4-28-2016

ORCA: Ordering-free Regions for Consistency and Atomicity

Christian DeLozier

Yuanfeng Peng

Ariel Eizenberg

Brandon Lucia

Joe Devietti
University of Pennsylvania

MS-CIS-16-01

ORCA: Ordering-free Regions for Consistency and Atomicity

Abstract

Writing correct synchronization is one of the main difficulties of multithreaded programming. Incorrect synchronization causes many subtle concurrency errors such as data races and atomicity violations. Previous work has proposed stronger memory consistency models to rule out certain classes of concurrency bugs. However, these approaches are limited by a program's original (and possibly incorrect) synchronization. In this work, we provide stronger guarantees than previous memory consistency models by punctuating atomicity only at ordering constructs like barriers, but not at lock operations. We describe the Ordering-free Regions for Consistency and Atomicity (ORCA) system which enforces atomicity at the granularity of ordering-free regions (OFRs). While many atomicity violations occur at finer granularity, in an empirical study of many large multithreaded workloads we find no examples of code that requires atomicity coarser than OFRs. Thus, we believe OFRs are a conservative approximation of the atomicity requirements of many programs. ORCA assists programmers by throwing an exception when OFR atomicity is threatened, and, in exception-free executions, guaranteeing that all OFRs execute atomically. In our evaluation, we show that ORCA automatically prevents real concurrency bugs. A user-study of ORCA demonstrates that synchronizing a program with ORCA is easier than using a data race detector. We evaluate modest hardware support that allows ORCA to run with just 18% slowdown on average over pthreads, with very similar scalability.

Disciplines

Computer Engineering | Computer Sciences

Comments

MS-CIS-16-01

ORCA: Ordering-free Regions for Consistency and Atomicity

Christian DeLozier[1], Yuanfeng Peng[1], Ariel Eizenberg[1], Brandon Lucia[2], and Joseph Devietti[1]

[1]University of Pennsylvania
[2]Carnegie Mellon University

Abstract

Writing correct synchronization is one of the main difficulties of multithreaded programming. Incorrect synchronization causes many subtle concurrency errors such as data races and atomicity violations. Previous work has proposed stronger memory consistency models to rule out certain classes of concurrency bugs. However, these approaches are limited by a program's original (and possibly incorrect) synchronization.

In this work, we provide stronger guarantees than previous memory consistency models by punctuating atomicity only at ordering constructs like barriers, but not *at lock operations. We describe the Ordering-free Regions for Consistency and Atomicity (ORCA) system which enforces atomicity at the granularity of* ordering-free regions *(OFRs). While many atomicity violations occur at finer granularity, in an empirical study of many large multithreaded workloads we find* no *examples of code that requires atomicity coarser than OFRs. Thus, we believe OFRs are a conservative approximation of the atomicity requirements of many programs. ORCA assists programmers by throwing an exception when OFR atomicity is threatened, and, in exception-free executions, guaranteeing that all OFRs execute atomically.*

In our evaluation, we show that ORCA automatically prevents real concurrency bugs. A user-study of ORCA demonstrates that synchronizing a program with ORCA is easier than using a data race detector. We evaluate modest hardware support that allows ORCA to run with just 18% slowdown on average over pthreads, with very similar scalability.

1. Introduction

Despite decades of research progress, writing correct and efficient multithreaded programs remains an open challenge. Driven by Moore's Law and demand for energy efficiency, multicore hardware is ubiquitous, from servers to embedded devices [18]. Parallelism is the norm and we must make multithreaded programming accessible to all programmers.

One promising way to simplify parallel programming is through stronger memory consistency models. Real-world consistency models like those for Java [37], C++ [6], and hardware architectures [47, 45, 36] are notoriously complicated. There have been many proposals [10, 54, 4, 38] to enforce stronger consistency models, from sequential consistency (SC) [28], in which individual instructions execute atomically, to stronger models [46, 33, 3, 17] in which atomicity is provided for groups of contiguous instructions. However, all of these existing schemes assume the program's existing synchronization is correct. Such an unsafe assumption leaves the door open for high-level concurrency bugs like atomicity violations.

Figure 1 shows a real-world atomicity violation bug, drawn from Mozilla Firefox. The str and length fields should be updated atomically, but were mistakenly placed in separate critical sections. This code is sequentially-consistent and data-race-free, showing that these safety properties are insufficient to provide the atomicity the program requires. Prior schemes that support atomicity of coarse-grained regions (*e.g.*, small regions not under programmer control [38, 46], synchronization-free regions (SFRs) [33], release-free regions (RFRs) [3]) also fail to guarantee correctness here because the faulty locking code actually *violates* the operations' atomicity.

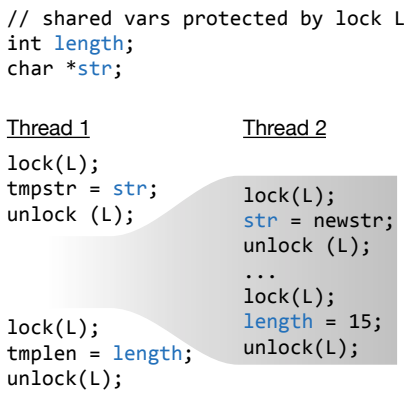

Figure 1: An atomicity violation bug from Firefox. Thread 1 reads an inconsistent state for the string. Figure reproduced from [32].

To deal with such subtly incorrect code, we propose the ORCA system, which enforces atomicity at the granularity of *ordering-free regions* of code – dynamic regions that are free of inter-thread ordering constructs like barriers or condition variables. ORCA is the first system to provide such strong atomicity guarantees, and as a direct consequence is the first system to treat a program's locking constructs as *untrusted*, since atomicity is not broken at lock acquires or releases as with previous schemes [38, 46, 33, 3]. ORCA's coarse-grained atomicity guarantees prevent misapplied synchronization from violating atomicity, even for difficult, high-level bugs like Figure 1. In an empirical study of over 780K lines of code in 15 real-world and benchmark programs (Section 2.3), we

find *no* examples of code that requires atomicity coarser than OFRs. Thus, we believe OFRs are a safe approximation of the atomicity requirements of many programs.

The ORCA consistency model provides a host of benefits. First, ORCA's coarse-grained OFR atomicity virtually eliminates atomicity violations, which we demonstrate empirically in our evaluation with real programs. Moreover, ORCA's strong atomicity guarantees subsume weaker properties like SC and data race freedom, so programmers using ORCA never encounter sequential consistency violations or data races. ORCA's regions span between ordering constructs, so they enforce a greater scope of atomicity than prior schemes (SFRs, RFRs). If a program's OFRs cannot execute atomically, ORCA raises a precise exception to alert the programmer. ORCA adopts a notion of atomicity based on *two-phase locking* [1] that is more general than that adopted by previous consistency models [33, 3, 17]. This allows ORCA to enforce stronger atomicity while simultaneously raising fewer exceptions than previous work.

When faced with an exception for an OFR conflict, the programmer may determine that full OFR atomicity is unnecessary, and can divide the region into two regions using an annotation to prevent future exceptions. Unlike the error-prone manual atomicity annotations in conventional parallel programming models, the ORCA runtime provides guidance on the necessity of annotations and correctly suggests their exact placement in over 80% of cases. Because ORCA treats locking constructs as untrusted, ORCA provides strong atomicity guarantees even for programs with *no locking at all* or with *incorrect or missing locks*, greatly simplifying multithreaded application development. We illustrate this benefit with a user study and show that ORCA automatically prevents real bugs.

ORCA is implemented via fine-grained, per-location locking. The ORCA hybrid hardware-software runtime system automatically ensures that a location's lock is acquired before each access to that location. Locks are released at ordering constructs. To make ORCA efficient, we evaluate lightweight hardware support to accelerate common operations like lock-ownership checks. We also show that we can coarsen ORCA's atomic regions while simultaneously improving its performance, at the cost of a small number of false exceptions.

This paper makes the following contributions:

- We describe the ORCA memory consistency model, which provides stronger atomicity guarantees, and a more general notion of serializability, than previous models
- We show that ORCA is able to prevent real atomicity violations caused by concurrency bugs, including new bugs missed by previous work
- We demonstrate that a hybrid hardware-software implementation of ORCA enables low (18%) performance overhead and very similar scalability to pthreads.
- We show that programs running with ORCA rarely trigger exceptions, and that most exceptions can be resolved with annotations automatically suggested by the ORCA runtime

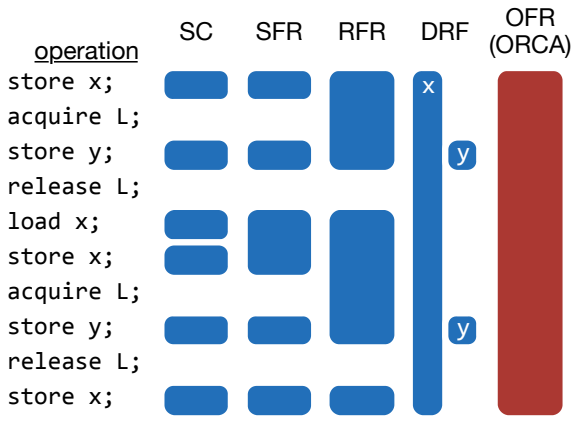

Figure 2: The span of atomicity under various memory consistency models. From left to right: sequential consistency [28], synchronization-free regions [33], release-free regions [3], data race freedom [16], and ordering-free regions (ORCA). OFRs offer superior atomicity by not treating the program's locking as trusted.

- We perform a user-study showing that it is easier to synchronize a program using ORCA than using a traditional data race detector.

This paper is organized as follows. Section 2 provides background on strong memory consistency models like ORCA. Section 3 explains how ORCA enforces OFR atomicity. Section 4 details ORCA's hardware support. Section 5 describes the ORCA hardware simulator. Section 6 and 7 show programmability and performance results, and Section 8 discusses related work.

2. Strong Memory Consistency Models

Though sequential consistency (SC) is often touted as a strong memory consistency model – and it is indeed stronger than many hardware and language models – SC guarantees atomicity only at the level of individual machine instructions. This guarantee is too weak to meet most programs' needs, so programmers must use locking or transactional memory to provide additional atomicity. To help catch bugs and simplify program reasoning, several stronger consistency models have been proposed. We characterize these proposals along two dimensions: the *granularity* of the code regions for which atomicity is guaranteed, and the notion of *serializability* used to determine whether atomicity has been violated. We address each of these dimensions in turn, and then discuss empirical results that identify how much atomicity real programs need.

2.1. Region Granularity

Figure 2 illustrates the relative strength of these models on a simple program, showing where atomic regions start and end in each model. In SC, regions contain a single instruction. With synchronization-free regions [33] and release-free regions [3], atomicity is broken only at synchronization op-

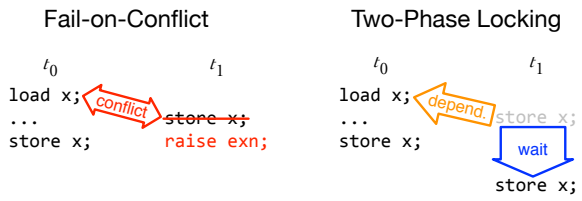

Figure 3: A simple program that raises an exception with the "fail-on-conflict" approach of previous work, but that can execute atomically under ORCA's two-phase locking approach.

erations or lock release operations, respectively. Data race freedom [17, 16] provides stronger atomicity that can extend across release operations, and is best thought of in terms of *per-variable atomic regions*. DRF provides atomicity for all accesses to x because any remote access to x within this program will be flagged as a data race. In contrast, DRF provides atomicity for only the individual accesses to y because another thread could access y while holding `L` without triggering a race (as with the atomicity violation bug in Figure 1).

By enforcing atomicity at the granularity of ordering-free regions (OFRs), ORCA provides atomicity across the critical sections modifying y, preventing potential atomicity violations on y and yielding stronger atomicity than previous models.

2.2. Region Serializability

The guarantees provided by strong memory consistency models can be thought of as ensuring that regions of dynamic instructions (SFRs, RFRs, OFRs, etc.) execute atomically and serializably, *i.e.*, an execution's behavior is equivalent to the behavior of some sequential execution of its regions. Previous memory consistency models [33, 3, 17] have taken a very conservative "fail-on-conflict" approach to serializability. They raise an exception whenever a memory conflict occurs between concurrent regions, where a conflict is a pair of memory operations to the same location, from different threads, where at least one operation is a write. This approach is sound, in that an execution free of exceptions is serializable. However, some executions with exceptions are also serializable. To reduce such warnings, ORCA adopts a more precise notion of serializability called *two-phase locking* (2PL) [1].

Figure 3 illustrates the distinction between the "fail-on-conflict" approach and 2PL via a simple program containing a single synchronization-free region – thus, SFRs, RFRs, DRF and OFRs all have the same extents for this program. With previous consistency models, the conflict between t_0's load and t_1's store will raise an exception, either precisely at t_1's store [33, 17] or delayed until the next release operation [3]. However, this program can in fact execute in a serializable fashion if t_1's store waits for t_0's region to finish. Both threads will execute atomically, with t_0 serialized before t_1. In fact, this program can always be executed serializably via waiting: whichever thread issues the first operation is serialized first and runs to completion while the other thread waits.

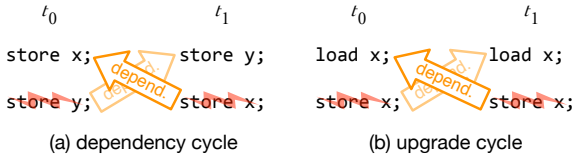

Figure 4: If a cycle of dependencies (orange arrows) arises, ORCA raises an exception (red lightning bolts) in each thread in the cycle.

ORCA enforces 2PL serializability of OFRs by tracking the dependences between OFRs, as shown by the orange arrow from t_1 to t_0 in Figure 3. ORCA raises an exception only when a dependency cycle arises, which indicates that 2PL serializability is about to be broken. While there are more refined notions of serializability than 2PL, they are expensive to maintain and do not offer much additional flexibility [1].

2.3. How much atomicity is necessary?

While it seems intuitive that OFRs are a conservative approximation of the atomicity that programs require, it is theoretically possible that a program could require atomicity that spans an ordering construct. To discover if such code exists, we empirically measured the frequency of critical sections that contain ordering synchronization with a Pin tool [35]. We used our tool to analyze over 780K lines of parallel code, examining all the inputs to all of the PARSEC benchmarks, in addition to the `apache` and `memcached` servers. We found that critical sections in these workloads *never* contained ordering synchronization, illustrating that ordering-free regions are indeed a conservative approximation of many applications' atomicity requirements.

3. OFR Atomicity with ORCA

In this section we describe the core algorithm that ORCA uses to enforce atomicity of OFRs, and several important optimizations for performance and programmability.

The base ORCA algorithm works as follows. On each memory access to a location x by a thread t, t acquires a mutex lock l_x that is associated with x if t does not already hold l_x. We assume for now that programs write memory but do not read it; we generalize to handling loads with reader-writer locks in Section 3.2. We also assume that a thread releases all locks that it holds whenever it encounters an ordering construct; we discuss the implications of releasing locks in Section 3.5. If t is not able to acquire l_x, then some other thread u has accessed x in its current OFR. t's inability to acquire l_x indicates a memory conflict between t and u. Existing consistency models raise an exception on t's access to x because of the memory conflict, but ORCA instead tracks a dependence from t to u and waits until u releases l_x, avoiding many exceptions.

3.1. Precise OFR Exceptions

ORCA's use of per-location locking allows it to detect precisely when a thread's next operation would violate 2PL se-

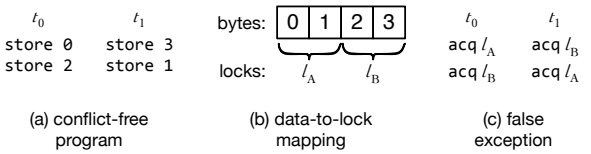

Figure 5: A false exception due to coarse-grained locking.

rializability, before the violation has occurred. ORCA tracks dependences between threads when memory conflicts arise, and if a dependency cycle arises then an OFRException is raised in each thread (Figure 4a). Waiting until a cycle arises avoids the spurious exceptions triggered in previous work [33, 3, 17]. When a thread t_0 becomes dependent on another thread t_1, t_0 enters a waiting state that naturally preserves t_0's program state. If an exception is raised, a programmer can examine an uncorrupted view of t_0's memory in which OFR atomicity has not been violated, and can see the specific operation in t_0 that would break 2PL serializability.

ORCA associates a lock with each byte of memory, as bytes are the minimal addressable unit of memory in most architectures. To combat the resulting space overhead, the ORCA architecture uses a compressed lock representation (Section 4.2) that provides good performance. Associating locks with larger memory regions, *e.g.*, words, would also save space but can unfortunately introduce false exceptions. Figure 5a shows a conflict-free program that could trigger an OFRException if locking were to be performed on pairs of bytes instead of single bytes (Figure 5b). Coarse-grained locking converts t_0 and t_1's independent writes into a cyclic dependence (Figure 5c).

3.2. More Concurrency with Reader-Writer Locks

To support read-sharing without serializing all accesses, ORCA can associate a *reader-writer* lock with each location instead of a *mutex* lock. Using reader-writer locks increases parallelism and can help avoid exceptions as well, *e.g.*, programs with cross-coupled read sharing will trigger an exception with mutex locking but not with reader-writer locks. Reader-writer locks do, however, introduce the possibility of *upgrade cycles* in which a lock is held by multiple readers, each of whom tries to upgrade to write ownership (Figure 4b). The program in Figure 4b does not trigger an exception with mutex locking, showing that neither mutex nor reader-writer locks result in strictly fewer exceptions for all programs. ORCA uses reader-writer locks by default to enable more parallelism. Upgrade cycles can be avoided via the use of ORCA's ORCA_require_mutex() annotation which associates a mutex lock with a memory location rather than a reader-writer lock. The ORCA runtime system automatically suggests the placement of mutex annotations when upgrade cycles occur.

3.3. Resolving Exceptions with Annotations

When an OFRException is raised, it indicates that 2PL serializability of an OFR is about to be broken. This exception may indicate an atomicity violation, or may indicate a situation in which OFR atomicity is unnecessary. ORCA helps the programmer to distinguish these situations by translating exceptions into targeted "multiple-choice questions" for the programmer to identify how to avoid the exception on future program runs. Returning to the code from Figure 4a, the programmer must choose to either 1) release the lock on x in t_0 with an ORCA_release() annotation, 2) release the lock on y in t_1, 3) release both locks, or 4) ensure that x and y are updated together by introducing a new lock or changing the order of accesses to x and y. While ORCA trusts the programmer to choose correctly based on application semantics, ORCA automatically suggests the right annotation 95% of the time (Section 6.2).

3.4. Detecting Dependence Cycles

The ORCA runtime uses a distributed deadlock detection algorithm [8] to detect dependence cycles. Cycle detection is performed only by waiting threads, keeping the cost of detection low. If a cycle is detected with a thread t_0 waiting on an access to location x, where the lock for x is held by another thread t_1, an OFRException is raised at t_0's current PC. The exception contains the PC at which t_1 last read and wrote x. Using this information, ORCA suggests the last-read and last-write PCs as possible source locations for release annotations. In the case of an upgrade cycle (Figure 4b), the runtime reports the variable on which the upgrade cycle occurred and suggests annotating that variable with a ORCA_require_mutex() annotation. ORCA does cycle detection in software at low cost; cycle detection is rare and involves only waiting threads.

3.5. Higher Performance with Lazy Releases

The ORCA execution model we have discussed thus far releases all locks at every ordering construct, which is sufficient to guarantee 2PL serializability of OFRs but comes with a large performance tax. This batch lock release operation requires either maintaining a list of all acquired locks, or adding per-thread version numbers to locks (i.e., vector clocks), which increases the size and complexity of locks substantially.

To combat this overhead we explore an optional *lazy release* policy that 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 atomicity 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. Lazy releases also avoid the overheads of batch lock

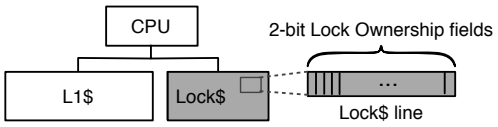

Figure 6: ORCA architecture, with new ORCA hardware shaded.

release operations. Thus, lazy releases provide improved performance *and* stronger atomicity.

A subtlety of lazy releases is that a dependence cycle may not violate 2PL serializability. We find in practice that exceptions almost always correspond to real 2PL serializability violations (Section 6.2) once simple elements of program structure, like pipelines or phased computations separated by barriers, are taken into account. ORCA handles these elements with *staged locking*. A thread is associated with a particular *stage* of a computation. A total ordering on stages allows a thread from a later stages to steal a lock acquired by another thread in an earlier stage, avoiding dependences and maintaining OFR atomicity. We extend ORCA locks with a simple *scalar clock* mechanism to track stages (Figure 7).

Consider a barrier-based program in which threads move from one stage to the next at each barrier. At each lock acquire, ORCA records the current thread's stage with the lock. A thread can steal locks held by a thread in a previous stage, but not from its own stage or future stages. The ORCA runtime identifies barrier-based stages automatically, requiring no programmer input; thread pipelines require a simple annotation per pipeline stage. Staging annotations are untrusted and are verified at runtime: an incorrect staging annotation that contradicts the program's sharing patterns will trigger an OFRException to support straightforward debugging.

4. Architecture Support for ORCA

When a thread accesses a memory location x in ORCA, it must hold the lock for x (acquiring it if necessary). In a conventional software system, these lock operations would impose a high cost. However, they are cheap with targeted hardware support, just as in the case of other rich abstractions like virtual memory, memory safety [13, 42] or data-race freedom [14]. In this section we describe how hardware support to translate from memory locations to locks and a dedicated lock cache accelerate frequent ORCA operations.

4.1. Translating from Data to Lock

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 (Figure 7a) 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 $l_x = (x << 3)|(1 << 63)$. This simple calculation is performed by the ORCA hardware in

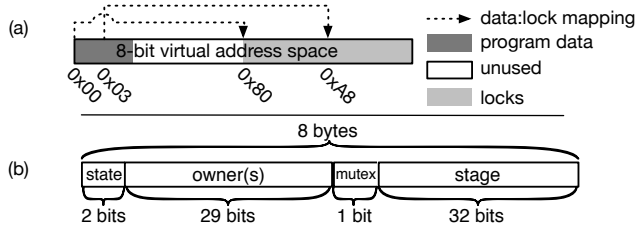

Figure 7: ORCA's (a) data:lock mapping for an example 8-bit virtual address space and (b) reader-writer lock format.

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:lock mapping. Paging does not affect the data:lock mapping and both program data and locks can be paged transparently.

4.2. Lock Cache

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 7b. 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 (Section 3.2) and the `stage` field is used for ORCA's staged locking optimization (Section 3.5).

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, like memory safety [13, 42] and data race detection [14], 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.

The lock cache maps a data address x to the ownership state of the corresponding lock l_x 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 6): an n-byte lock cache line holds information for $4n$ locks which correspond to $4n$ contiguous bytes of program memory.

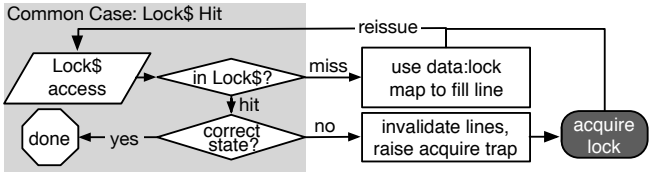

Figure 8: Lock cache operation. Only the *acquire lock* is done in software.

The operation of the lock cache is outlined in Figure 8. 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:lock mapping described in Section 4.1. 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 (Section 3.5), 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.

4.3. ISA Extensions

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

ORCA also adds an ORCA_load_lock instruction that takes an address x and loads the corresponding lock l into the data cache. ORCA_load_lock eliminates the extra instructions needed by software to compute lock addresses. ORCA_load_lock 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 (Figure 7b) 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.

4.4. Support for Flexible Locking

ORCA's simple data:lock translation process is efficient, but it requires a fixed lock format and rigid address space layout. 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 ORCA design that uses a four-level trie (like a page table) to map data to locks.

In this *lock trie* design, the data-to-lock mapping and lock format are under software control. On a lock cache miss, a software handler is invoked to walk the lock trie and gather ownership information. This ownership information can be stored in the lock cache, allowing lock cache hits to be serviced entirely by hardware. Similar to a TLB in a conventional processor, the lock cache accelerates the mapping from data:lock implicitly along with ownership information.

4.5. Memory Consistency Model Considerations

Since every memory access in ORCA is a potential lock acquire, reordering memory accesses is tantamount to reordering lock acquires. In conventional programming models, reordering lock acquires is prohibited because it may introduce data races and deadlocks [37, 6, 5]. In ORCA, however, reordering synchronization cannot introduce data races since all accesses to a given location are protected by the same lock. If a lock acquire and data access are reordered as a unit, reordering preserves this invariant.

To avoid introducing false dependence cycles through memory reordering, it is sufficient to enforce sequentially consistent (SC) [28] execution for ORCA programs, which requires the participation of the compiler and hardware. There are many research proposals for high-performance implementations of SC hardware [21, 10, 54, 4] and compilers [39, 49] that show little performance degradation compared to relaxed consistency models. Such a system could serve as useful platform for ORCA. Moreover, ORCA does not prohibit reordering for all shared memory accesses, but only for accesses that lead to lock acquires.

5. Experimental Setup

The ORCA **architecture simulator** is based on the cache modules of the open-source PIN-based ZSim simulator [44]. Our baseline configuration is a 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 continuing 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 [24] 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 $76\mu m^2$, 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.

5.1. Improving Schedule Coverage

Because the presence of exceptions with ORCA can be schedule-dependent, we take several measures to ensure that we have tested a wide variety of schedules. First, we run each benchmark with the simdev through simlarge inputs; the running time of the native inputs on our simulator is prohibitively long.

Second, we adopt the Lockout deadlock injection tool [27] 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 benchmark. 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. Although it is possible that further OFRExceptions 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 ORCA program can be run with an OFRException handler that logs exceptions and continues execution, similar to a conventional programming model but with the advantage of exception logs for post-mortem debugging.

5.2. System Implementation Issues

ORCA provides additional annotations to handle synchronization for external libraries. ORCA does not currently instrument external libraries. Instead, annotations are used to identify each library call and whether the call is a logical read or a write of the library's data structures, similar to the annotations in [55]. ORCA acquires a corresponding lock on the library object (*e.g.* the base address of an STL `vector`) that is held until an explicit release operation, just like regular ORCA locks, to ensure atomicity across library calls.

6. Usability Evaluation

In this section, we evaluate the usability of OFR atomicity as implemented by ORCA. We performed a survey-based user study that compared the difficulty of fixing data races in a parallel program with the difficulty of fixing OFRExceptions. We also evaluated ORCA's applicability to real programs by running several PARSEC [2] benchmarks (`blackscholes`, `bodytrack`, `canneal`, `dedup`, `ferret`, `fluidanimate`, `streamcluster`, and `swaptions`) with ORCA, as well as the real-world `memcached` application. Our experiences are detailed in two case studies. Finally, we report new bugs discovered in the PARSEC benchmark suite and discuss how ORCA automatically prevents them.

6.1. User Study

We empirically evaluated the difficulty of inserting ORCA 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 program had a 3D vector class with a setter for each coordinate and a normalize method that accessed the coordinates directly. Participants were asked to ensure 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 OFRException reports generated by ORCA, the other by reports from a data race detector which are similar to the exceptions generated by previous memory consistency models [33, 3, 17]. We randomized the order of the variants to account for learning effects.

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. We focused on the second group that got one variant correct, but not the other. We define the probability, p_{orca}, of getting the OFRException variant correct, but the data race variant incorrect. We define the probability, p_{race}, of getting the data race variant correct, but the OFRException 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 OFRException variant and not the data race variant suggests that using ORCA's OFRExceptions 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 ORCA's coarse-grained atomicity.

App	Placement (§6.2) pthreads Ord	Atom	ORCA Mtx	Rel	Exact	Close	Other	Discov. (§6.3) ORCA Real	False
blscholes	0	0	0	0	0	0	0	0	0
bodytrack	15	34	8	24	22	2	0	6	0
canneal	1	13	3	6	6	0	0	3	0
dedup	7	13	4	25	13	2	10	8	0
ferret	6	7	3	6	18	0	4	4	0
fluid	14	10	2	8	6	2	0	4	0
stream	28	6	0	0	0	0	0	0	0
swaptions	0	0	0	0	0	0	0	0	0
memcachd	37	62	26	182	155	27	0	174	5

Table 1: Characterization of annotation placement and discovery. Shaded rows indicate benchmarks for which ORCA has the same or fewer total atomicity annotations than pthreads.

6.2. ORCA Annotation Characterization

We evaluate ORCA's usability by characterizing the annotations required to prevent OFRExceptions in selected PARSEC benchmarks and `memcached`. We compare ORCA's annotation burden to that of adding synchronization in the conventional pthreads parallel programming model. As described in Section 5.1, we ran each benchmark with ORCA until no additional OFRExceptions were found. For each OFRException, we added one or more annotations to the benchmark to prevent the OFRException in future runs.

The left half of Table 1 reports the annotation burden for ORCA and the synchronization burden using pthreads. For most benchmarks (shaded rows), ORCA requires the same or fewer release and mutex annotations than pthreads requires atomicity constructs. Moreover, ORCA annotations are far simpler to add than pthreads critical sections due to the reliable guidance provided by the ORCA runtime system. The *Exact*, *Close*, and *Other* columns of Table 1 detail how many ORCA annotations are correctly placed according to the ORCA runtime system's suggestions. The *Exact* column shows that 82% of ORCA annotations are placed exactly where the runtime system suggests, as described in Section 3.3. An additional 13% of annotations are placed within a few lines of where the runtime system suggests (the *Close* column). Close annotations are often placed outside of an if-statement or loop, rather than inside the block of code to ensure that the annotation covers both conditional cases. Finally, the *Other* column indicates that just 5% of annotations needed to be manually placed by the programmer. These annotations lie in the `queue` code used in `dedup` and `ferret` to implement the parallel pipeline. We illustrate a manual annotation in a case study of `dedup` (Section 6.4). `dedup` and `ferret` also required annotations for staged locking to identify the pipeline structure. These benchmarks required 5 and 6 stage annotations, respectively.

6.3. Resolving OFRExceptions

As described in Section 3, ORCA raises an OFRException when a thread's operation may violate 2PL serializability of OFRs. With the lazy release optimization (Section 3.5), 98% of the OFRExceptions raised by ORCA indicate a real 2PL serializability violation (Table 1). The false exceptions (just 2%) all occurred in `memcached` due to an unstructured condition variable wait between two threads. Without lazy releases, only real exceptions are possible, but the performance benefits of lazy release (Figure 11) make the small programmability burden compelling, especially given ORCA's automatic and accurate annotation suggestion.

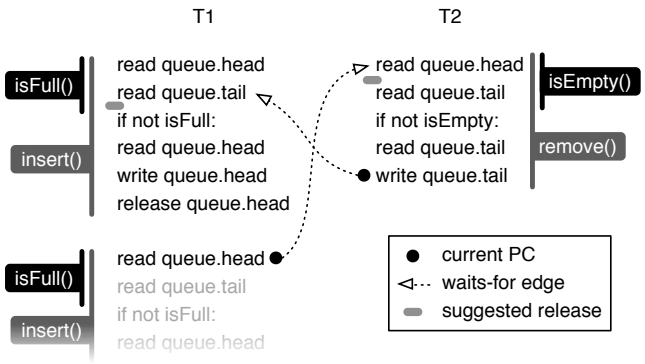

Figure 9: Example of a dependence cycle from `dedup` that required programmer intervention. `queue.head` is protected by a mutex lock while `queue.tail` is protected by a default reader-writer lock.

6.4. Case Study: dedup

As mentioned in the previous section, applying ORCA to the queue implementation in `dedup` presented a few unique challenges. Figure 9 illustrates a dependence cycle from `dedup` where ORCA 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 OFRException for which ORCA does not suggest the correct location.

In the encountered dependence cycle, T1 performs an insert which requires calling `isFull()` to ensure there is space in the queue. At the end of the insert, T1 releases the mutex lock on `queue.head` while holding a read lock on `queue.tail`. Next, T2 performs a remove, first checking that the queue is non-empty. At the end of the remove, T2 gets blocked trying to acquire write permissions on `queue.tail`, which is not protected by a mutex lock (yet). Next, when T1 attempts to perform another insert, T1 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`, ORCA 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

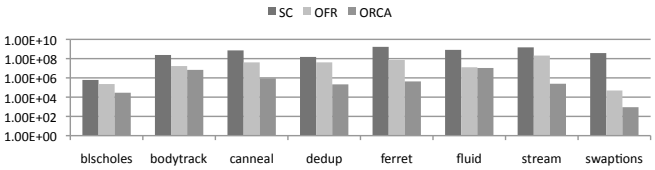

Figure 10: Number of potential atomicity violations for `pthreads` and ORCA. Note the log y-axis.

this subtle case, ORCA 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.

6.5. Case Study: memcached

We ported `memcached` to ORCA to see how a real-world application works with ORCA. While ORCA requires 182 release annotations to be added to the code, ORCA gave correct suggestions for 155 annotations; the other 27 releases were inserted close to where ORCA suggested. Besides these release annotations, 26 mutex annotations were added, all of which could be done mechanically with the suggestions given by ORCA. 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 ORCA can scale to non-trivial real-world applications like `memcached`, and that the ORCA can provide accurate annotation guidance in large code bases.

6.6. Potential for Atomicity Violations

Consistency models guarantee and enforce atomicity at different granularities, as discussed in Section 2. Sequential Consistency (*SC*) only guarantees atomicity for each individual instruction that accesses memory. Ordering-free region atomicity (*OFR*) guarantee much coarser atomicity between ordering constructs. ORCA lazily enforces OFR atomicity, preserving atomicity across ordering boundaries in many cases. Figure 10 shows how often atomicity is broken under each of these consistency models. For *SC*, atomicity is broken on every shared memory operation for the location that is accessed. For *OFR*, atomicity is broken at each ordering construct for every location that the thread has accessed since the last ordering construct. For *ORCA*, atomicity is broken when a location's lock is released, whether at an ordering construct or via an annotation. For all benchmarks, ORCA provides stronger atomicity than both SC and OFR. Overall, ORCA's lazy enforcement of OFRs yields stronger atomicity than both SC and eager OFRs.

6.7. Bugs in PARSEC

By analyzing ORCA's annotations and serialization of OFRs, we were able to identify 7 new bugs in the PARSEC benchmark suite. Specifically, we found atomicity violations in the `pthreads` versions of `bodytrack`, `ferret`, `fluidanimate`, and `streamcluster`. We verified each of these bugs by directly instrumenting the code to ensure that an atomicity violation did in fact exist. For 5 of these bugs, ORCA automatically prevents the bugs from manifesting as failures by correctly enforcing OFR atomicity without any programmer involvement. The remaining two bugs (in `fluidanimate` and `ferret`) initially raised an OFRException. ORCA precisely reported the annotations needed to resolve the OFRExceptions with no need for manual reasoning.

We give two illustrative examples of ORCA's ability to automatically prevent concurrency bugs. In `bodytrack`, an object's *this* pointer is passed to another thread before the object's constructor finishes. Writes to the *this* pointer in the constructor are concurrent with the thread's accesses. This bug is automatically prevented by ORCA because the parent thread holds a write lock on the *this* pointer that prevents the child threads from using it until the parent thread joins. In `fluidanimate`, the *border* array tracks shared matrix entries, and the code locks only those shared entries. On the `native` input, *border* is computed incorrectly, leading to a bug. ORCA automatically serializes updates to shared entries just as if *border* were computed correctly.

7. Performance Evaluation

We evaluated ORCA's performance using the hardware simulator described in Section 5. First, we demonstrate that the hardware support described in Section 4 improves ORCA's performance. Both hardware address translation and the lock cache offer significant performance benefits. We show that even with hardware support, eagerly releasing all locks at the end of each OFR is expensive. We profile and discuss the runtime overheads incurred by ORCA. Finally, we show that ORCA provides significant parallel speedups over a serial baseline, scaling nearly as well as pthreads. Overall, ORCA simplifies parallel programming while still allowing performance benefits from parallel execution.

7.1. ORCA's Performance with Hardware Support

Our main result is that our proposed hardware support enables efficient execution for ORCA. We simulated four different hardware and software configurations. Figure 11 plots the performance of each of these configurations on a simulated 4-core machine, normalized to a simulated execution of pthreads.

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

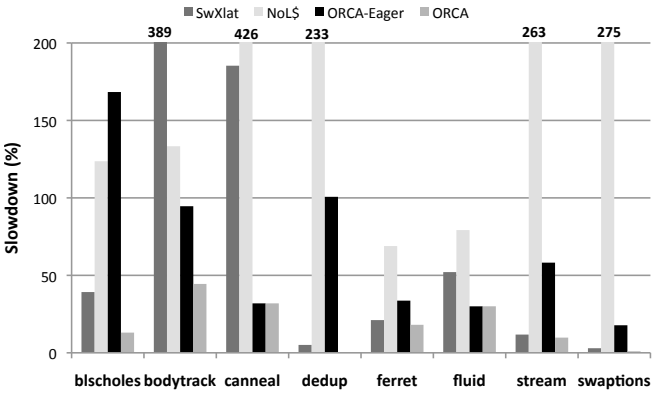

Figure 11: Slowdown of ORCA configs as compared to simulation without ORCA turned on (lower is better). SwXlat shows the performance of the flexible locking scheme. NoL$ shows the performance without the lock cache but with hardware translation from memory address to lock address. ORCA uses both the lock cache and hardware address translation.

other benchmarks, as shown in (Figure 12). 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.

As described in Section 3.5, 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. As discussed in Section 6.3, 98% of the OFRExceptions raised by ORCA with lazy releases result from real violations of OFR atomicity. Thus, we believe the lazy release optimization presents a worthwhile trade-off.

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,

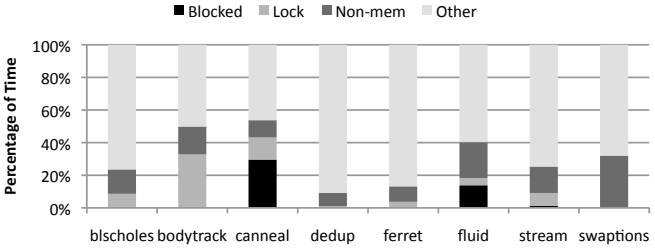

Figure 12: Characterization of where ORCA spends its time: *Blocked* due to serialization, waiting for *Lock* accesses that miss in the lock cache, executing *Non-memory* instructions, or *Other* time spent in the memory hierarchy.

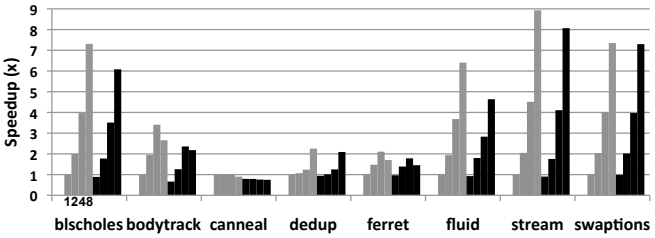

Figure 13: Speedup of benchmarks with 1,2,4, and 8 threads (higher is better). All bars are normalized to pthreads execution with 1 thread (serial). pthreads bars are in grey, ORCA bars are in black.

decreasing pressure on the data cache.

7.1.1. Overhead Breakdown Figure 12 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.

7.1.2. Speedup from Parallel Execution We measured how ORCA's performance scales with additional cores, showing that ORCA is scalable. Figure 13 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, re-

spectively. 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 13 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.

To summarize, ORCA incurs only a small performance overhead compared to expert-synchronized pthreads code, and scales similarly to those expert-synchronized programs. ORCA provides strong atomicity for parallel programs while still enabling the performance benefits of parallelism.

8. Related Work

ORCA is motivated by several areas of prior work on multithreaded programmability. Section 2 compares ORCA with other strong memory consistency models, and we describe ORCA's relationship with other relevant work here.

Like ORCA, **data-centric synchronization** (DCS) schemes also provide atomicity guarantees for parallel code. DCS schemes explicitly associate synchronization objects (*i.e.*, locks) with data. Some work [51, 52, 9] asks a programmer to specify a mapping of each shared variable to a lock. Other work infers the mapping [26] at the risk of missing synchronization. DCS schemes provide atomicity at the granularity of function calls, which is sufficient for many critical sections but not all, *e.g.* the queue implementation in dedup (Section 6.4). dedup's queue requires that a lock acquired in a callee is held across a function return and released in the caller. A DCS system would introduce a silent atomicity violation into the dedup code. In contrast, ORCA's OFR atomicity guarantees do not rely on any specific code structure and provide the required atomicity for dedup.

There is abundant prior work on **detecting concurrency bugs**. ORCA shares a similar goal with techniques for detecting atomicity violations [12, 30, 29, 34, 31, 11, 56, 19, 20, 43], which use heuristics to decide where atomic regions should be, striking a balance between missing and reporting spurious atomicity violations. In contrast, ORCA provides strong atomicity guarantees that are capable of preventing most atomicity violations from manifesting in the first place.

Transactional memory (TM) systems ask the programmer to specify declarative atomic blocks that are then implemented via optimistic concurrency [23, 48] or an automatically-derived locking discipline [15, 40]. TM is still vulnerable to most of the concurrency bugs that plague conventional lock-based programming because a TM system trusts the programmer to demarcate transaction boundaries correctly. For example, weakly-atomic STM systems have complicated semantics in the presence of incorrectly-specified transactions [41]. Strongly-atomic TM systems remain vulnerable to atomicity violations and data races. Nevertheless, TM is a potentially valuable implementation technique for future versions of ORCA. In particular, TM's ability to rollback execution could allow automatic recovery from ORCA exceptions, reducing the burden on ORCA programmers still further.

Cooperability schemes [58, 57] use yield annotations to document where thread interference can occur, leveraging static analysis to ensure that yields account for all possible interference. Cooperability is a sound summary of a program's existing synchronization but does not automatically enforce atomicity guarantees as ORCA does.

The TCC [22] and Automatic Mutual Exclusion (AME) [25] 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 ORCA does. Both schemes employ weaker notions of serializability than ORCA, and incur additional complexity due to the use of always-on optimistic concurrency which complicates I/O and other system calls.

Techniques for **program synthesis** of parallel programs [50, 53, 7] often operate by refining overly-coarse atomicity under the guidance of programmer-specified proofs or invariants. ORCA's dynamic approach can scale to at least medium-size parallel programs like PARSEC benchmarks, beyond the scope that synthesis systems support.

9. Conclusion

We have described the ORCA memory consistency model that enforces atomicity for ordering-free regions of code. ORCA provides low-level safety properties like sequential consistency and data race freedom by construction, and provides superior atomicity and serializability properties to previous consistency models. These features allow ORCA to prevent the manifestation of nearly all atomicity violation bugs, including several new concurrency bugs we discovered through this work. In a user study, we found that synchronizing a program with ORCA is easier than when using a data race detector. ORCA provides these strong safety properties without sacrificing the performance benefits of parallelism. With lightweight hardware support, a hybrid hardware-software ORCA system incurs just 18% performance overhead, and very similar scaling to pthreads.

References

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

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

[3] Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Brandon 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.

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

[5] Hans-J. Boehm. Reordering constraints for pthread-style locks. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2007.

[6] Hans-J. Boehm and Sarita 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.

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

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

[9] 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, February 2007.

[10] 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.

[11] Luis Ceze, Christoph von Praun, Călin Caşcaval, 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.

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

[13] Joe 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*, March 2008.

[14] Joseph Devietti, Benjamin P. Wood, Karin Strauss, Luis Ceze, Dan 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.

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

[16] Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan 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, October 2012.

[17] Tayfun Elmas, Shaz Qadeer, and Serdar 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.

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

[19] 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, January 2004.

[20] Cormac Flanagan, Stephen N. Freund, and Jaeheon 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.

[21] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Two techniques to enhance the performance of memory consistency models. In *Proceedings of the International Conference on Parallel Processing*, pages 355–364, August 1991.

[22] Lance Hammond, Brian D. Carlstrom, Vicky Wong, Ben Hertzberg, Mike Chen, Christos Kozyrakis, and Kunle 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, October 2004.

[23] Maurice Herlihy and J. Eliot 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.

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

[25] Michael Isard and Andrew Birrell. Automatic mutual exclusion. HotOS '07, pages 3:1–3:6.

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

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

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

[29] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan 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, October 2007.

[30] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan 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 37–48, October 2006.

[31] 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, November 2009.

[32] 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.

[33] 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.

[34] 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.

[35] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the SIGPLAN 2005 Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, 2005.

[36] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek 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.

[37] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *Proceedings of The 32nd ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 378–391, January 2005.

[38] Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish 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.

[39] Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. A case for an sc-preserving compiler. In *Proceedings of the SIGPLAN 2011 Conference on Programming Language Design and Implementation*, pages 199–210, June 2011.

[40] Bill McCloskey, Feng Zhou, David Gay, and Eric 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, January 2006.

[41] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Bratin Saha, and Adam Welc. Single Global Lock Semantics in a Weakly Atomic STM. In *Proceedings of the Third ACM SIGPLAN Workshop on Transactional Computing*, February 2008.

[42] 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.

[43] Chang-Seo Park and Koushik Sen. Randomized active atomicity violation detection in concurrent programs. SIGSOFT '08/FSE-16, pages 135–145.

[44] 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.

[45] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek 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.

[46] Aritra Sengupta, Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Milind 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, March 2015.

[47] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. X86-tso: A rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, July 2010.

[48] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213, August 1995.

[49] Abhayendra Singh, Satish Narayanasamy, Daniel Marino, Todd Millstein, and Madanlal Musuvathi. End-to-end sequential consistency. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 524–535, Washington, DC, USA, 2012. IEEE Computer Society.

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

[51] Mandana Vaziri, Frank Tip, and Julian 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, January 2006.

[52] Mandana Vaziri, Frank Tip, Julian Dolby, Christian Hammer, and Jan Vitek. A type system for data-centric synchronization. In *Proceedings of the 24th European conference on Object-oriented programming*, pages 304–328, 2010.

[53] Martin Vechev, Eran Yahav, and Greta 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.

[54] Thomas F. Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. Mechanisms for store-wait-free multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007.

[55] Edwin Westbrook, Raghavan Raman, Jisheng Zhao, Zoran Budimlic, and Vivek Sarkar. Dynamic determinism checking for structured parallelism. In *WoDet*, 2014.

[56] Min Xu, Rastislav Bodík, and Mark 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.

[57] 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.

[58] 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.