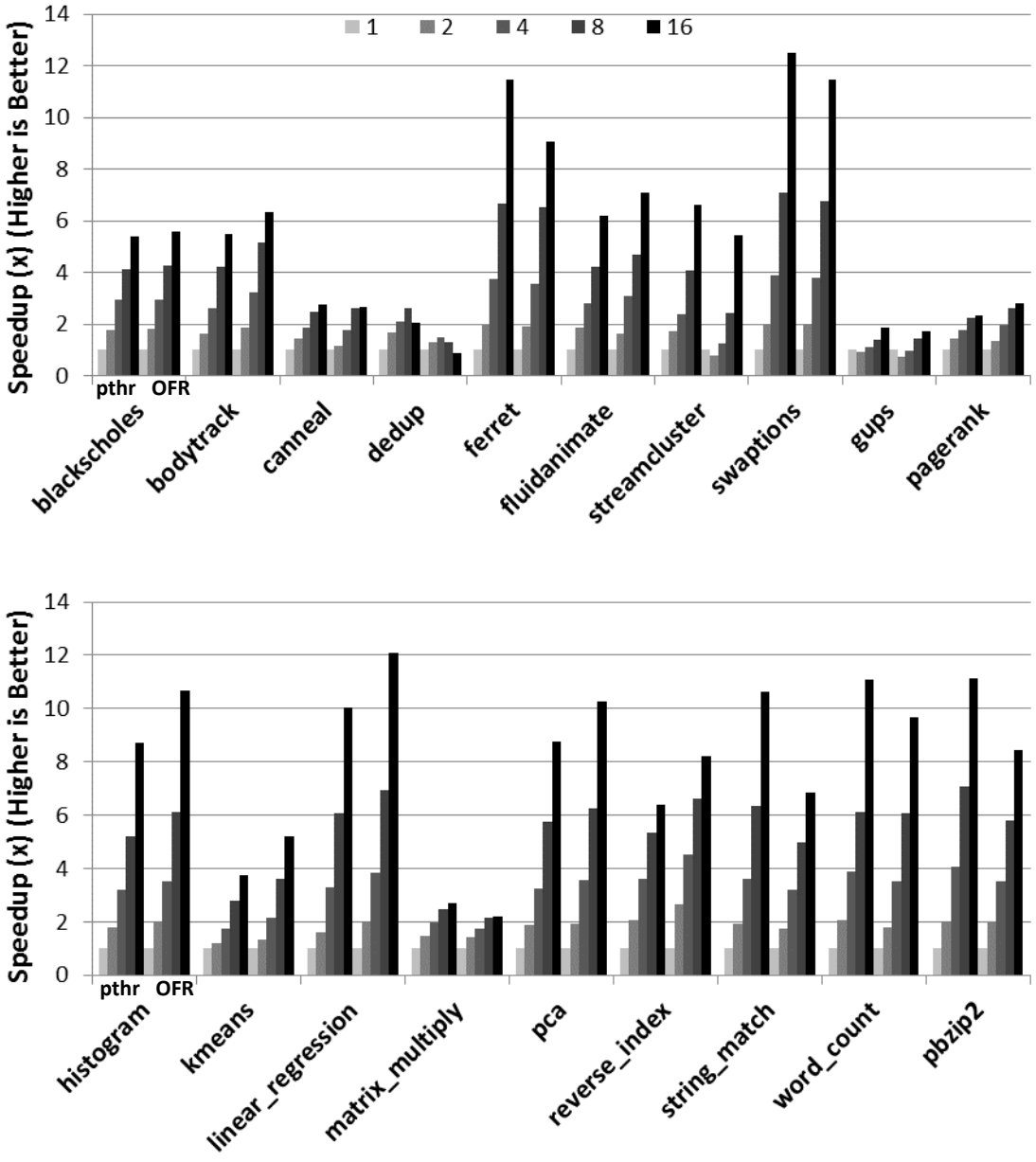

Figure 7.5: Scalability of benchmarks under pthreads and SOFRITAS

Figure 7.5 compares the scalability of SOFRITAS with pthreads. We show the scalability of each application using both pthreads and SOFRITAS. Each pthreads bar is normalized to the single-threaded execution using pthreads, and each SOFRITAS bar is normalized to the single-

threaded execution using SOFRITAS. For many benchmarks, SOFRITAS scales similarly to pthreads, as can be seen in the matching bar clusters.

For all the 19 benchmarks, SOFRITAS provides both increased atomicity and a parallel speedup over the single-threaded pthreads baseline. Although the absolute speedup using SOFRITAS is not as large as the speedup using an expert-synchronized pthreads implementation, SOFRITAS yields some performance benefits from parallel execution for many benchmarks.

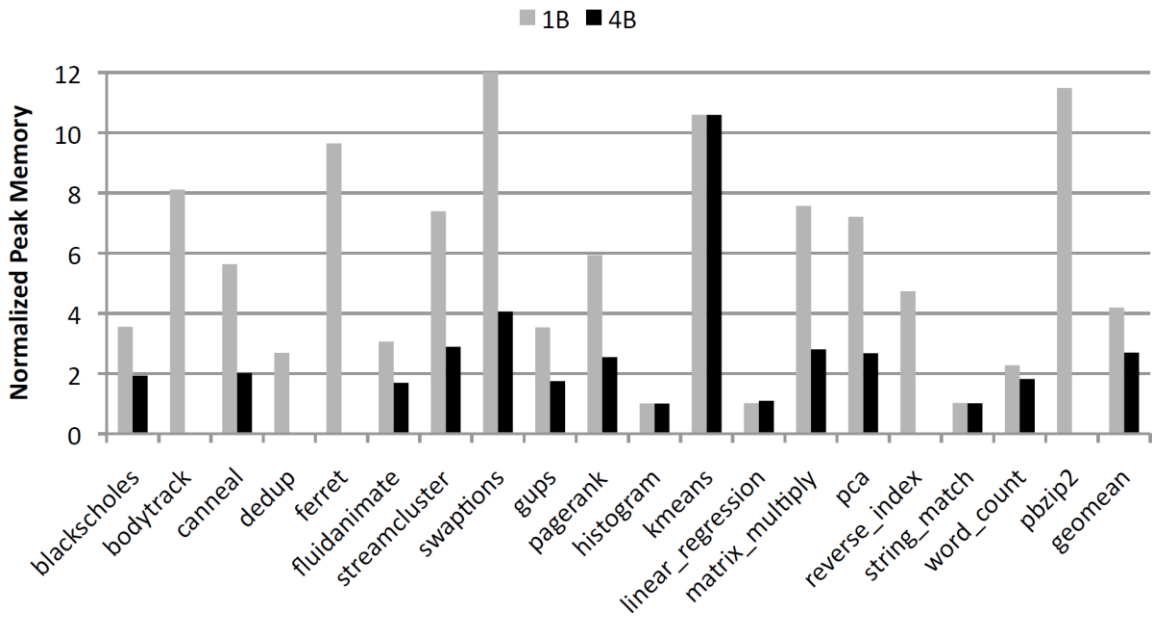

Figure 7.6: Memory overheads for SOFRITAS with 1 byte and 4 byte mappings

Figure 7.6 reports the memory overhead for SOFRITAS compared to pthreads execution with both using 16 threads. Memory usage is recorded using the getrusage system call which reports the maximum resident set size during the application's execution. The 1B bars report the overhead for using a 1-byte-per-lock mapping, which is necessary for benchmarks that share byte-sized data. In many cases, SOFRITAS can use a wider-granularity mapping of 4-bytes per lock, as shown in the 4B bars. The benchmarks without 4B bars (bodytrack, dedup, ferret, reverse index) did not run correctly with a 4- byte mapping.

SOFRITAS generally consumes less space with the 4B mapping (2.70x on average) than with the 1B mapping (4.19x on average). The exceptions fall into two cases. In benchmarks with heaps under 50MB, like kmeans, there is not much SOFRITAS metadata to begin with, and the fixed costs of SOFRITAS's other internal data structures magnify the memory overhead. An analogous situation arises in benchmarks with large memory regions mapped for I/O, such as histogram, linear regression and string match, as there is comparatively little heap on which the 4B mapping can save space. Moreover, the SOFRITAS runtime system uses simple bump-pointer allocation to provide pages to the tcmalloc memory allocator. In future work, this allocation scheme can be improved to maintain a free page list instead, which should reduce memory overheads further.

7.2.1 Comparison to Other Region-Based Models

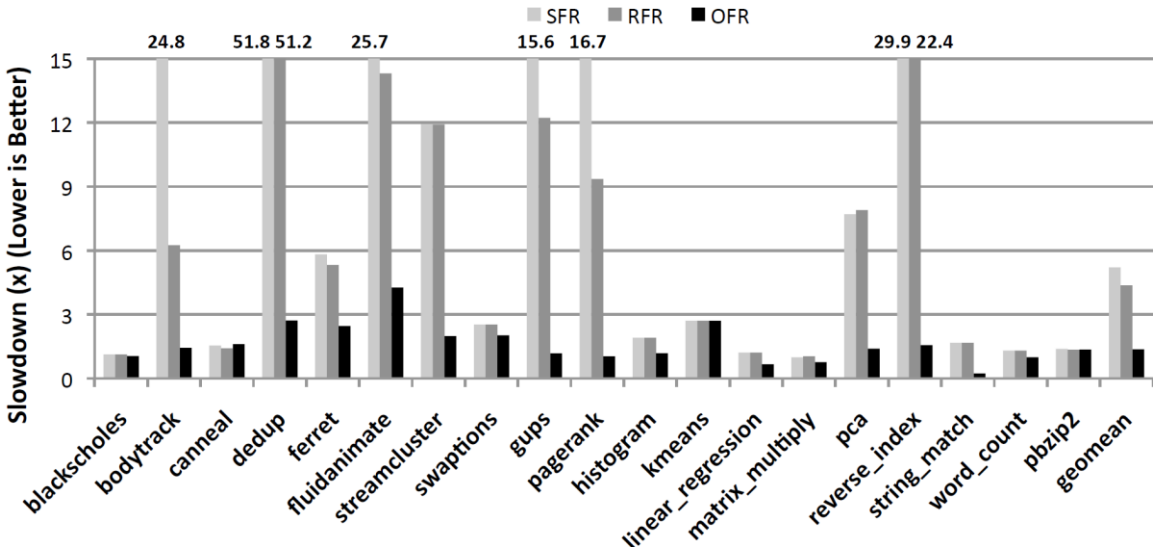

Figure 7.7: Comparison of SFR, RFR, and OFR models as implemented by SOFRITAS

Although SOFRITAS was designed to implement OFR serializability, the framework provided by SOFRITAS can also be used to implement SFR and RFR atomicity. Figure 7.7 provides a comparison of the performance of SFR, RFR, and OFR serializability as normalized to the pthreads baseline. We note that the implementation of SFRs and RFRs in SOFRITAS does not

include optimizations that have been used in existing work, such as Valor [8]. These optimizations allow deferred checking of reads that occur in release-free regions. With these optimizations, the performance comparison would not be quite as stark.

As shown, OFRs (as implemented in SOFRITAS) entail lower runtime slowdowns than both SFRs and RFRs for all benchmarks. On benchmarks with frequent lock acquires and releases, OFRs provide both more atomicity and lower runtime slowdowns. The higher performance cost for SOFRITAS using SFRs and RFRs stem from a few sources. First, compiler optimizations are limited in scope for SFRs and RFRs compared to OFRs because locks must be reacquired after each pthread mutex operation. Second, SFRs and RFRs require more batch lock releases than OFRs. SFRs require a batch release at each lock acquire, and both SFRs and RFRs require a batch release at each lock release. On average, SFRs exhibit 5.21x slowdown and RFRs exhibit 4.36x slowdown over all benchmarks. Valor recently implemented RFR-based atomicity for Java with only 1.99x average slowdown for RFRs and 2.04x slowdown for SFRs. Valor's results come from a managed runtime, making them hard to compare directly to our unmanaged C/C++ implementation. We note, however, that while SOFRITAS's RFR and SFR run time costs are much higher than Valor's, SOFRITAS's 1.59x average slowdown is on par with Valor and SOFRITAS's OFRs provide a coarser atomicity guarantee than RFRs. SFR and RFR models require the same number of atomicity and ordering constructs as pthreads and thus require similar programmer effort compared to OFRs.

7.2.2 Optimizations

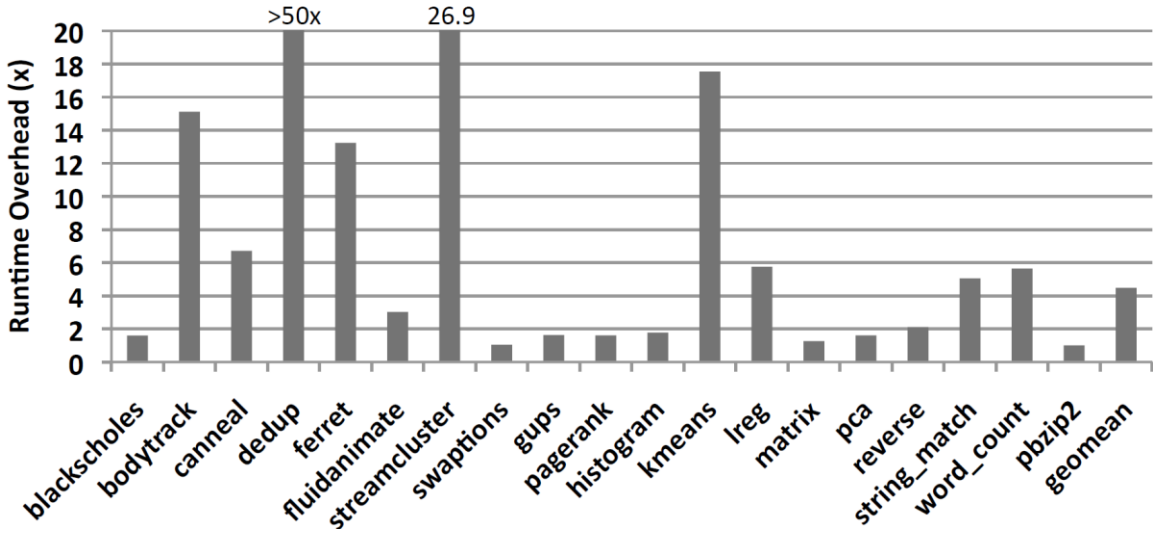

Figure 7.8: Overheads of using memset instead of madvise

As discussed in prior sections, SOFRITAS uses multiple low-level optimizations to reduce performance overheads. To efficiently release locks at OFR boundaries, SOFRITAS calls madvise instead of using memset. Figure 7.8 shows the overhead of using memset instead of madvise as normalized to the SOFRITAS baseline. On average, using memset incurs an overhead of 4.47x over the baseline SOFRITAS system. On dedup and streamcluster, the overheads of using memset are extremely high. streamcluster suffers from overheads of 26.9x, and our experiments on dedup timed out at 50x over the normal runtime for the application.

madvise provides a large performance gain over memset because memset always zeroes the page, but madvise does not always need to zero out the page. Instead, the operating system maps the madvised page to a read-only zero page. If the thread attempts to write to the madvised page, the operating system will on-demand allocate a new physical page and zero it out. However, if the thread does not write to a page that has been cleared, other threads can confirm that locks have been released without triggering a new page allocation (and subsequent zeroing) because the checks are read-only.

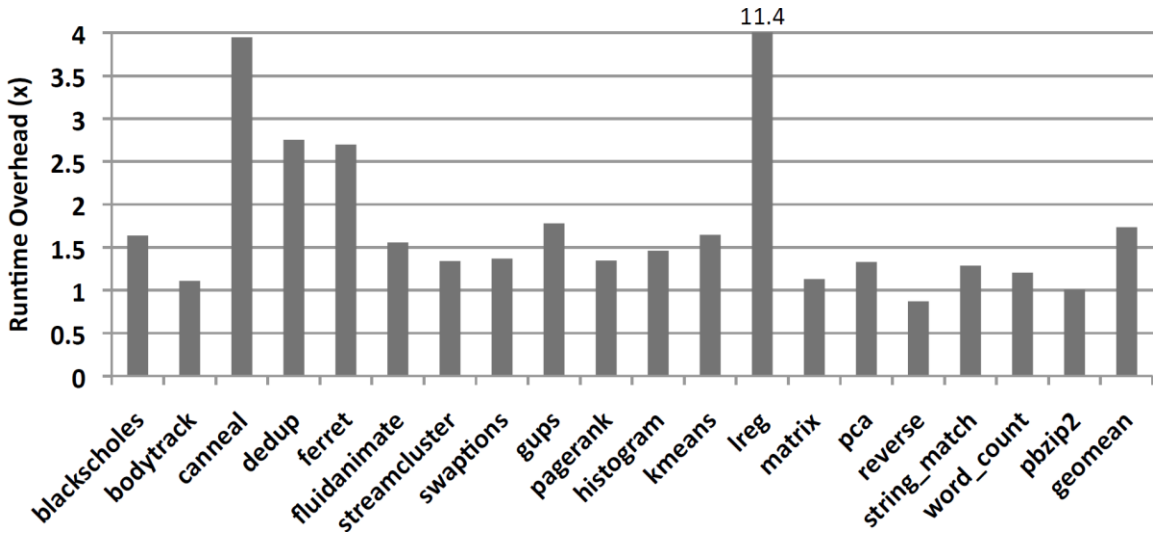

Figure 7.9: Overheads of not inlining lock checks

SOFRITAS also relies on efficient lock checks, which are much more common than lock acquires. SOFRITAS inlines lock checks for efficiency because frequent function calls can be expensive, especially when they involve saving and restoring registers on the stack. Figure 7.9 shows the overheads incurred by SOFRITAS when no lock checks are inlined. On average, SOFRITAS incurs a 1.73x overhead over the baseline system when no lock checks are inlined.

7.3 Hardware Support

Hardware support for ordering-free regions was initially designed for the ORCA system, which has multiple design flaws that have been corrected by SOFRITAS. As future work, we intend to design and test hardware-support for the SOFRITAS runtime system. As a precursor to this work, we examine the benefits of using hardware-support for the ORCA system.

The ORCA architecture simulator is based on the cache modules of the open-source PIN-based ZSim simulator [65]. Our baseline configuration is an 8-core system with coherent 32KB 8-way associative L1 caches, private 256KB 8-way L2 caches, and a shared 8MB 16-way L3 cache. All line sizes are 64B. The simulator models a simple prefetcher that fetches the next two cache lines on a miss in parallel with the execution. L1 cache hits take 1 cycle, remote L1 hits

15 cycles, L2 hits 10 cycles, L3 hits 35 cycles, and main memory 120 cycles. All other instructions take a single cycle. For our simulations, we use the simmedium inputs for PARSEC.

On each memory access, the simulator checks the lock cache for lock ownership information. Our baseline lock cache is 16KB, direct-mapped, with 64 lock states per line (16B lines). A CACTI [37] model 32nm of the lock cache reports an access time of 0.26ns and total access energy of 8.8pJ. The area of a lock cache is 76μm2, and lock caches make up 0.001% of the area of a four core Intel Sandy Bridge CPU. Because ORCA's per-byte locks occupy just 2 bits in the lock cache (an effective 4x compression ratio), a 16KB lock cache readily covers the 32KB data cache. Lock cache accesses proceed in parallel with data cache accesses.

Our main result is that our proposed hardware support enables efficient execution for ORCA. We simulated four different hardware and software configurations. Figure 7.10 plots the performance of each of these configurations on a simulated 4-core machine, normalized to a simulated execution of pthreads.

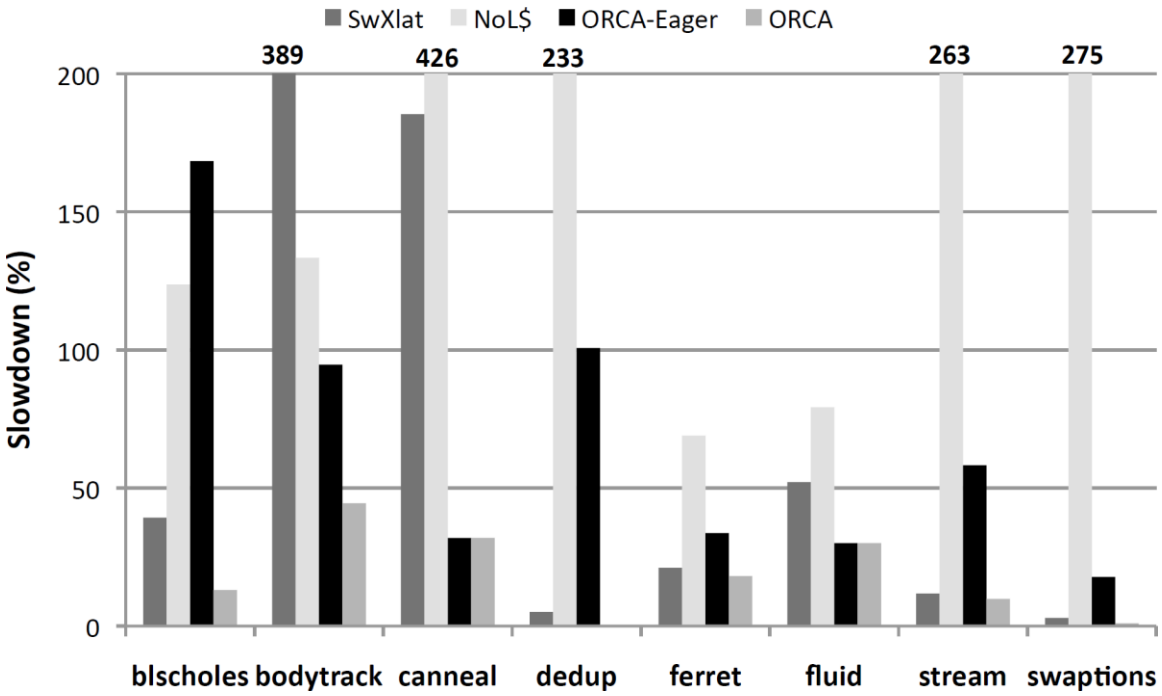

Figure 7.10: Overheads of ORCA configurations. SwXlat shows performance with flexible shadowspace trie. NoL$ shows performance without the lock cache. ORCA-Eager shows performance of ORCA with eager releases

The ORCA bar shows ORCA's performance with hardware support for address translation and with the lock cache. ORCA imposes a slowdown of just 18% on average compared to pthreads, with a worst-case slowdown of 44% (bodytrack). bodytrack requires comparatively more lock operations than other benchmarks. These lock operations lead to increased pressure in the lock cache and data cache as lock state must be updated frequently. canneal and fluidanimate exhibit similar behavior to a lesser extent.

Lazy releases can be used to improve performance and atomicity in exchange for a few false exceptions. The ORCA-Eager bar demonstrates the performance of an implementation of ORCA that eagerly releases locks at the end of every OFR. Our modeling of ORCA-eager is optimistic in that it assumes no overhead for tracking what locks need to be released, modeling only the cost of releasing them. Even given this optimistic modeling, ORCA-Eager exhibits an average overhead of 61% compared to the baseline and 40% compared to ORCA with lazy

releases. 98% of the OFR exceptions raised by ORCA with lazy releases result from real violations of OFR serializability. SOFRITAS improves upon the naïve implementation of eager releases used by ORCA, negating much of the performance overhead of eager releases.

The SwXlat configuration demonstrates the cost of software lock address translation. In this configuration, both the runtime system and the lock cache map memory addresses to lock addresses using the translation trie. blackscholes and fluidanimate are particularly affected by the extra cache pollution generated by trie accesses.

NoL$ shows the performance of ORCA with hardware address translation but without the lock cache. The data demonstrates that the lock cache is essential in a high-performance ORCA implementation. Removing the lock cache greatly increases ORCA's performance overhead to 169% on average. The lock cache gives swaptions, canneal and streamcluster an especially noticeable performance boost as these workloads suffer from a high data cache read miss rate without the lock cache. ORCA pollutes the data cache with its locks due to frequent ownership checks on held locks. With the lock cache, these ownership checks are removed, decreasing pressure on the data cache.

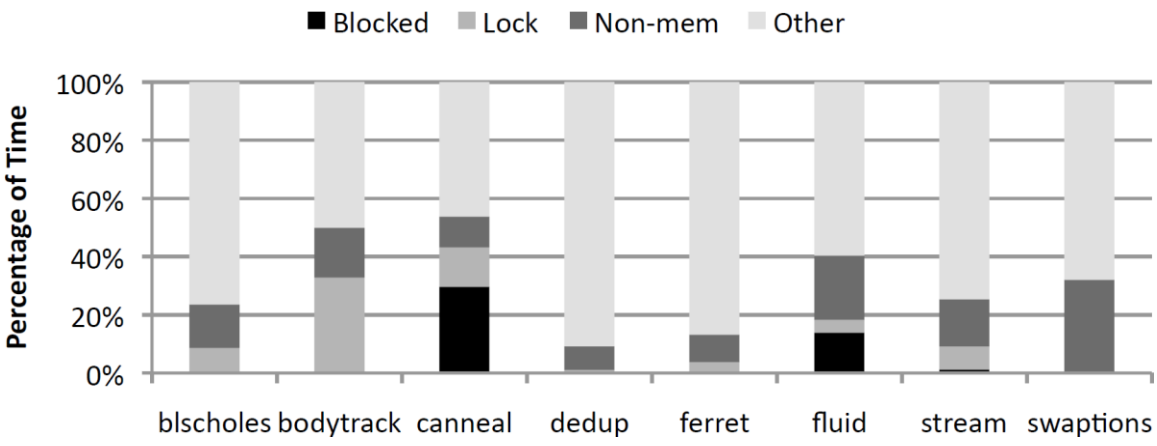

Figure 7.11: Overview of cycle overheads for ORCA

Figure 7.11 provides a breakdown of the overheads for ORCA. Blocked time is when a thread is serialized waiting for a lock held by another thread. Only canneal and fluidanimate exhibit non-trivial serialization, due to conflicting array accesses. Lock time is spent waiting for

data-cache accesses to ORCA locks. Such accesses are caused by lock cache misses and when a lock needs to be acquired or released. bodytrack exhibits the most lock overhead due to frequent lock acquires and releases on its shared data structures. Non-memory time is the cost of executing non-memory instructions, which is not significantly affected by ORCA. The remaining time (Other) is spent in the memory hierarchy due to program data cache accesses or indirect pressure caused by ORCA's cache pollution.

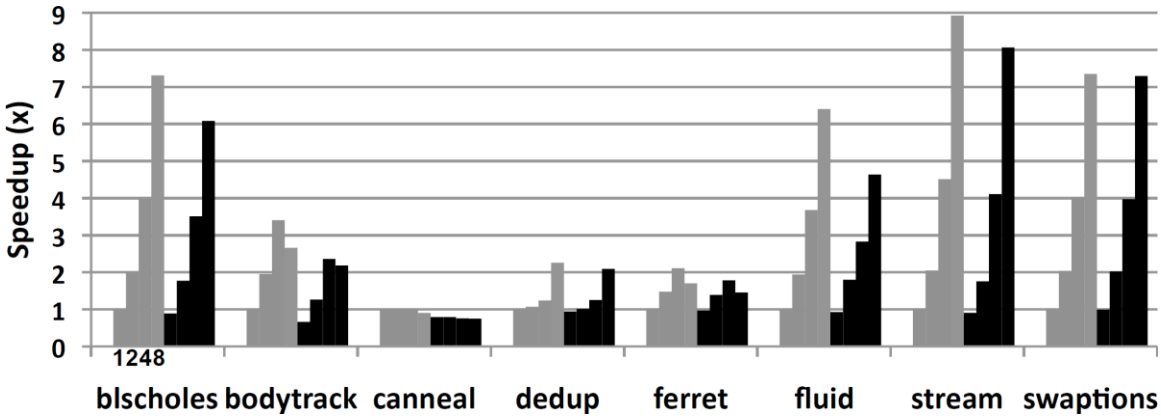

Figure 7.12: Scalability of ORCA across 1-8 threads

We measured how ORCA's performance scales with additional cores, showing that ORCA is scalable. Figure 7.12 shows the simulated runtime of the pthreads and ORCA versions of each benchmark with one to eight threads, normalized to serial pthreads execution. This graph demonstrates that ORCA can readily exploit parallelism, scaling as well as pthreads up to eight threads.

With a single thread, ORCA suffers a performance penalty of 14% on average compared to pthreads. This single-thread overhead can be largely attributed to increased pressure in the data cache due to first-time lock acquires and lock cache misses, both of which require loading a lock into the data cache. With two threads, ORCA provides an average speedup of 1.41x over the serial baseline. With four and eight threads, ORCA's average speedup increases to 2.25x and 3.09x, respectively. In the best case, streamcluster has a parallel speedup of 8x given eight

threads. Thus, ORCA can provide speedups through parallel execution while simultaneously providing increased atomicity.

Figure 7.12 also shows that ORCA's performance scales very similarly to pthreads. Up to eight threads, benchmarks have similar contours for both pthreads and ORCA. canneal does not scale well with either ORCA or pthreads because additional threads in canneal are used to generate a more precise result with more iterations of the genetic algorithm rather than reducing the amount of work done by each thread. Further, canneal's random access patterns admit little cache locality. ferret does not scale to 8 threads because this benchmark uses 8 threads per intermediate pipeline stage for a total of 35 threads. This overprovisioning of threads causes cache interference as the lock cache stores a per-thread lock status that cannot be shared across threads mapped to the same core. Thus, overprovisioning threads causes thrashing in the lock cache. Tagging lock cache lines with thread IDs could help alleviate this issue.

7.4 Discussion

Across three separate implementations of OFR serializability, a few results remain constant. The scalability of parallel applications is not limited by OFR serializability. Up to 16 threads, parallel applications continue to scale under OFR serializability. The overheads for enforcing OFR serializability are low enough that the scalability provided by parallel execution yields a speedup over serial execution even at low thread counts. This means that a programmer can readily exploit parallel execution using an OFR programming model. OFR serializability can make writing a parallel application easier than it is using pthreads, so novice parallel programmers may be more able to exploit parallelism via an OFR programming model than they could with a standard parallel programming model.

Hardware support is clearly beneficial for lowering the runtime performance impact of OFR serializability. Even with ORCA's less than ideal runtime system, hardware support lowered

the overheads of OFR serializability to just 18%. With a more efficient software runtime system like SOFRITAS, the overheads with hardware support should be even lower.

The overheads of a software-only implementation are low due to a number of small optimizations that add up to large performance gains. An initial, unoptimized version of SOFRITAS shows 100x overheads on the swaptions benchmark. With an optimized shadowspace layout, aggressive inlining of lock checks, the use of madvise for eager, batch releases, and additional compiler and runtime system optimizations, the overhead of SOFRITAS on swaptions was lowered to approximately 2x. This order-of-magnitude decrease was possible due to the combination of these optimizations.

8 RELATED WORK

Section 2 discussed related work that can be considered background information regarding ordering-free regions. In this section, we discuss work that is related to ordering-free regions.

8.1.1 DATA-CENTRIC SYNCHRONIZATION

Data-centric synchronization schemes explicitly associate locks with data and then assure that this locking discipline is automatically enforced. In some systems [15, 73, 74] a programmer specifies the variable-to-lock association. This association can also be inferred [40] at the risk of missing synchronization. Data-centric synchronization provides atomicity at the granularity of function calls, which is sufficient for many critical sections but not all, e.g. the queue implementation in the PARSEC dedup benchmark [16].

8.1.2 TRANSACTIONAL MEMORY

Transactional memory (TM) systems leverage programmer-specified atomic blocks [36, 69] that can be implemented via optimistic or pessimistic concurrency [26, 54]. Like conventional locking, programming with TM involves incrementally strengthening a program's atomicity. Transactional memory can ease the task of ensuring that code is correctly synchronized by ensuring that all memory locations are atomic within the bounds of a transaction. However, the programmer must still properly place the start and end of each transaction to cover the instructions that are required for the desired atomicity. Transactional memory increases the width of regions checking for conflicts over all memory locations rather than implicitly associating locks with memory locations. Consider the examples provided in Figure 2.1. In this example, naively replacing locks with transaction boundaries will not prevent the bug because the critical region is simply too small to provide the required atomicity.

The TCC [34] and Automatic Mutual Exclusion (AME) [39] systems place all code inside transactions, providing coarse-grained atomicity. However, AME and TCC target new programming models (task parallel and parallelization of sequential code, respectively) instead of providing stronger guarantees for existing multithreaded code as OFR serializability does. Both schemes employ weaker notions of serializability than OFR serializability and incur additional complexity due to the use of always-on optimistic concurrency which complicates I/O and other system calls.

8.1.3 COOPERATIVE CONCURRENCY

Cooperative concurrency [78, 79] systems add yield annotations to a program to document where thread interference can arise. Cooperability provides a sound summary of the side effects of a program's existing synchronization but does not automatically enforce atomicity guarantees.

8.1.4 PROGRAM SYNTHESIS

Techniques for program synthesis of parallel programs [12, 71, 75] often operate by refining overly-coarse atomicity under the guidance of programmer-specified proofs or invariants. OFR serializability's dynamic approach can scale to at least medium size parallel programs like PARSEC benchmarks, beyond the scope that synthesis systems support.

8.1.5 BARRIER INFERENCE

Ordering-free regions infer the required atomicity for a parallel program, but the algorithm assumes that the ordering synchronization (condition waits, barrier waits, thread start and join) is correct. Prior work has attempted to infer barrier synchronization in parallel applications [1,81]. These works examine the structure of a program and attempt to infer where ordering synchronization should be placed. This work is complementary to the atomicity inference performed by ordering-free regions.

9 CONCLUSIONS AND FUTURE WORK

This dissertation has examined the use of ordering-free region serializability in parallel programming. Ordering-free regions provide more atomicity than previous parallel programming models, including sequential consistency, synchronization-free regions, and release-free regions, by extending the region of atomicity between consecutive ordering-free constructs. Ordering-free regions are based on the 2PL model of serialization, and the theoretical guarantees provided by ordering-free region serializability are at least as strong as those provided by conflict serializability. We discussed the implementation tradeoffs inherent in implementing a runtime system that enforces ordering-free region serializability and introduced two such systems: ORCA and SOFRITAS. With ORCA, hardware support was introduced to accelerate the lock checks performed frequently in order to enforce ordering-free region serializability. We examined the usability of ordering-free regions as compared to existing programming models, like pthreads, and found that using ordering-free regions was at least as easy as writing parallel programs with existing models. In many cases, ordering-free regions provide benefits over conventional models, including the ability to easily find and fix data-races and atomicity violations. In a small user study, we found that novice programmers were more likely to correctly use ordering-free region exceptions to fix a buggy parallel program than to use data-race reports to accomplish the same task. We measured the performance overheads of three systems for enforcing ordering-free region serializability and found that the performance overheads were low overall. The scalability of parallel applications under ordering-free region serializability is comparable to the scalability of the same applications under a conventional parallel programming model.

Ordering-free regions offer an opportunity to ease the burden of writing parallel applications, especially for novice parallel programmers. In today's world, programmers not only need to be able to write applications for multicore processors but also manage the complexity of writing applications for heterogeneous systems that include graphics processors, machine learning accelerators, field programmable gate arrays, and other complex hardware. Helping

programmers manage this complexity is of the utmost importance. Of course, everyone thinks that their own model for managing the complexity of parallel programming is the necessary panacea. In any case, it seems clear that conventional models are too complex and need to be replaced with a safer methodology. Ordering-free regions are a step in the right direction of making parallel programming easier and more reliable.

This dissertation has examined many facets of parallel programming with ordering-free regions. There is always more work to be done, and the following subsections discuss directions for future work on ordering-free regions.

9.1 Hardware Support for SOFRITAS

The SOFRITAS software-only system demonstrated that OFR serializability could be provided in software at relatively low runtime overheads of only 1.59x. Unfortunately, 1.59x is still a high overhead in a language like C++ where programmers expect little to no overhead from the runtime system. To further reduce the overhead of OFR serializability, hardware support should be designed to fit the SOFRITAS lock and shadowspace designs.

Hardware support for SOFRITAS should accomplish a few goals. First, the hardware support should aim to avoid polluting the data cache with locks. In a software-only system, locks are data, and those locks take up space that can otherwise be used for program data in the cache hierarchy. Second, the hardware support should reduce the register pressure and complexity of the instrumentation. In the current design, the SOFRITAS compiler uses additional registers to implement lock checks and acquires. With ISA support, those additional register uses can be avoided by using special instructions and possibly special registers to support lock checks and acquires.

ORCA's hardware is a reasonable starting point for designing hardware support for SOFRITAS. The ORCA lock cache is designed to support 2-bit lock states and was the inspiration for SOFRITAS's thread-local lock shadowspaces. In ORCA, loading a line into the

lock cache required loading a byte of memory for each 2 bits being loaded into the lock cache. With SOFRITAS, a single load can retrieve (at least) 64 bits of lock state from the thread-local shadowspace. As noted in prior sections, the hardware address translation used by ORCA would need to be updated to account for the less-rigid shadowspace layout that SOFRITAS assumes.

Aside from updating the ORCA hardware to interact with SOFRITAS, other opportunities exist in designing hardware support for SOFRITAS. The lock states used by SOFRITAS correspond closely with existing MESI cache coherence protocols. Prior work has explored using cache coherence protocols to accelerate data-race detection [25,55,57,82]. Ordering-free region serializability may benefit from similar optimizations. By attaching lock states to the coherence protocol, SOFRITAS may be able to optimize lock acquires for common cases like shared reads.

9.2 Further User Study

As part of this dissertation, we have rewritten a number of parallel applications to use ordering-free regions instead of conventional programming models. However, we have not examined the process and potential benefits of writing parallel programs from scratch with ordering-free regions. Recent work has shown the benefits of developing parallel applications from scratch with transactional memory in mind, leading to median speedups of 4.1x over naïve applications that use transactional memory [60]. Parallel applications may yield similar speedups when developed with ordering-free regions in mind.

Aside from the potential performance benefits, further study is needed to assess whether or not region-based parallel programming models truly ease the process of writing parallel applications from scratch. We intend to perform a large-scale user study on writing parallel applications from scratch with various parallel programming models, including ordering-free regions. By examining how novice programmers write parallel programs from scratch, we hope to understand how models like region-based parallel programming can make the task of writing parallel applications easier.

BIBLIOGRAPHY

[1] Alexander Aiken and David Gay. 1998. Barrier inference. In Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '98). ACM, New York, NY, USA, 342-354.

[2] Sarita V. Adve and Kourosh Gharachorloo. 1996. Shared Memory Consistency Models: A Tutorial. Computer 29, 12 (December 1996), 66-76.

[3] Vignesh Balaji, Brandon Lucia, and Radu Marculescu. Overcoming the data- flow limit on parallelism with structural approximation. In WAX 2016, 2016.

[4] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. 2010. CoreDet: a compiler and runtime system for deterministic multithreaded execution. In Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems (ASPLOS XV). ACM, New York, NY, USA, 53-64.

[5] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. 2001. Composing high-performance memory allocators. In Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation (PLDI '01). ACM, New York, NY, USA, 114-124.

[6] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. 1987. Concurrency Control and Recovery in Database Systems. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[7] Christian Bienia. Benchmarking Modern Multiprocessors. PhD thesis, Princeton University, January 2011.

[8] Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Brandon Lucia. 2015. Valor: efficient, software-only region conflict exceptions. In Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015). ACM, New York, NY, USA, 241-259.

[9] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: java benchmarking development and analysis. In Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA '06). ACM, New York, NY, USA, 169-190.

[10] Colin Blundell, Milo M.K. Martin, and Thomas F. Wenisch. 2009. InvisiFence: performance-transparent memory ordering in conventional multiprocessors. In Proceedings of the 36th annual international symposium on Computer architecture (ISCA '09). ACM, New York, NY, USA, 233-244.

[11] Hans J. Boehm and Sarita Adve. Foundations of the c++ concurrency memory model. In Proceedings of the SIGPLAN 2008 Conference on Programming Language Design and Implementation, PLDI ’08, 2008.

[12] Matko Botinčan, Mike Dodds, and Suresh Jagannathan. 2013. Proof-Directed Parallelization Synthesis by Separation Logic. ACM Trans. Program. Lang. Syst. 35, 2, Article 8 (July 2013), 60 pages.

[13] Gabriel Bracha and Sam Toueg. Distributed deadlock detection. Distributed Computing, 2(3):127–138, 1987.

[14] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. A randomized scheduler with probabilistic guarantees of finding bugs. In Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems (ASPLOS XV). ACM, New York, NY, USA, 167-178.

[15] Luis Ceze, Pablo Montesinos, Christoph von Praun, and Josep Torrellas. Colorama: Architectural support for data-centric synchronization. In Proceedings of the 13th Symposium on High-Performance Computer Architecture, pages 133–144, Feb. 2007.

[16] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. Bulksc: Bulk enforcement of sequential consistency. In Proceedings of the 34th Annual International Symposium on Computer Architecture, June 2007.

[17] Luis Ceze, Christoph von Praun, Calin Cascaval, Pablo Montesinos, and Josep Torrellas. Concurrency control with data coloring. In Proceedings of the 2008 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness, pages 6–10, 2008.

[18] Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. Journal on Software Testing, Verification & Reliability, 13(4):220–227, 2003.

[19] Madan Das, Gabriel Southern, and Jose Renau. Section based program analysis to reduce overhead of detecting unsynchronized thread communication. In Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP), pages 283–284, 2015.

[20] Christian DeLozier, Joseph Devietti, and Milo M. K. Martin. 2014. MAMA: Mostly Automatic Management of Atomicity. In 5th Workshop on Determinism and Correctness in Parallel Programming (WoDet ’14).

[21] Christian DeLozier, Ariel Eizenberg, Shiliang Hu, Gilles Pokam, and Joseph Devietti. 2017. TMI: thread memory isolation for false sharing repair. In Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17). ACM, New York, NY, USA, 639-650.

[22] Christian DeLozier, Ariel Eizenberg, Brandon Lucia, and Joseph Devietti. 2018. SOFRITAS: Serializable Ordering-Free Regions for Increasing Thread Atomicity Scalably. In Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18). ACM, New York, NY, USA, 286-300.

[23] Christian DeLozier, Yuanfeng Peng, Ariel Eizenberg, Brandon Lucia, and Joseph Devietti. Orca: Ordering-free regions for consistency and atomicity. Technical Report MS-CIS-16-01, University of Pennsylvania, May 2016.

[24] Joseph Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. Hardbound: Architectural support for spatial safety of the c programming language. In Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, Mar. 2008.

[25] Joseph Devietti, Benjamin P. Wood, Karin Strauss, Luis Ceze, Daniel Grossman, and Shaz Qadeer. Radish: Always-on sound and complete ra detection in software and hardware. In Proceedings of the 39th Annual International Symposium on Computer Architecture, ISCA '12, pages 201–212, Washington, DC, USA, 2012. IEEE Computer Society.

[26] Dave Dice, Ori Shalev, and Nir Shavit. 2006. Transactional locking II. In Proceedings of the 20th international conference on Distributed Computing (DISC'06), Shlomi Dolev (Ed.). Springer-Verlag, Berlin, Heidelberg, 194-208.

[27] Y. Duan, D. Koufaty and J. Torrellas. SCsafe: Logging sequential consistency violations continuously and precisely. 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), Barcelona, 2016, pp. 249-260.

[28] Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Daniel Grossman, and Hans J. Boehm. Ifrit: Interference-free regions for dynamic data-race detection. In Proceedings of the 27th SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Application (OOPSLA), pages 467–484, Oct. 2012.

[29] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2007. Goldilocks: a race and transaction-aware java runtime. In Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07). ACM, New York, NY, USA, 245-255.

[30] Engadget. Intel announces Edison: a 22nm dual-core PC the size of an SD card, Jan. 2014. http://www.engadget.com/2014/01/06/intel-edison/.

[31] Cormac Flanagan and Stephen N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In Proceedings of The 31st ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL), pages 256–267, Jan. 2004.

[32] Cormac Flanagan and Stephen N. Freund. The roadrunner dynamic analysis framework for concurrent programs. In Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, pages 1–8, 2010.

[33] Cormac Flanagan, Stephen N. Freund, and J. Yi. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In Proceedings of the SIGPLAN 2008 Conference on Programming Language Design and Implementation, PLDI ’08, pages 293–303, 2008.

[34] Sanjay Ghemawat and Paul Menage. TCMalloc: Thread-Caching Malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html

[35] Lance Hammond, Brian D. Carlstrom, Vicky Wong, Ben Hertzberg, Mike Chen, Christos Kozyrakis, and Kunle Olukotun. 2004. Programming with transactional coherence and consistency (TCC). In Proceedings of the 11th international conference on Architectural support for programming languages and operating systems (ASPLOS XI). ACM, New York, NY, USA, 1-13.

[36] Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional memory: architectural support for lock-free data structures. In Proceedings of the 20th annual international symposium on computer architecture (ISCA '93). ACM, New York, NY, USA, 289-300.

[37] Hewlett Packard Development Company, L.P. HP Labs: CACTI. http://quid.hpl.hp.com:9081/cacti.

[38] International Standard ISO/IEC 14882:2011. Programming Languages – C++. International Organization for Standards, 2011.

[39] Michael Isard and Andrew Birrell. 2007. Automatic mutual exclusion. In Proceedings of the 11th USENIX workshop on Hot topics in operating systems (HOTOS'07), Galen Hunt (Ed.). USENIX Association, Berkeley, CA, USA, , Article 3 , 6 pages.

[40] Stefan Kempf, Ronald Veldema, and Michael Philippsen. 2013. Compiler-Guided identification of critical sections in parallel code. In Proceedings of the 22nd international conference on Compiler Construction (CC'13), Ranjit Jhala and Koen Bosschere (Eds.). Springer-Verlag, Berlin, Heidelberg, 204-223.

[41] Ali Kheradmand, Baris Kasikci, and George Candea. Lockout: Efficient testing for deadlock bugs. In WoDet, 2014.

[42] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Transactions on Computers, C-28(9):690–691, Sept. 1979.

[43] Tongping Liu and Emery D. Berger. 2011. SHERIFF: precise detection and automatic mitigation of false sharing. In Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications (OOPSLA '11). ACM, New York, NY, USA, 3-18.

[44] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. 2007. MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles (SOSP '07). ACM, New York, NY, USA, 103-116.

[45] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. 2006. AVIO: detecting atomicity violations via access interleaving invariants. In Proceedings of the 12th international conference on Architectural support for programming languages and operating systems (ASPLOS XII). ACM, New York, NY, USA, 37-48.

[46] Brandon Lucia and Luis Ceze. Finding concurrency bugs with context-aware communication graphs. In Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, pages 553–563, Nov. 2009.

[47] Brandon Lucia, Luis Ceze, and Karin Strauss. Colorsafe: Architectural support for debugging and dynamically avoiding multi-variable atomicity violations. In Proceedings of the 37th Annual International Symposium on Computer Architecture, ISCA '10, pages 222–233, New York, NY, USA, 2010. ACM.

[48] 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 Proceedings of the 37th Annual International Symposium on Computer Architecture, ISCA '10, pages 210–221, New York, NY, USA, 2010. ACM.

[49] Brandon Lucia, Joseph Devietti, Karin Strauss, and Luis Ceze. Atom-aid: Detecting and surviving atomicity violations. In Proceedings of the 35th Annual International Symposium on Computer Architecture, pages 277–288, June 2008.

[50] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI '05). ACM, New York, NY, USA, 190-200.

[51] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. 2012. An axiomatic memory model for POWER multiprocessors. In Proceedings of the 24th international conference on Computer Aided Verification (CAV'12), P. Madhusudan and Sanjit A. Seshia (Eds.). Springer-Verlag, Berlin, Heidelberg, 495-512.

[52] Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java memory model. In Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '05). ACM, New York, NY, USA, 378-391.

[53] Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. 2010. DRFX: a simple and efficient memory model for concurrent programming languages. In Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10). ACM, New York, NY, USA, 351-362.

[54] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. 2006. Autolocker: synchronization inference for atomic sections. In Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '06). ACM, New York, NY, USA, 346-358.

[55] Sang L. Min and Jong-Deok Choi. 1991. An efficient cache-based access anomaly detection scheme. In Proceedings of the fourth international conference on Architectural support for programming languages and operating systems (ASPLOS IV). ACM, New York, NY, USA, 235-244.

[56] Abdullah Muzahid, Shanxiang Qi, and Josep Torrellas. 2012. Vulcan: Hardware Support for Detecting Sequential Consistency Violations Dynamically. In Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45). IEEE Computer Society, Washington, DC, USA, 363-375.

[57] Abdullah Muzahid, Dario Suárez, Shanxiang Qi, and Josep Torrellas. 2009. SigRace: signature-based data race detection. In Proceedings of the 36th annual international symposium on Computer architecture (ISCA '09). ACM, New York, NY, USA, 337-348.

[58] Santosh Nagarakatte, Sebastian Burckhardt, Milo M.K. Martin, and Madanlal Musuvathi. 2012. Multicore acceleration of priority-based schedulers for concurrency bug detection. In Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12). ACM, New York, NY, USA, 543-554.

[59] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In Proceedings of the 39th Annual International Symposium on Computer Architecture, ISCA ’12, pages 189–200, Washington, DC, USA, 2012. IEEE Computer Society.

[60] Donald Nguyen and Keshav Pingali. 2017. What Scalable Programs Need from Transactional Memory. In Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17). ACM, New York, NY, USA, 105-118.

[61] Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. . . . and region serializability for all. In Presented as part of the 5th USENIX Workshop on Hot Topics in Parallelism, Berkeley, CA, 2013. USENIX.

[62] Chang-Seo Park and Koushik Sen. 2008. Randomized active atomicity violation detection in concurrent programs. In Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering (SIGSOFT '08/FSE-16). ACM, New York, NY, USA, 135-145.

[63] Xuehai Qian, Josep Torrellas, Benjamin Sahelices, and Depei Qian. 2013. Volition: scalable and precise sequential consistency violation detection. In Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems (ASPLOS '13). ACM, New York, NY, USA, 535-548.

[64] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. 2007. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA '07). IEEE Computer Society, Washington, DC, USA, 13-24

[65] Daniel Sanchez and Christos Kozyrakis. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. In Proceedings of the 40th Annual International Symposium on Computer Architecture, ISCA ’13, pages 475–486, New York, NY, USA, 2013. ACM.

[66] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER multiprocessors. In Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11). ACM, New York, NY, USA, 175-186.

[67] Aritra Sengupta, Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Milind Kulkarni. 2015. Hybrid Static–Dynamic Analysis for Statically Bounded Region Serializability. In Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15). ACM, New York, NY, USA, 561-575.

[68] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM 53, 7 (July 2010), 89-97.

[69] Nir Shavit and Dan Touitou. 1995. Software transactional memory. In Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing (PODC '95). ACM, New York, NY, USA, 204-213.

[70] L. A. Smith, J. M. Bull, and J. Obdrzálek. A parallel java grande benchmark suite. In Proceedings of SC2001, pages 8–8, Nov. 2001.

[71] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. 2008. Sketching concurrent data structures. In Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08). ACM, New York, NY, USA, 136-148.

[72] University of Michigan. Concurrency Bugs, 2012. https://github.com/jieyu/concurrency-bugs.

[73] Mandana Vaziri, Frank Tip, and Julian Dolby. 2006. Associating synchronization constraints with data in an object-oriented language. In Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '06). ACM, New York, NY, USA, 334-345.

[74] Mandana Vaziri, Frank Tip, Julian Dolby, Christian Hammer, and Jan Vitek. 2010. A type system for data-centric synchronization. In Proceedings of the 24th European conference on Object-oriented programming (ECOOP'10), Theo D'Hondt (Ed.). Springer-Verlag, Berlin, Heidelberg, 304-328.

[75] Martin Vechev, Eran Yahav, and Greta Yorsh. 2010. Abstraction-guided synthesis of synchronization. In Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '10). ACM, New York, NY, USA, 327-338.

[76] Thomas F. Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2007. Mechanisms for store-wait-free multiprocessors. In Proceedings of the 34th annual international symposium on Computer architecture (ISCA '07). ACM, New York, NY, USA, 266-277.

[77] Min Xu, Rastislav Bodík, and Mark D. Hill. 2005. A serializability violation detector for shared-memory server programs. In Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI '05). ACM, New York, NY, USA, 1-14.

[78] Jaeheon Yi, Tim Disney, Stephen N. Freund, and Cormac Flanagan. Cooperative types for controlling thread interference in java. In Proceedings of the 2012 International Symposium on Software Testing and Analysis, ISSTA 2012, pages 232–242, New York, NY, USA, 2012. ACM.

[79] Jaeheon Yi, Caitlin Sadowski, and Cormac Flanagan. Cooperative reasoning for preemptive execution. In Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming, PPoPP ’11, pages 147–156, New York, NY, USA, 2011. ACM.

[80] Jie Yu and Satish Narayanasamy. 2009. A case for an interleaving constrained shared-memory multi-processor. In Proceedings of the 36th annual international symposium on Computer architecture (ISCA '09). ACM, New York, NY, USA, 325-336.

[81] Yuan Zhang and Evelyn Duesterwald. 2007. Barrier matching for programs with textually unaligned barriers. In Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '07). ACM, New York, NY, USA, 194-204.

[82] Pin Zhou, Radu Teodorescu, and Yuanyuan Zhou. 2007. HARD: Hardware-Assisted Lockset-based Race Detection. In Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA '07). IEEE Computer Society, Washington, DC, USA, 121-132.