SOFRITAS: Serializable Ordering-Free Regions for Increasing Thread Atomicity Scalably

Christian DeLozier
delozier@cis.upenn.edu
University of Pennsylvania

Ariel Eizenberg
arieleiz@cis.upenn.edu
University of Pennsylvania

Brandon Lucia
blucia@andrew.cmu.edu
Carnegie Mellon University

Joseph Devietti
devietti@cis.upenn.edu
University of Pennsylvania

Abstract

Correctly synchronizing multithreaded programs is challenging and errors can lead to program failures such as atomicity violations. Existing strong memory consistency models rule out some possible failures, but are limited by depending on programmer-defined locking code. We present the new Ordering-Free Region (OFR) serializability consistency model that ensures atomicity for OFRs, which are spans of dynamic instructions between consecutive ordering constructs (*e.g.*, barriers), without breaking atomicity at lock operations. Our platform, Serializable Ordering-Free Regions for Increasing Thread Atomicity Scalably (SOFRITAS), ensures a C/C++ program's execution is equivalent to a serialization of OFRs by default. We build two systems that realize the SOFRITAS idea: a concurrency bug finding tool for testing called SofriTest, and a production runtime system called SoPro.

SofriTest uses OFRs to find concurrency bugs, including a multi-critical-section atomicity violation in memcached that weaker consistency models will miss. If OFRs are too coarse-grained, SofriTest suggests refinement annotations automatically. Our software-only SoPro implementation has high performance, scales well with increased parallelism, and prevents failures despite bugs in locking code. SoPro has an average overhead of just 1.59x on a single-threaded execution and 1.51x on sixteen threads, despite pthreads' much weaker memory model.

CCS Concepts • **Computer systems organization** → **Multicore architectures**; • **Software and its engineering** → **Runtime environments**;

Keywords atomicity, region serializability, data races, C, C++

1 Introduction

Following a decades-long trend toward pervasive parallelism, shared-memory multi-threaded programs, written in widespread languages like C, C++, and Java, are the applications in the cloud, mobile devices, and even embedded systems [20]. Nearly every programmer today must write parallel programs and there is an urgent need to make it simple to write efficient parallel code.

A system's memory consistency model crucially affects a system's performance and programmability. The memory models for Java [36], C++ [7], and various hardware architectures [35, 44, 46] permit aggressive optimization, but are complex and inaccessible to most programmers. Systems with a Sequentially Consistent (SC) model [6, 11, 37, 53] give sequential interleaving semantics to parallel executions, with interleaving at instruction granularity. Recent research in "strong consistency models" has followed a trend toward offering atomicity with ever-coarser region definitions, such as multi-instruction regions [37], loop-free regions [45], synchronization-free regions (SFRs) [33], or release-free regions (RFRs) [5, 58]. In general, offering atomicity at coarser granularity limits the possible thread interleavings of a program and thereby simplifies the task of writing correct code.

These existing systems guarantee that all executions exhibit region serializability, and throw an exception otherwise. This guarantee simplifies language semantics, but does not go much further because serializability is provided only for *programmer-demarcated* regions, which may be insufficient for correctness. If the programmer gets the region boundaries wrong, these prior systems cannot help.

In this work, we develop SOFRITAS, a new, software-only region-based memory consistency model. One of SOFRITAS's key contributions is to provide atomicity at a granularity much coarser than existing proposals: extending beyond SFRs and RFRs to *ordering-free regions* (OFRs) of code that are punctuated only by ordering constructs. We define ordering constructs as barrier wait, condition variable wait, and thread fork and join. Note that condition variable notify is not a synchronization operation in the C/C++ standard because of the possibility of spurious wakeups. Accordingly, SOFRITAS does not break atomicity on notify operations. With SOFRITAS, a program's behavior is equivalent to a serialization of atomically-executed OFRs (otherwise a precise exception is raised). We demonstrate two systems that utilize OFR atomicity: a testing tool SOFRITEST that finds new concurrency bugs that other strong memory consistency models cannot, and an always-on pure-software runtime for production SOPRO that uses OFR atomicity to automatically prevent concurrency bugs from manifesting as errors.

SOFRITEST assumes that ordering constructs are correctly placed by the programmer, and that every OFR should execute atomically. Since SOFRITEST's region atomicity is coarser than the critical sections defined by lock acquires and releases, the presence of locking operations in source code is no longer needed for atomicity. Eliminating any dependence on the correctness of locking code provides a significant benefit: **code can run correctly despite missing or incorrectly-placed locks**. This allows SOFRITEST to find high-level atomicity violation bugs (such as the one illustrated in Figure 1) that prior strong memory consistency models cannot.

Assuming that OFRs should execute atomically appears to be an empirical upper bound on the atomicity that real programs require – a dynamic analysis run on all inputs to all PARSEC benchmarks, and multiple runs of Apache, memcached and pbzip2 found no examples of code that required atomicity *coarser* than an OFR. However, OFR atomicity can sometimes be too coarse and programs may require RFR-, SFR- or even instruction-level atomicity to make progress. If necessary, a programmer using SOFRITEST can refine atomicity using annotations to harmonize with the program's atomicity requirements. There is of course a risk that refinement requires extensive programmer effort. Perhaps surprisingly, we find that this is not the case, for two reasons. First, when OFRs are unserializable, SOFRITEST raises a **precise exception** that exactly identifies the code and data involved. A user study (Section 6.2.1) shows that these exceptions are more useful for writing correct code than the reports from a data race detector. Second, SOFRITEST provides **automatic refinement suggestions** to programmers instructing them precisely how to annotate their code. These suggestions are highly accurate: SOFRITEST suggested the right refinement annotation at the right code location 97% of the time in our evaluation (Section 6.2.3). Ultimately, we find that starting
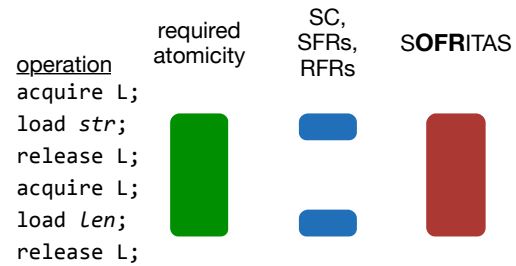

Figure 1. An atomicity violation bug drawn from Firefox [32], where a string's contents and length are read in two separate critical sections, allowing inconsistency to arise. The bug is exposed under the SC, SFR and RFR consistency models. SOFRITAS offers stronger atomicity that automatically prevents the bug from manifesting.

from a safe, overly-atomic foundation and refining to regain progress is an easier path towards correct parallel software than today's approach of building up atomicity from scratch.

The second system we describe and evaluate is SOPRO, the first pure-software, strong memory consistency enforcement mechanism for C and C++ programs. OFR atomicity allows SOPRO to automatically prevent real concurrency bugs from causing a failure, where weaker models permit the failure (Section 6.2.2).

SOPRO uses a fine-grained memory ownership mechanism that ensures a thread has permission to read or write a location before each access (*i.e.*, per-location reader/writer locks). SOPRO *monotonically* acquires a region's reader/writer locks and releases them only when a region completes (implementing strong, strict 2-phase locking [3]). Lock ownership checks in the common case comprise just seven x86 CPU instructions that make cache-friendly accesses to thread-local data. Threads only access their own thread-local data to check ownership, and other threads will only perform reads on that data in the case of a contended lock. Ownership checks are designed such that sequential accesses to data will access sequential lock metadata in memory, providing spatial locality in the cache. SOPRO also leverages existing virtual memory support to quickly release locks in bulk at region boundaries. Coupled with a novel high-locality memory layout for locks, SOPRO provides the benefits of OFR serializability with scalability and performance: SOPRO's average slowdown of 1.59x compares favorably with the 1.99x slowdown of the state-of-the-art Java-based Valor system [5], though of course the implementations and benchmarks differ significantly. Moreover, Valor benefits from a lazy conflict detection optimization that is permissible because deleterious side-effects from racy code are sandboxed by the JVM. In addition to delaying exception delivery, which can frustrate debugging, lazy conflict detection in our C/C++ context would require additional runtime instrumentation to provide sandboxing [14], mitigating the benefits. Ultimately, SOPRO provides coarser atomicity (OFRs), more precise exception delivery, and better performance than Valor.

SoPro is useful on its own for programs that run well with OFR atomicity, and it can transparently hide the failures due to some concurrency bugs, even high-level atomicity violations. SofriTest and SoPro are also useful together where suggestions from SofriTest help general programs to run exception-free under SoPro.

This paper makes the following contributions:

- We describe the SOFRITAS memory consistency model, which provides ordering-free region (OFR) serializability guarantees that are stronger than previous models
- We show that SofriTest detects real bugs, including a multi-critical-section atomicity violation in memcached, and that its annotation suggestions are useful for adapting programs to run with OFR atomicity
- We demonstrate that a pure-software implementation of SoPro automatically prevents 5 of the 7 concurrency bug failures we find with SofriTest, while achieving acceptable (1.59x) performance overhead.
- To the best of our knowledge, SoPro is the first pure-software strong memory consistency model for unmanaged C/C++ code.

This paper is organized as follows. Section 2 provides background on strong memory consistency models. Section 3 explains the SOFRITAS algorithm and API. Section 4 describes SOFRITAS's guarantees formally. Section 5 describes the software implementation and optimizations shared by both SofriTest and SoPro. Section 6 presents an evaluation of SofriTest's usability and bug detection capabilities, and of SoPro's performance. Finally, we discuss related work in Section 7.

2 Background: Strong Consistency Models

There have been several proposals of strong memory consistency models that help catch bugs, simplify reasoning for programmers, and simplify language specifications. These proposals can be characterized along two dimensions: the *granularity* of the code regions for which serializability is guaranteed, and the precision with which serializability violations are detected. SOFRITAS improves upon prior work along both dimensions. We address these dimensions in turn, and then discuss empirical measurements of atomicity with our model and those of previous work.

2.1 Why is OFR atomicity needed?

Figure 1 shows a distilled version of an atomicity violation bug from Firefox [32]. Two separate critical regions read the string str and length len, potentially observing them while they are inconsistent (i.e., during an update). The required atomicity for this code is indicated by the green marker on the left, but the provided locking is insufficient to enforce this atomicity.

The blue markers in the middle indicate the span of atomic regions under three consistency models: sequential consistency (SC) [28, 37, 45], synchronization-free regions (SFRs) [33, 40] and release-free regions (RFRs) [5], which all have the same region boundaries for this program. With SC, each individual instruction is atomic. SFRs break atomicity at lock acquires and releases, and at ordering constructs. RFRs break atomicity at lock releases and release ordering constructs (barrier waits, condition notifies and fork). None of these models, however, enforce atomicity for long enough regions to prevent reading inconsistent data in Figure 1's example.

SOFRITAS, which breaks atomicity only at ordering constructs, provides atomicity across the critical sections that access the string's fields because there is no ordering construct between them. SOFRITAS's coarse-grained OFR atomicity, as shown by the red marker on the right, prevents the bug from manifesting by ensuring that str and len are read atomically. In Section 6.2.2, we examine a real concurrency bug in memcached [49] which is detected and prevented by OFR atomicity but not by weaker atomicity models.

2.1.1 Quantifying Atomicity
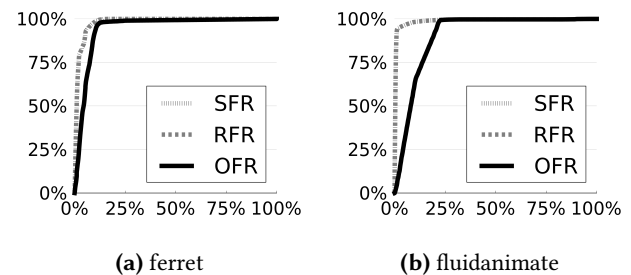

(a) ferret   **(b)** fluidanimate

Figure 2. CDFs showing the percentage of memory locations (x-axis) that are atomic over a percentage of regions (y-axis). In fluidanimate, most SFRs and RFRs protect <1% of memory locations, whereas OFRs protect roughly 20% of memory locations.

While OFRs have intuitive benefits over finer-grained atomic regions, it is not obvious that these advantages provide any benefit given the structure of real code. With frequent ordering synchronization, OFRs, SFRs and RFRs may be similarly-sized in practice. We analyzed how much atomicity various consistency models provide in real code. For each region r, we record r's *width* – how many distinct memory locations were accessed within r. At the end of the execution, we compute how many regions have a width of w as a fraction of all regions, and plot this as a cumulative distribution function. Our width metric captures the ability of a consistency model to enforce atomicity across memory locations, reducing the probability of multi-variable atomicity violations like the bug in Figure 1.

Figure 2 shows atomicity measurements for some representative programs. In these CDFs, a program with a curve
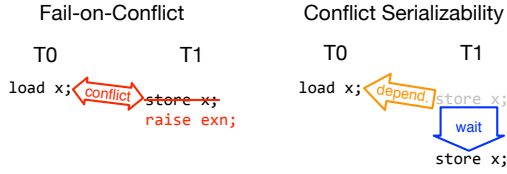
Figure 3. A simple program that raises an exception with the "fail-on-conflict" approach, but is exception-free under conflict serializability.

Figure 4. A program that can throw an exception with reader-writer locking but not with mutex locking. Arrows indicate dependencies between threads; the dashed arrow can arise only with reader-writer locking.

that rises more gently has *more* atomicity because it has a large proportion of wide regions and a small proportion of low-width (narrow) regions. Curves that rise steeply indicate that most regions are narrow. Figure 2a shows results for ferret, where shared queues lead to periodic ordering synchronization that punctuates atomic regions under all models. SFRs and RFRs are always narrow, while OFRs are much wider in general and the widest OFRs are much wider than the widest SFRs/RFRs. Figure 2b shows results for fluidanimate, which has complex fine-grained locking that makes regions narrow for SFRs and RFRs, while OFRs are considerably wider. Overall, we find that 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. We also find no significant difference in atomicity between SFRs and RFRs, suggesting that the benefits of moving from SFRs to RFRs are limited.

2.2 Conflict Serializability

Work on strong memory consistency models, like SOFRITAS, provides a guarantee that completed executions are equivalent to ones in which all regions execute serializably. When an execution is not serializable, an exception is thrown. Some previous work [5, 33, 45] has used a highly conservative "fail-on-conflict" policy to detect unserializable executions. These systems raise an exception whenever a memory conflict occurs between concurrently-executing regions, where a conflict is defined as a pair of memory operations to the same location, from different threads, with at least one operation being a write.

The "fail-on-conflict" strategy offers an asymmetric guarantee: an exception-free execution is serializable at region granularity, while any exception raised is due to a data race. However, some executions with exceptions are *also* serializable. To reduce the number of exceptions, the FastRCD-A system [58] adopts the more precise notion of *conflict serializability*, which SOFRITAS also implements via strong, strict two-phase locking [3].

Figure 3 illustrates the distinction between the "fail-on-conflict" approach and conflict serializability via a simple program. Conflict serializability waits when it encounters a conflict, which allows it to execute this program serializably
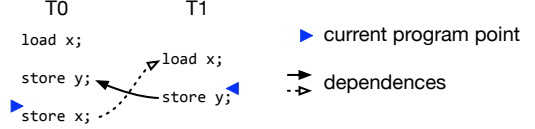
every time. [58] demonstrates that conflict serializability reduces region conflicts for some programs by orders of magnitude compared to the "fail-on-conflict" approach. Conflict serializability is thus especially necessary in conjunction with OFRs, as large regions increase the probability of region conflicts.

3 OFR Atomicity with SOFRITAS

In this section we describe SOFRITAS's algorithm for enforcing OFR conflict serializability, how SOFRITAS provides precise exceptions, and how exceptions can be resolved via user annotations. These elements are used by both SOFRITEST and SOPRO.

3.1 Core Algorithm

Each memory location x is associated with a reader-writer 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. t acquires l_x in read-mode for a read and in write-mode for a write. At the end of a region, when t encounters an ordering construct – a fork, join, condition signal, condition 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 and at least one of t's or u's accesses is a write (i.e., t's and u's accesses conflict). t's inability to acquire l_x indicates a memory conflict between t and u. Some existing consistency models raise an exception on t's access to x due to the memory conflict, but SOFRITAS 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.

SOFRITAS's use of reader-writer locks (instead of mutex locks) increases parallelism by allowing read-sharing of data, which is crucial for good performance and scalability. However, neither reader-writer locking nor mutex locking result in strictly fewer exceptions on all programs. Reader-writer locks introduce a notion of a *lock upgrade*, where a thread t holds a lock l_x in read-mode and then tries to write to x, requiring that l_x be upgraded to write-mode. This upgrade requires waiting for all existing readers of x to release their locks. Figure 4 shows a sample program that can raise an exception under reader-writer locking, but not with mutex locking. Thread t_0 is blocked due to a lock upgrade (dashed

arrow), while thread t_1 waits on t_0 for its write to y. With mutex locks, t_0's lock upgrade cannot arise. Instead, the threads get serialized at their initial reads of x, and once they acquire l_x they can run to completion.

3.2 SOFRITAS API

SOFRITAS has a small API consisting of four annotations that allow programmers to refine a program's region specification and optimize performance. These annotations are used for both SofriTest and SoPro.

A SofritasRelease() annotation refines a program's region specification, sub-dividing a region into smaller regions, *e.g.*, to eliminate an exception. The basic SofritasRelease() annotation explicitly releases a specified location's lock and we include "syntactic sugar" API calls that batch release locks on objects and arrays.

A SofritasRequireMutex() annotation associates a *mutex* lock with a memory location, rather than a reader-writer lock to avoid *upgrade cycles* (Figure 4). The SOFRITAS compiler automatically places SofritasRequireMutex() annotations in most cases (Section 5.2) and the runtime system automatically suggests the placement of a mutex annotation if an upgrade cycle occurs.

A SofritasEndOFR() annotation ends a region before execution reaches an ordering operation. SofritasEndOFR() is useful when only short regions of code need to be atomic, but ordering operations only rarely end regions. For example, SofritasEndOFR() is helpful in a pipeline parallel application that requires atomicity of pairs of dequeue and enqueue operations only, but rarely executes ordering operations, enforcing much coarser atomicity. Note that a single SofritasEndOFR() annotation may eliminate the need for several SofritasRelease() annotations.

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

3.3 Precise OFR Exceptions

SOFRITAS associates a lock with each memory location. The mapping from memory locations to locks is determined by the granularity at which an application accesses memory. For many applications, SOFRITAS can associate one lock with each 4-byte word of memory. However, in some cases, applications share data at byte granularity. In those cases, SOFRITAS must associate a lock with each byte of memory in order to avoid false positives. SOFRITAS's locking granularity is configurable by the programmer but remains

fixed for an execution. To reduce the costs of lock overhead, SOFRITAS assumes that programs always access a given memory location with loads/stores of a consistent width, *i.e.*, a 4-byte integer is never accessed with single-byte loads and stores. This permits locking only the first byte in a multi-byte access. A future version of SOFRITAS could be extended to lock every byte within each access, using modern processors' 128-bit CAS instructions, or hardware transactional memory support like Intel's TSX instructions, to reduce the overheads of these additional lock acquires.

To reduce the space costs of byte-granularity locking, SOFRITAS adopts a cache-friendly lock representation to maximize locality within each thread (Section 5). SOFRITAS's fine-grained locking allows it to detect precisely when a thread's next operation threatens conflict serializability, *before* the violation has occurred. When an access to memory location x by thread t_0 conflicts with an earlier access to x by another thread t_1, t_0 enters a waiting state until t_1 releases its lock on x. If the threads execute multiple conflicting accesses that forces them to wait for one another, SOFRITAS raises an exception. Through the exception, the programmer can examine an uncorrupted view of t_0's memory in which OFR atomicity has not been violated, and can see the specific operations in t_0 involved in the conflict cycle.

The SOFRITAS runtime uses a distributed deadlock detection algorithm [9] to detect conflict cycles. Only waiting threads run cycle detection, putting the work of deadlock detection off of the execution's critical path.

3.4 Resolving Exceptions with Annotations

SOFRITAS triggers an OFRException when executing OFRs have *at least two* conflicts and the conflicts form a cycle in the conflict graph [3]. An OFRException indicates that the program permitted an unserializable execution of its regions and shows where and how the program must be modified to avoid this exception in the future. Returning to the code from Figure 4, SOFRITAS will suggest to either 1) release the lock on y in t_0 with a SofritasRelease() annotation, 2) release the lock on x in t_1, 3) release both locks, or 4) ensure that x and y are updated together by changing x to use mutex locking or altering the order of stores in t_0. By default, SOFRITAS suggests option 3) on an exception. While SOFRITAS trusts the programmer to choose correctly based on application semantics, SOFRITAS automatically suggests the right annotation 97% of the time (Section 6.2.3).

3.5 Working with Library Code

Using a library with an application running on SOFRITAS involves a few extra steps for the library writer. Library writers should identify library objects, so that SOFRITAS can associate a reader-writer lock with each one. Library API calls should be annotated as logical reads or writes of a library object, *e.g.*, 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. We have found that 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 of our benchmarks use these. Crucially, SOFRITAS still provides coarse-grained atomicity for accesses to library objects: the SOFRITAS 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, atomic operations, etc. This internal synchronization lives outside SOFRITAS. In future work, we plan to extend SOFRITAS's library support to provide synchronization at finer granularity than entire objects, and to explore how a library's internal synchronization can be simplified in the presence of SOFRITAS.

4 Proof of Correctness

This section shows formally that SOFRITAS enforces conflict serializability. We use the following numbered properties of SOFRITAS to support our argument.

1. SOFRITAS associates a single reader-writer lock with each program location.
2. No access to a location by a thread proceeds without first holding the location's lock in the correct mode for the access (i.e., read vs. write mode).
3. No lock is ever released by a thread until all of the thread's region's accesses complete.
4. Conflicting lock acquires (i.e., at least one acquire is a write-mode acquire) are globally ordered.
5. SOFRITAS performs precise cyclic lock waiting (i.e., deadlock) detection.

Theorem 1 (Exception-free serializability). *If a SOFRITAS execution of OFRs is free of OFRExceptions, then the execution is conflict serializable.*

Proof. We prove this claim by assuming that an OFRException-free execution was *not* conflict serializable and showing that this assumption leads to a contradiction. For simplicity we prove the two-OFR case, but our argument generalizes to arbitrary-length conflict cycles.

If an execution is not conflict serializable, then the definition of conflict serializability 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 (1) and (2), SOFRITAS ensures o_i and o_j acquire the lock for x in the correct mode before the access. By the definition of a conflict, one (or both) of o_i or o_j is writing x and by (2) the writer(s) must hold the lock in write mode before the write. By (4), the regions' lock acquires are ordered because the corresponding accesses conflict on x. By (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 our 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 our 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 (5), SOFRITAS precisely detects this deadlock and reports an exception *violating our assumption that the execution was exception-free.* Thus, our assumption leads to a contradiction, proving that an exception-free execution is conflict serializable. □

Theorem 2 (Unserializability of OFRException). *If a SOFRITAS execution triggers an OFRException, then the execution is not conflict serializable.*

Proof. As above, we prove the two-OFR case, but our argument generalizes to arbitrary conflicts. We assume SOFRITAS has generated an OFRException and show that it implies a violation of conflict serializability. By (5), an OFRException corresponds to SOFRITAS 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 (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 (1) and (4). Consequently, the regions' impending accesses form two conflicts, one on x and one on y. Furthermore, by (1) and (5), because the regions cyclically wait to acquire locks, the corresponding access conflicts are also cyclic.

For brevity, we omit our case-based analysis of all the possible combinations of cyclically conflicting regions. Instead, we claim that all OFRException-triggering combinations of o_i and o_j that have such a cyclic conflict form a cycle in the conflict graph. By the definition of conflict serializability [3], a cycle in the conflict graph indicates a violation of conflict serializability. □

Discussion. We note that SOFRITAS's correctness proof is *bijective* – an execution free of exceptions is conflict serializable and an exception indicates that an execution is not conflict serializable. By contrast, prior work [5, 33, 45] provided a weaker correctness argument of asymmetric form – an exception-free execution is serializable, but an exception corresponds only to a *conflict*, indicating a data-race, but not necessarily a violation of conflict serializability.
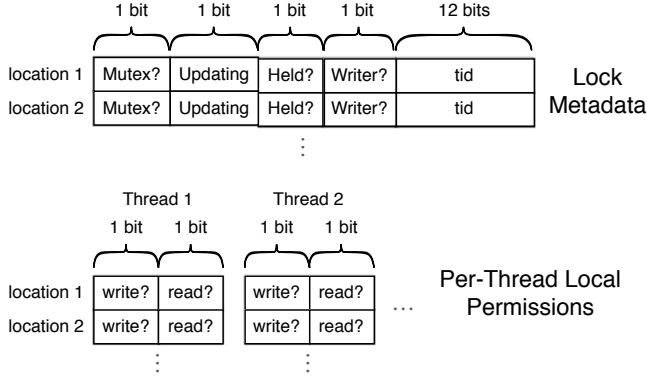
Figure 5. SOFRITAS locks consist of global metadata and per-thread local permissions. Global and per-thread metadata are stored in separate places in memory. Only per-thread metadata is accessed on lock ownership checks, which are frequent compared to lock acquires.

Figure 6. SOFRITAS lock state transition diagram. Transitions shown assume that no other thread currently has permissions that prevent the acquire.

5 Implementing OFR Atomicity

SOFRITAS requires efficient support for checking and acquiring locks before each load and store instruction, which we implemented in a compiler and a runtime library. The following sections motivate and describe SOFRITAS's lock implementation, which is shared by both SofriTest and SoPro.

5.1 Lock Implementation

SOFRITAS's locks are designed to support efficient lock ownership checks, as these checks vastly outnumber lock acquires on most programs (see Column 2 of Table 1). Figure 5 shows the structure of the locks used by SOFRITAS to enforce OFR atomicity. Each lock is split into disjoint structures: 16 bits of global metadata and 2 bits (per-thread) of thread-local permissions. 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. 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 SofritasRequireMutex() 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.
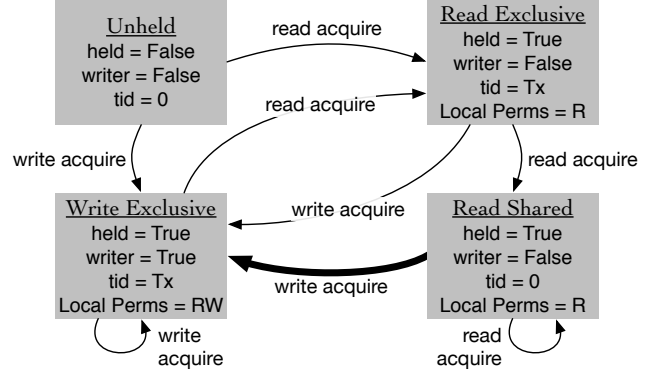
This admits an efficient implementation of bulk releases via the madvise system call, using the MADV_DONTNEED flag to zero a thread's entire local permissions space via page remapping. We found that madvise is noticeably faster than using memset/bzero to zero memory directly, as it avoids repeatedly zeroing memory on pages that are never used by a thread.

Since only local permissions are updated on a release, global metadata can become stale in that it may reflect state before or after the most recent release operation. The definitive state of a lock is recorded in local permissions, and global metadata serves as a conservative summary of local permissions. 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 thread-local permissions must be consulted is for a read-shared to write-exclusive transition (heavy arrow in Figure 6), where the writer waits for all readers to release their locks.

SOFRITAS uses a modified version of the *tcmalloc* allocator. Calls to sbrk and mmap for large memory allocations are redirected to the SOFRITAS runtime so that the global metadata and thread-local shadow spaces are placed directly after the heap. Lock lookups are thus a simple offset from a given heap address.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| | | | | Batch | pthreads | | SofriTest | | | | |
App	Checks	Acquires (Read, Write)	Releases	Releases	Ord	Atom	Mutex	Release	OFR	Easy	Hard
blackscholes	7.13 B	2.8% (65.0%, 35.0%)	-	16	2	-	-	-	-	-	-
bodytrack	95.67 B	3.4% (38.9%, 61.1%)	370 M	180 K	17	34	3	20	-	20	-
canneal	21.61 B	23.5% (58.3%, 41.7%)	2.1 B	48 K	3	13	1	7	1	7	-
dedup	3.11 B	28.8% (98.8%, 1.21%)	7.8 M	91 K	9	13	5	18	1	18	-
ferret	187.90 B	5.9% (88.1%, 11.9%)	70 K	33 K	8	7	2	7	1	4	3
fluidanimate	228.67 B	20.2% (8.6%, 91.4%)	228 B	40 K	16	10	5	20	-	20	-
streamcluster	428.34 B	51.5% (99.2%, 0.8%)	4.4 B	638 K	30	6	-	11	-	11	-
swaptions	196.01 B	0.1% (2.8%, 97.2%)	-	16	2	-	-	-	-	-	-
gups	500.03 M	80.0% (50.0%, 50.0%)	100 M	16	2	2	-	1	-	1	-
pagerank	1.23 B	25.3% (79.4%, 20.6%)	247 M	17	2	10	-	7	-	7	-
histogram	3.75 B	0.1% (54.7%, 45.3%)	-	16	2	-	-	-	-	-	-
kmeans	14.78 B	1.1% (99.6%, 0.4%)	-	3 K	2	-	-	-	-	-	-
linear_regression	4.87 K	50.0% (53.3%, 46.7%)	-	16	2	-	-	-	-	-	-
matrix_multiply	2.01 B	0.1% (0.1%, 99.9%)	-	16	2	-	-	-	-	-	-
pca	16.10 B	0.4% (52.9%, 47.1%)	8 M	32	2	4	-	3	-	3	-
reverse_index	2.14 B	49.3% (99.5%, 0.5%)	157 K	30	2	4	1	2	-	2	-
string_match	1.42 B	0.1% (5.3%, 94.7%)	-	16	2	-	-	-	-	-	-
word_count	740.51 M	0.5% (66.8%, 33.2%)	-	156	2	-	-	-	-	-	-
pbzip2	121.9 K	69.2% (40.1%, 59.9%)	60.2 K	245	34	103	7	10	-	10	-

Table 1. Frequency of SofriTest operations and annotations. Acquires (Column 2) are listed as a percentage of checks (Column 1), and subdivided into the fraction of read, and write, acquires as a percentage of all acquires.

5.2 Compiler Support

Immediately before each load or store instruction, the SOFRI-TAS compiler inserts calls to perform a read or write acquire, respectively. The performance critical lock ownership check is inserted as inlined assembly because 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. The SOFRITAS compiler inserts a simple function call that is marked as always_inline to ensure that subsequent passes inline the function. The inserted function contains inline assembly for the lock ownership check and a non-inlined function call to the acquire function.

The SOFRITAS compiler elides instrumentation for locations that do not escape the stack. The escape analysis is borrowed from the Section Based Program Analysis [15]. 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 [2]. 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).

Many of the programs that we studied required atomic updates on counters. A counter update is most straightforwardly instrumented as both a load and a store. This naive instrumentation is likely to lead to an upgrade dependency cycle between multiple threads that successfully acquire a read lock on the counter load and then attempt to acquire a write lock for the store. To prevent this common scenario, any load that is post-dominated by a store is instrumented as a store instead. This optimization often reduces the need for SofritasRequireMutex() annotations.

The SOFRITAS compiler also inserts hooks to handle program events including program start, thread create, thread join, barrier waits, and condition variable operations.

6 Evaluation

6.1 Experimental Setup

We evaluated SofriTest and SoPro by running and annotating selected benchmarks from PARSEC [4], Phoenix [43], approximate computing benchmarks [1], 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.

6.2 SofriTest Usability

In this section, we evaluate how well SofriTest can find and fix concurrency bugs. We conduct this evaluation along three dimensions: a user study assessing the debugging utility of SOFRITAS, describing the concurrency bugs that SofriTest finds in our workloads, and characterizing the accuracy and utility of the atomicity refinement annotations that SofriTest suggests.

6.2.1 User Study

We conducted a user study comparing the utility of an OFR atomicity system like SOFRITAS with that of a conventional data race detector, for debugging a simple parallel program [16]. The study asked 45 graduate students in computer science to add missing synchronization to a short program. The program had a 3D vector class with a setter for each individual coordinate (x,y,z) and a normalize method that updated all three components of the vector. Participants were asked to ensure that the setter methods could execute in parallel with each other and were expected to infer that fine-grained locking was necessary. Participants were asked to synchronize the code twice - once with the help of output from a data race detector and once with the help of OFRException reports. The order in which they were asked to synchronize the code was randomized to account for learning effects.

The survey partitioned participants into three groups: (i) thos who correctly synchronized both variants, (ii) those who correctly synchronized one variant but not the other, and (iii) thos who incorrectly synchronized both variants. Of participants that incorrectly added synchronization with one tool's output, but did so correctly with the other tool's, participants given the OFR tool's output were statistically significantly more likely to correctly add the synchronization. The result suggests that using an OFR tool like SOFRITAS for adding synchronization is easier than using outputs from a data race detector, which are analogous to the exceptions generated by previous memory consistency models [19, 33] – though some consistency models [5, 58] raise delayed exceptions only at region boundaries which are even less useful.

The survey asked students to rate their own knowledge of parallel programming and also to define mutexes, data races, and deadlocks to assess their prior knowledge. On average, students rated their own parallelism expertise at 3.18 out of 7 and scored 3.84 points out of 7 total points in defining parallel programming terms. We found no correlation between the student's parallelism expertise and their ability to correctly synchronize either variant of the test code.

6.2.2 Detecting and Preventing Concurrency Bugs

SofriTest identified 6 concurrency bugs in PARSEC benchmarks, and one in memcached. Specifically, we found concurrency bugs in the pthreads versions of bodytrack (2 bugs), ferret (1 bug), fluidanimate (1 bug), and streamcluster (2 bugs). We verified each of these bugs manually. The bugs in ferret and streamcluster have been reported by prior work [33]. To our knowledge, the bugs in bodytrack and fluidanimate have not been previously identified. SoPro prevents 5 of the 6 bugs automatically, requiring no annotations to do so. For the final bug in fluidanimate, SofriTest raised an OFRException and precisely identified the necessary annotation to fix the bug with no need for manual reasoning. We give three illustrative examples below of SofriTest's ability to detect concurrency bugs.

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 well-defined in C++ [25] for single-threaded code, with parallelism this behavior constitutes an atomicity violation on vptr. SoPro automatically prevents this failure by ensuring the main thread holds a lock on vptr until the main thread completes its work and joins with the workers.

fluidanimate In fluidanimate, an atomicity violation arises due to a faulty manual optimization. The border array tracks shared matrix entries, and the code locks only those shared entries. On the native input, border is computed incorrectly, causing some shared entries to be accessed without synchronization. SofriTest automatically acquires a lock on the cnumPars array that serializes accesses to indices of the array that are shared by multiple threads. SofriTest suggests a SofritasRelease() annotation on the accessed index of the cnumPars array to prevent SOFRITAS from throwing an OFRException when threads (non-concurrently) access the same index of the array. If the programmer attempts to perform the same faulty optimization with SOFRITAS annotations, SofriTest detects the concurrency bug and pinpoints the array accesses that violate the required atomicity of the application.

memcached To evaluate SofriTest's performance on a larger code base, we examined a known concurrency bug found in memcached [49, 57]. 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 SofriTest on the memcached-127 bug. With no additional annotations, SofriTest detects the concurrency bug via an OFRException and pinpoints the cache item update as the correct location for an annotation.

6.2.3   Annotation Characterization

Atomicity refinement annotations are used to enable application progress and prevent SOFRITEST from detecting previously-found concurrency bugs. Columns 5-11 of Table 1 compare pthreads synchronization calls with the SOFRITEST annotations needed to allow our programs to run exception-free. Column 5 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. Column 6 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 1 report the number of annotations used to refine the coarse atomicity provided by OFRs. Column 7 shows the number of SofritasRequireMutex() annotations required. In all cases, SOFRITEST correctly suggested that a SofritasRequireMutex() annotation is required by examining the lock state when an OFRException occurs. If a lock has multiple shared readers and at least one thread is attempting to acquire write privileges, a SofritasRequireMutex() annotation is almost certainly required. The SOFRITAS compiler analysis (Section 5.2) avoids the need for 13 additional mutex annotations.

Column 8 reports the number of SofritasRelease() annotations required for each application. In most cases, the number of SofritasRelease() 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 6.2.2.

Column 9 reports the number of SofritasEndOFR() or SofritasContinueOFR() 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 SofritasEndOFR() annotation in each benchmark to represent this. canneal represents a different case in which SofritasRelease() annotations handle all of the necessary releases for the benchmark, making the batch lock release operations at each barrier wait superfluous. To improve the performance of canneal, we add a single SofritasContinueOFR() annotation to the barrier

wait to prevent the batch lock release. This optimization yields a 4x speedup.

The final two columns of Table 1 report the ease of adding annotations using SOFRITEST. When an OFRException occurs, SOFRITEST suggests the location and type of annotation that it thinks is required to refine atomicity and avoid the exception. Column 10 reports the number of annotations that we found to be easy to place using the suggestions provided by SOFRITEST. These annotations were either located at the exact line suggested by SOFRITEST or close to the suggested line. In the six close cases, SOFRITEST 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. A simple post-dominator analysis could suggest the location of these close cases automatically. Column 11 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, SOFRITEST correctly suggests a release annotation on the *tail* pointer. Once this annotation has been added, one of the two suggestions provided by SOFRITEST on the next OFRException 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. Despite the fact that not all of the suggestions provided by SOFRITEST 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 OFRException. Further, one of the two suggestions is correct, leaving the programmer with a multiple-choice question of how to resolve the OFRException.

Beyond a comparable annotation burden, SOFRITEST provides fail-stop exceptions and precisely suggests fixes for missing annotations. In contrast, missing locks in pthreads and other models [5, 58] are not fail-stop and are not accompanied by code suggestions.

6.3   SoPro Evaluation

We evaluate the runtime performance, scalability, and memory overheads of SOPRO as compared to pthreads execution. We report average performance over 10 runs. We use a 4-byte-per-lock mapping for all programs except bodytrack, dedup, ferret, reverse_index, and pbzip2 which share byte-sized data and therefore required a 1-byte-per-lock mapping.

6.3.1   Runtime Overheads

Figure 7 presents the runtime slowdown of SOPRO over pthreads. For each thread count, we normalize to the pthreads execution for the same thread count. Across all thread counts,
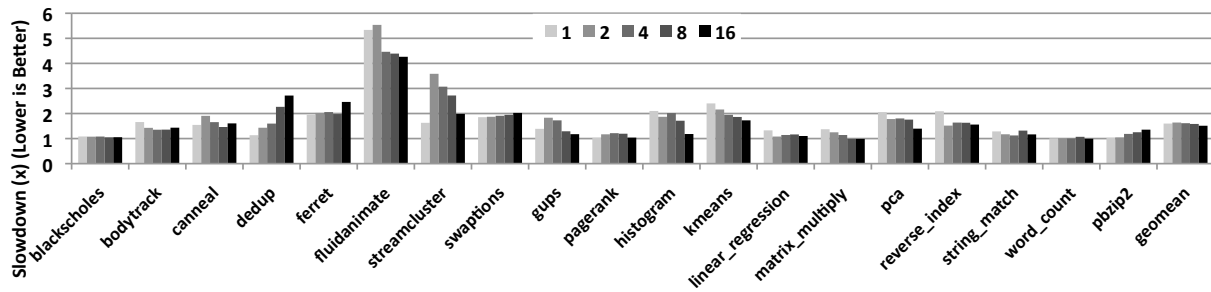
Figure 7. Runtime overheads for SoPro with 1, 2, 4, 8 and 16 threads (light to dark bars) normalized to the pthreads baseline for the same thread count.

SoPro's average runtime slowdown is only 1.59x, substantially lower than even the 1.99x overhead of Valor [5], the most closely related prior work. Moreover, SoPro provides potentially more useful consistency exceptions immediately when a conflict occurs, rather than lazily reporting exceptions at a region's end like Valor. To our knowledge, SoPro is the lowest-overhead, coarse-grained memory consistency enforcement mechanism implemented for C and C++ programs. With its low average overhead, including many benchmarks with overheads below 2x, SoPro is a viable candidate for providing strong atomicity guarantees even in deployed systems.

Some benchmarks had larger slowdowns that can be attributed to frequent ordering operations (*e.g.*, barriers) – Column 4 of Table 1 shows that fluidanimate and streamcluster all perform many batch releases at the end of OFRs. Although SoPro has highly-optimized batch releases, clearing the thread-local shadow spaces too frequently can be detrimental to performance because locks must be reacquired after each batch release. As listed in column 2 of Table 1, linear regression requires few lock checks because the majority of memory accesses in the benchmark target a read-only memory mapped file. SoPro can safely elide checks to this read-only memory mapped file.

Figure 8 compares the scalability of SoPro with pthreads. We show the scalability of each application using both pthreads and SoPro. Each pthreads bar is normalized to the single-threaded execution using pthreads, and each SoPro bar is normalized to the single-threaded execution using SoPro. SoPro scales similarly to pthreads, as can be seen in the matching bar clusters.

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

6.3.2 Memory Usage

Figure 9 reports the memory overhead for SoPro 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, SoPro 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.

SoPro 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 SoPro metadata to begin with, and the fixed costs of SoPro's other internal data structures magnify the memory overhead. A similar 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 SoPro 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.

6.3.3 Optimizations

As discussed in prior sections, SoPro uses multiple low-level optimizations to reduce performance overheads. To efficiently release locks at OFR boundaries, SoPro calls madvise instead of using memset. Figure 10 shows the overhead of using memset instead of madvise as normalized to the So-Pro baseline. On average, using memset incurs an overhead of 3.88x over the baseline SoPro system.

SoPro also relies on efficient lock checks, which are much more common than lock acquires. SoPro inlines lock checks for increased efficiency because frequent function calls can be expensive, especially when they involve saving and restoring registers on the stack. Figure 11 details the overheads incurred by SoPro when no lock checks are inlined. On average, SoPro incurs a 1.73x overhead over the baseline system when no lock checks are inlined.
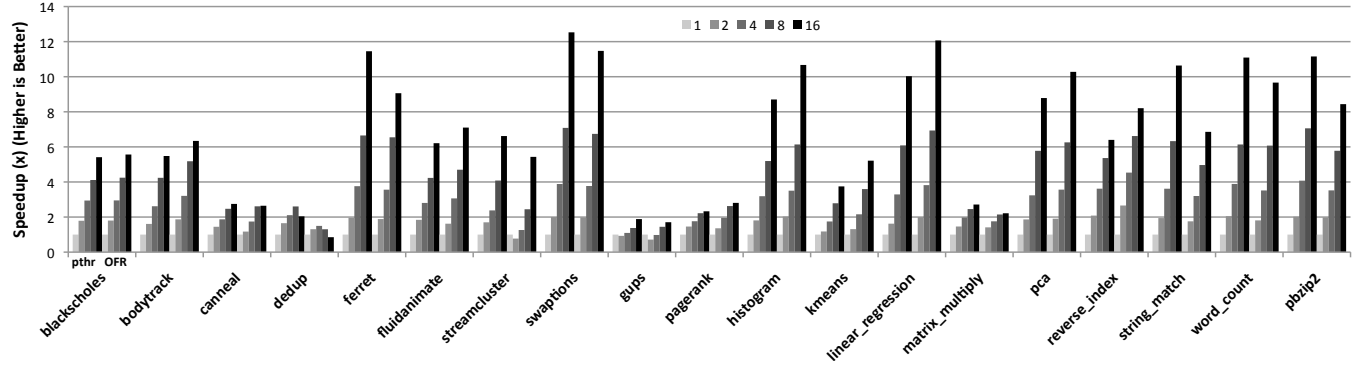
Figure 8. Scalability of SoPro as compared to the pthreads baseline. Each set of bars is normalized to single-threaded execution in the given model.
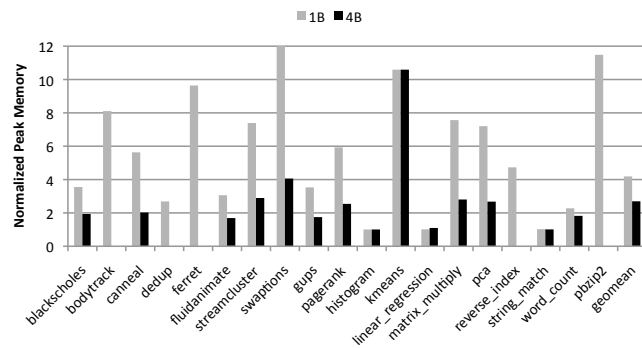

Figure 9. Memory overheads for SoPro compared to pthreads at 16 threads. 1B maps each byte to a lock and 4B maps 4 bytes to a lock.
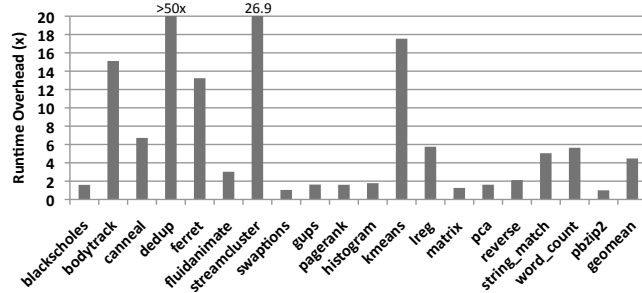

Figure 11. Overheads from not inlining lock checks

Figure 10. Overheads for using memset instead of madvise

7 Related Work

SOFRITAS is motivated by several areas of prior work on multithreaded programmability. Section 2 provides an in-depth comparison of SOFRITAS with other strong memory consistency models, and we describe SOFRITAS's relationship with other relevant work here.

Several schemes have been proposed for detecting **sequential-consistency violations** with custom hardware support [18, 39, 42]. These schemes detect a cycle of data races, which indicates that SC has been compromised, and
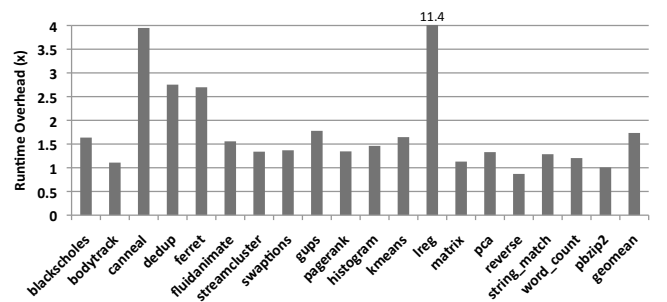
use speculation to rollback execution to before the SC violation occurred. SOFRITAS's dependence cycle detection works similarly, but is a pure-software approach and enforces serializability at OFR (instead of single-instruction) granularity.

Data-centric synchronization schemes explicitly associate locks with data and then assure that this locking discipline is automatically enforced. In some systems [10, 50, 51] a programmer specifies the variable-to-lock association. This association can also be inferred [27] 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. The queue implementation in dedup requires that a lock acquired in a callee is held across a function return and released by the caller. Releasing the lock at the return introduces an atomicity violation bug. In contrast, SOFRITAS's OFR atomicity guarantees do not rely on any specific code structure and provide the required atomicity for dedup.

SOFRITAS shares a similar goal with techniques for **detecting atomicity violations** [12, 13, 21, 22, 29–31, 34, 41, 54], which use heuristics to decide where atomic regions should start and end, striking a balance between missing real atomicity violations and reporting spurious ones. In contrast,

SOFRITAS is an execution model that provides strong atomicity guarantees that can prevent many atomicity failures, even on the very first execution.

Transactional memory (TM) systems leverage programmer-specified atomic blocks [24, 47] that can be implemented via optimistic or pessimistic concurrency [17, 38]. Like conventional locking, programming with TM involves incrementally strengthening a program's atomicity instead of SOFRITAS's top-down atomicity refinement approach. TM thus remains vulnerable to the atomicity violations and data races that plague lock-based programming because a TM system trusts the programmer to place transactions correctly. Nevertheless, TM is a potentially valuable implementation technique for future versions of SOFRITAS. In particular, TM-inspired rollback techniques could allow automatic recovery from SOFRITAS exceptions, reducing the burden on SOFRITAS programmers still further.

The TCC [23] and Automatic Mutual Exclusion (AME) [26] systems place all code inside coarse-grained transactions. Instead of providing a stronger execution model for existing code, TCC and AME target new programming models: parallelization of sequential code and task parallelism, respectively. Both schemes employ less precise notions of serializability than SOFRITAS, and incur additional complexity due to the use of optimistic concurrency which complicates I/O and other irrevocable operations. Furthermore, neither scheme provides automated guidance on atomicity refinement as SOFRITAS does. Transaction boundaries also break atomicity for all variables, unlike SOFRITAS which can relax atomicity on individual variables at a time to minimize the risk of atomicity violations.

Cooperative concurrency [55, 56] 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 as SOFRITAS does.

Program synthesis of parallel programs [8, 48, 52] often works by refining overly-coarse atomicity under programmer guidance, similar to the SOFRITAS approach. SOFRITAS's dynamic techniques scale to much larger programs, however, than synthesis currently supports.

8   Conclusion

We introduced the SOFRITAS system, which provides an OFR serializability memory model that is stronger and more precise than previous work. The SofriTest system detects new and known concurrency bugs in PARSEC and memcached. We show that the SoPro runtime system requires just a 1.59x average runtime overhead and scales similarly to pthreads up to 16 threads. SofriTest and SoPro require a similar number of annotations compared to pthreads synchronization, but SofriTest provides automatic, targeted assistance in refining OFR atomicity when necessary.

References

[1] V. Balaji, B. Lucia, and R. Marculescu. Overcoming the data-flow limit on parallelism with structural approximation. In *WAX 2016*, 2016.

[2] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. 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, pages 53–64, New York, NY, USA, 2010. ACM.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[4] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[5] S. Biswas, M. Zhang, M. D. Bond, and B. Lucia. 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, pages 241–259, New York, NY, USA, 2015. ACM.

[6] C. Blundell, M. M. K. Martin, and T. Wenisch. Invisifence: Performance-transparent memory ordering in conventional multiprocessors. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, June 2009.

[7] H.-J. Boehm and S. V. Adve. Foundations of the c++ concurrency memory model. In *Proceedings of the SIGPLAN 2008 Conference on Programming Language Design and Implementation*, PLDI '08, 2008.

[8] M. Botinčan, M. Dodds, and S. Jagannathan. Proof-directed parallelization synthesis by separation logic. *ACM Trans. Program. Lang. Syst.*, 35(2):8:1–8:60, July 2013.

[9] G. Bracha and S. Toueg. Distributed deadlock detection. *Distributed Computing*, 2(3):127–138, 1987.

[10] L. Ceze, P. Montesinos, C. von Praun, and J. Torrellas. Colorama: Architectural support for data-centric synchronization. In *Proceedings of the 13th Symposium on High-Performance Computer Architecture*, pages 133–144, Feb. 2007.

[11] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk Enforcement of Sequential Consistency. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007.

[12] L. Ceze, C. von Praun, C. Caşcaval, P. Montesinos, and J. Torrellas. Concurrency control with data coloring. In *Proceedings of the 2008 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, pages 6–10, 2008.

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

[14] L. Dalessandro and M. L. Scott. Sandboxing transactional memory. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 171–180, New York, NY, USA, 2012. ACM.

[15] M. Das, G. Southern, and J. 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 2015, pages 283–284, New York, NY, USA, 2015. ACM.

[16] C. DeLozier, Y. Peng, A. Eizenberg, B. Lucia, and J. Devietti. Orca: Ordering-free regions for consistency and atomicity. Technical Report MS-CIS-16-01, University of Pennsylvania, May 2016.

[17] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, pages 194–208, Berlin, Heidelberg, 2006. Springer-Verlag.

[18] Y. Duan, D. Koufaty, and J. Torrellas. Scsafe: Logging sequential consistency violations continuously and precisely. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 249–260, March 2016.

[19] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A race and transaction-aware java runtime. In *Proceedings of the SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 245–255, June 2007.

[20] 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/.

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

[22] C. Flanagan, S. 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.

[23] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with transactional coherence and consistency (tcc). In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1–13, Oct. 2004.

[24] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.

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

[26] M. Isard and A. Birrell. Automatic mutual exclusion. HotOS '07, pages 3:1–3:6, 2007.

[27] S. Kempf, R. Veldema, and M. Philippsen. Compiler-guided identification of critical sections in parallel code. In *Proceedings of the 22nd International Conference on Compiler Construction*, pages 204–223, 2013.

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

[29] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. Muvi: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pages 103–116, Oct. 2007.

[30] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, page 37âĂŞ48, Oct. 2006.

[31] B. Lucia and L. 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.

[32] B. Lucia, L. Ceze, and K. 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.

[33] B. Lucia, L. Ceze, K. Strauss, S. Qadeer, and H.-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.

[34] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-aid: Detecting and surviving atomicity violations. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, pages 277–288, June 2008.

[35] S. Mador-Haim, L. Maranget, S. Sarkar, K. Memarian, J. Alglave, S. Owens, R. Alur, M. M. K. Martin, P. Sewell, and D. Williams. An axiomatic memory model for power multiprocessors. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 495–512, Berlin, Heidelberg, 2012. Springer-Verlag.

[36] J. Manson, W. Pugh, and S. V. Adve. The java memory model. In *Proceedings of The 32nd ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 378–391, Jan. 2005.

[37] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. Drfx: A simple and efficient memory model for concurrent programming languages. In *Proceedings of the SIGPLAN 2010 Conference on Programming Language Design and Implementation*, pages 351–362, June 2010.

[38] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: Synchronization inference for atomic sections. In *Proceedings of The 32nd ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 346–358, Jan. 2006.

[39] A. Muzahid, S. Qi, and J. Torrellas. Vulcan: Hardware support for detecting sequential consistency violations dynamically. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 363–375, Washington, DC, USA, 2012. IEEE Computer Society.

[40] J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. . . . and region serializability for all. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Parallelism*, Berkeley, CA, 2013. USENIX.

[41] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. SIGSOFT '08/FSE-16, pages 135–145, 2008.

[42] X. Qian, J. Torrellas, B. Sahelices, and D. Qian. 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, pages 535–548, New York, NY, USA, 2013. ACM.

[43] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[44] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding power multiprocessors. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 175–186, New York, NY, USA, 2011. ACM.

[45] A. Sengupta, S. Biswas, M. Zhang, M. D. Bond, and M. Kulkarni. Hybrid static–dynamic analysis for statically bounded region serializability. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 561–575, Mar. 2015.

[46] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. X86-tso: A rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, July 2010.

[47] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213, Aug. 1995.

[48] A. Solar-Lezama, C. G. Jones, and R. Bodik. Sketching Concurrent Data Structures. In *Proceedings of the SIGPLAN 2008 Conference on Programming Language Design and Implementation*, PLDI '08, pages 136–148, 2008.

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

[50] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *Proceedings of The 32nd ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 334–345, Jan. 2006.

[51] M. Vaziri, F. Tip, J. Dolby, C. Hammer, and J. Vitek. A type system for data-centric synchronization. In *Proceedings of the 24th European conference on Object-oriented programming*, pages 304–328, 2010.

[52] M. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. In *Proceedings of The 37th ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 327–338, 2010.

[53] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Mechanisms for store-wait-free multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007.

[54] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *Proceedings of the SIGPLAN 2005 Conference on Programming Language Design and Implementation*, PLDI '05, pages 1–14, 2005.

[55] J. Yi, T. Disney, S. N. Freund, and C. 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.

[56] J. Yi, C. Sadowski, and C. 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.

[57] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, 2009.

[58] M. Zhang, S. Biswas, and M. D. Bond. Avoiding consistency exceptions under strong memory models. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2017, pages 115–127, New York, NY, USA, 2017. ACM.