Fast RMWs for TSO: Semantics and Implementation

Bharghava Rajaram
University of Edinburgh
r.bharghava@ed.ac.uk

Vijay Nagarajan
University of Edinburgh
vijay.nagarajan@ed.ac.uk

Susmit Sarkar
University of St Andrews
ss265@st-andrews.ac.uk

Marco Elver
University of Edinburgh
marco.elver@ed.ac.uk

Abstract

Read-Modify-Write (RMW) instructions are widely used as the building blocks of a variety of higher level synchronization constructs, including locks, barriers, and lock-free data structures. Unfortunately, they are expensive in architectures such as x86 and SPARC which enforce (variants of) Total-Store-Order (TSO). A key reason is that RMWs in these architectures are ordered like a memory barrier, incurring the cost of a write-buffer drain in the critical path. Such strong ordering semantics are dictated by the requirements of the strict atomicity definition (type-1) that existing TSO RMWs use. Programmers often do not need such strong semantics. Besides, weakening the atomicity definition of TSO RMWs, would also weaken their ordering – thereby leading to more efficient hardware implementations.

In this paper we argue for TSO RMWs to use weaker atomicity definitions – we consider two weaker definitions: type-2 and type-3, with different relaxed ordering differences. We formally specify how such weaker RMWs would be ordered, and show that type-2 RMWs, in particular, can seamlessly replace existing type-1 RMWs in common synchronization idioms – except in situations where a type-1 RMW is used as a memory barrier. Recent work has shown that the new C/C++11 concurrency model can be realized by generating conventional (type-1) RMWs for C/C++11 SC-atomic-writes and/or SC-atomic-reads. We formally prove that this is equally valid using the proposed type-2 RMWs; type-3 RMWs, on the other hand, could be used for SC-atomic-reads (and optionally SC-atomic-writes). We further propose efficient microarchitectural implementations for type-2 (type-3) RMWs – simulation results show that our implementation reduces the cost of an RMW by up to 58.9% (64.3%), which translates into an overall performance improvement of up to 9.0% (9.2%) on a set of parallel programs, including those from the SPLASH-2, PARSEC, and STAMP benchmarks.

Categories and Subject Descriptors C.1.2 [*Processor Architectures*]: Multiple Data Stream Architectures (Multiprocessors)–Parallel processors; D.1.3 [*Concurrent Programming*]: Parallel programming

General Terms Design, Performance, Experimentation

Keywords Read-Modify-Write (RMW), Total-Store-Order (TSO), Atomics

1. Introduction

Read-Modify-Write (RMW) instructions are primitive synchronization operations used to solve a variety of concurrency problems. Herlihy [15] showed that the ability to read and write to an address atomically is critical to solve the *consensus* problem, which abstracts important synchronization problems. Most modern processor architectures have support for such RMW instructions – examples include test-and-set (TAS), fetch-and-add (FAA), compare-and-swap (CAS), and load-linked/store conditional (LL/SC).

In this paper, we concentrate on *Total-Store-Order* (TSO) architectures, variants of which are implemented on mainstream processors like x86 and SPARC, and on the use of RMWs to implement synchronization constructs in TSO. The most pertinent study of such techniques [6, 27] deals with the new C/C++11 concurrency model [7, 10], a model which introduces synchronization reads and writes of various flavors; these reads and writes are referred to as atomics. Batty et al. [6] have shown that this model is correctly implementable on TSO by replacing C/C++11 SC-atomic-writes and/or SC-atomic-reads by RMWs, leaving other language constructs (reads, writes, fences) to be implemented by plain TSO reads, writes and barriers.

Unfortunately, RMWs are costly in current TSO architectures, where they are ordered similarly to a memory barrier [16, 26], incurring the cost of a write-buffer drain in the critical path. When an RMW is issued, the write-buffer is first drained; then the read and the write (of the RMW) are performed atomically – typically by locking the cache-line locally and denying coherence requests to the locked cache-line until the write completes. Thus, instructions following the RMW are allowed to complete only after the write (of the RMW) and the pending writes prior to it complete [24]. As a quick illustration, we measured an average latency of 67 cycles for an RMW on an 8-core *Intel Sandybridge* processor, using the Splash-2 benchmark suite. The latency does not significantly change if we insert a memory barrier (`mfence` instruction) after each RMW, strengthening the hypothesis of a forced write-buffer drain. Since efficient synchronization is important to effectively harness the power of multicores, it is highly desirable that RMWs are efficient. Nevertheless, the optimization of RMWs has historically received little attention [3].

T0	T1	T0	T1	T0	T1	T0	T1	T0	T1
x=1	y=1	W(x)	W(y)	W(x)	W(y)	W(x)	W(y)	RMW(x)	RMW(y)
if(y==0)	if(x==0)	R(y)	R(x)	RMW(z_1)	RMW(z_2)	RMW(y)	RMW(x)	R(y)	R(x)
// critical	// critical			R(y)	R(x)				
(a)		(b)		(c)		(d)		(e)	

Figure 1. Dekker's Algorithm: (a) code snippet. (b) reads and writes involved: $W(x)$ denotes a write to address x, $R(x)$ denotes a read from address x. (c) using RMWs as memory barriers. (d) replacing reads with RMWs. (e) replacing writes with RMWs. In all subfigures, initially, x=y=0.

Semantically speaking, why are TSO RMWs ordered like a memory barrier? We observe that the ordering of RMWs with other memory accesses in TSO depends on the precise semantics of how *atomic* they have to be with respect to those other accesses. TSO can be defined in terms of a global memory order, a relation over memory accesses in the program. Existing TSO RMWs are defined to prevent *writes to any address* from appearing between the read and the write in this global memory order [16, 26]. We call this strict definition *type-1* atomicity. We show that this strict atomicity definition, combined with the other TSO ordering rules, results in type-1 RMWs being strongly ordered with respect to memory operations before and after it, just like a memory barrier.

This strong ordering is exploited by programmers in various synchronization primitives. Fig. 1(a) shows the key steps involved in the implementation of Dekker's algorithm for achieving mutual exclusion and Fig. 1(b) shows the same code in terms of reads and writes. For correctness, at least one of the reads should return a value of 1; otherwise both of the threads can enter the critical section simultaneously. One way to ensure this is by inserting memory barriers between the writes and the reads. In fact, since type-1 RMWs behave like memory barriers, they can be used instead of memory barriers as shown in Fig. 1(c). Alternatively, as shown in Fig. 1d (Fig. 1e), correctness can also be ensured by replacing reads (and/or writes) with RMWs, since type-1 RMWs are strongly ordered with respect to memory operations before and after it in program order. For the same reason, the C/C++11 concurrency model can be implemented on TSO by replacing SC-atomic-reads (and/or SC-atomic-writes) with RMWs [6].

The goal of this paper is to examine whether the ordering of TSO RMWs can be weakened in ways that enable a more efficient implementation, while remaining strong enough for it to replace existing RMWs in synchronization idioms. In other words, can we design fast yet portable RMWs for TSO?

Our approach here is guided by the requirements of general programs, in particular by just what properties are needed for the C/C++11 implementation. Thus, this is hardware design exploiting the freedom provided by language-level concurrency models, and sufficing for those requirements.

Since the ordering semantics of an RMW depends on its atomicity semantics, our approach to weakening the ordering semantics is through weakening the atomicity semantics. In contrast to the strict *type-1* atomicity which disallows writes of *any* address between the read and the write, we consider two weaker atomicity definitions: the *type-2* atomicity which disallows only reads and writes of the same address as the RMW; and the even weaker *type-3* atomicity, which disallows only writes to the same address as the RMW.

Our key contribution is to derive the ordering semantics of the proposed weaker RMWs, and examine if the ordering is strong enough to replace existing RMWs in synchronization idioms (§2.4, §2.5). Unlike a type-1 RMW, a type-2 RMW is not explicitly ordered with respect to memory operations before and after it. Thus, a type-2 RMW cannot be used as a memory barrier like in Fig. 1(c). However, we show that a type-2 RMW appears strongly ordered with respect to any memory operation that synchronizes with the RMW i.e. any memory operation from another thread that is to the same address as the RMW. Indeed, like before, Dekker's algorithm can be ported to TSO by replacing reads (and/or writes) with type-2 RMWs. It is worth noting that in the scenario shown in Fig. 1c (Fig. 1d), each of the RMWs appear to be strongly ordered with respect to the writes (reads) from the other thread which synchronize with the RMW; this strong ordering is again able to guarantee correctness. For similar reasons, C/C++11 can be ported to TSO by replacing SC-atomic-writes (and/or SC-atomic-reads) with type-2 RMWs. Thus, type-2 RMWs are able to replace existing type-1 RMWs in all synchronization idioms, except when used as a memory barrier.

A type-3 RMW is also not explicitly ordered with respect to memory operations before and after it, and hence cannot be used a memory barrier (like a type-2 RMW). However, unlike a type-2 RMW, it appears strongly ordered only with respect to a write/RMW (but not a read) that synchronizes with the RMW. Therefore, Dekker's algorithm can be ported to TSO by replacing reads (but not writes) with type-3 RMWs. Similarly, C/C++11 can be ported to TSO by replacing SC-atomic-reads (but not SC-atomic-writes) with type-3 RMWs. Table 1 summarizes the use of type-1, type-2 and type-3 RMWs in various synchronization idioms.

In our final contribution, we propose efficient microarchitectural implementations of the weaker RMWs, which, in contrast to existing implementations, do not incur the cost of a write-buffer drain (§3). Our implementation of a type-2 RMW allows instructions following it to retire as soon as the read obtains exclusive ownership of the cache-line and locks it locally. The write simply retires into the tail of the write-buffer – thus the write-buffer drain is moved out of the critical path. To guarantee atomicity, coherence requests to the locked cache-line are denied until the write (of the RMW) and the pending writes prior to it complete. However, to prevent a potential deadlock we need to ensure that the above pending writes will eventually complete, and not be blocked by an RMW from another processor. We ensure this by tracking the list of unique RMW addresses in per-processor bloom filters. When a pending write (before the RMW) is found to conflict with the list of maintained RMW addresses, we revert to draining the write-buffer, thus avoiding the possibility of a deadlock (§3.2).

The type-3 RMW implementation is almost identical, with one difference. Since type-3 atomicity permits reads to the same ad-

Table 1. Conventional RMW (type-1) vs proposed RMWs (type-2, type-3)

Atomicity Definition	Dekker's with reads replaced by RMWs?	Dekker's with writes replaced by RMWs?	Dekker's with RMWs as barriers?	C/C++11 by replacing SC-atomic-reads with RMWs?	C/C++11 by replacing SC-atomic-writes with RMWs?
type-1	✓	✓	✓	✓	✓
type-2	✓	✓	✗	✓	✓
type-3	✓	✗	✗	✓	✗

dress as the RMW between the read and the write, the read need not obtain exclusive ownership of the cache-line – leading to a potentially more efficient implementation (§3.3). Our experimental results (§4) from benchmarks chosen from Splash-2, PARSEC, STAMP, and lock-free data structures show that in comparison with the existing type-1 RMW, our proposed type-2 RMW (type-3 RMW) is up to 58.9% (64.3%) cheaper, which translates into an overall performance improvement of upto 9.0%(9.2%)

We are not the first to propose weaker atomicity semantics for RMWs in general. In fact, Gharachorloo et al. [12] have already observed that it is sufficient for RMWs to use a type-3 definition for atomicity. However, in order for their TSO specification to be compliant with the original TSO specification, additional program order edges are added to RMWs, making the RMWs strongly ordered. In other words, by explicitly adding additional program order edges, the RMWs in their specification are effectively made equivalent to type-1 RMWs. In this paper, we consider the case in which the atomicity definitions are weakened, but additional program order edges are not added to the RMW. Besides, our proposed type-2 atomicity definition, to the best of our knowledge has not been considered before. More on related work in §5.

2. Semantics of TSO RMWs

In this section we will propose definitions of atomicity weaker than the standard strong definition for RMWs in TSO, and derive the ordering properties that apply. We will then use those ordering properties to demonstrate the use of weakened RMWs in synchronization – in particular, we will demonstrate when they are sufficient to implement the C/C++11 concurrency model.

We begin with recalling the base TSO model (without RMWs), and then add our new formulations of atomicity. The base TSO model follows Alglave [2], where our atomicity definitions fit most naturally. We present here only a brief introduction to Alglave's formulation published previously. Readers, particularly those familiar with alternative TSO formulations, should refer to Alglave's thesis for more details. The thesis has a proof of equivalence with the SPARC definition of TSO [25] is given, which is separately shown by Owens et al [22] to resemble the x86 multiprocessor model.

2.1 Base TSO

As usual in axiomatic memory models, we first derive a set of candidate executions from a program. Each candidate execution contains a set of events and relations over them, and represents a conceivable execution path (with control-flow unfolded, and values for each read in the program). In the next step, the memory model will carve out (via conditions on those relations) which of these candidate executions are allowed by the model.

The events (memory reads, writes, and barriers) are annotated with their thread, type, and for memory accesses the associated address and value. From the program we derive the program order (po) relation, a local (per-thread) total order over events from the same thread as they appear in the program. We also consider two relations which are existentially quantified over: a reads-from map (rf) and write-serialization (ws), both relations over events. The relation rf maps, for each read, the write that the read takes its value from to the read. The relation ws is a linear order per location relating all (and only) the writes to the same location, and represents the coherence order of the system (in prior work, this relation is also called coherence co).

For ease of stating the memory model, we derive various additional relations from the above. The from-reads relation (fr) relates a read to all writes to the same location that come after (in ws) the write it reads from (given by rf). The external-reads-from relation (rfe) is the subrelation of rf which is restricted to reads which read from a different-thread write. The communication relation com is the union of ws, rfe, and fr.

A preserved-program-order relation (ppo) relates all memory operations from the same thread in program order, according to TSO ordering rules. Thus it relates all memory operations, except writes to program order-subsequent reads: In other words, $W \xrightarrow{\mathsf{po}} W$, $R \xrightarrow{\mathsf{po}} W$, $R \xrightarrow{\mathsf{po}} R$ all belong to ppo also.

A barrier-separated relation (bar) relates memory operations (on the same thread) separated in program order by a memory barrier.

The behavior of a program is the set of corresponding execution witnesses which are valid. A valid execution witness is one where the union of com, ppo, and bar is acyclic, and satisfies the uniproc condition. The uniproc condition states that the relation com is consistent with the per-thread order of memory operations to the same location. The first condition says that a happens-before-like relation is acyclic. In this case we call a linear extension of of com, ppo, and bar the global-happens-before relation (ghb). Informally, it is the global memory order (also known as execution order) in which memory operations appear to perform.

2.2 Adding RMWs to the model

We now consider events coming from RMWs. These correspond to one read and one write to the same location – we denote the read part of the RMW as R_a and the write part of the RMW as W_a. In an RMW, the read part comes before the write in program order – consequently, the read R_a reads an earlier value and not the value written by W_a. In addition to this, R_a and W_a need to be performed atomically, where atomicity is one of the following three definitions:

- **Type-1 Atomicity.** This is a strict definition of atomicity, used by existing TSO RMWs [16, 26], that prevents *writes of any address* from appearing between the read and the write in the global memory order. More formally, with type-1 RMWs added to the TSO model, valid execution witnesses are ones which further impose that there is no event in ghb between R_a and W_a.
- **Type-2 Atomicity.** This is a weakening which only prevents *reads and writes of the same address as the RMW* from appearing between R_a and W_a in the global memory order. More formally: $\{\forall M(x) : M(x) \xrightarrow{ghb} R_a(x) \vee W_a(x) \xrightarrow{ghb} M(x)\}$.
- **Type-3 Atomicity.** This is a further weakening which merely prevents *writes of the same address as the RMW* from appearing between R_a and W_a in the global memory order. More formally: $\{\forall W(x) : W(x) \xrightarrow{ghb} R_a(x) \vee W_a(x) \xrightarrow{ghb} W(x)\}$.

It is important to note that even type-3 atomicity, the weakest of the atomicity definitions, satisfies the notion of atomicity required for solving the *consensus* problem [15] – consensus being the abstract problem that models synchronization idioms. Nonetheless, this does not imply that the three types of RMWs can be used interchangeably. In fact, we shall see that each of the three atomicities gives rise to RMWs that are ordered differently.

Atomicity-induced orderings. Each atomicity definition, by disallowing a specific set of memory operations between R_a and W_a in the global memory order – effectively requires both R_a and W_a of the RMW to be ordered identically with such disallowed memory operations. For example, if just R_a (and not W_a) is originally ordered before a disallowed memory operation M in the ghb ($R_a \xrightarrow{ghb} M$), then atomicity requires W_a to also be ordered before M ($W_a \xrightarrow{ghb} M$) – otherwise M could end up between R_a and W_a in the ghb. In other words, the atomicity constraint induces additional memory orderings – the atomicity relation ato is used to refer to such atomicity-induced orderings. In the above example, the ordering $W_a \xrightarrow{ato} M$ would be an atomicity-induced ordering. Accounting for such atomicity-induced orderings, the global memory order (ghb) is the linear extension of the union of com, ppo, bar, and ato. A valid execution witness, like before, is one which has an acyclic union of the above relations (including ato), and satisfies the uniproc condition. Next, we will derive the atomicity-induced memory ordering constraints for each of the atomicity definitions.

2.3 Type-1 RMWs

The strict type-1 definition of atomicity combined with TSO's preserved program order ensures that a type-1 RMW is strongly ordered with respect to memory operations before and after it.

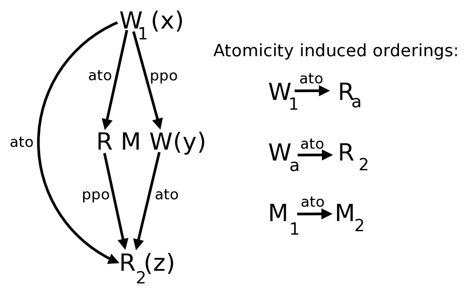


Figure 2. Additional memory orderings induced by type-1 RMW

Lemma 1. *An RMW placed between a write W_1 and a read R_2, results in the enforcement of $W_1 \xrightarrow{ato} R_a$, $W_a \xrightarrow{ato} R_2$ and consequently, $W_1 \xrightarrow{ato} R_2$.*

Proof. Type-1 atomicity mandates that either $W_a \xrightarrow{ghb} W_1$ or $W_1 \xrightarrow{ghb} R_a$. As shown in Fig. 2, $W_1 \xrightarrow{ppo} W_a$. This implies $W_1 \xrightarrow{ato} R_a$. Next, we prove the second part: $W_a \xrightarrow{ato} R_2$. As shown in Fig. 2, $R_a \xrightarrow{ppo} R_2$. This implies that either R_2 occurs after W_a in the ghb or R_2 is between R_a and W_a. Meanwhile, type-1 atomicity mandates that there cannot be any writes between R_a and W_a in the ghb; in particular there cannot be any writes to location z. This implies that even if R_2 were to occur between R_a and W_a, it can be safely be moved after W_a. This in turn implies $W_a \xrightarrow{ato} R_2$. Finally, $W_1 \xrightarrow{ato} R_2$, because of transitivity ($W_1 \xrightarrow{ato} R_a$ and $R_a \xrightarrow{ppo} R_2$). □

Such strongly ordered type-1 RMWs result in costly implementations that involve a write-buffer drain; however, they can be used to port synchronization idioms to TSO without requiring additional memory barriers. Below, we demonstrate how type-1 RMWs are used in various synchronization idioms:

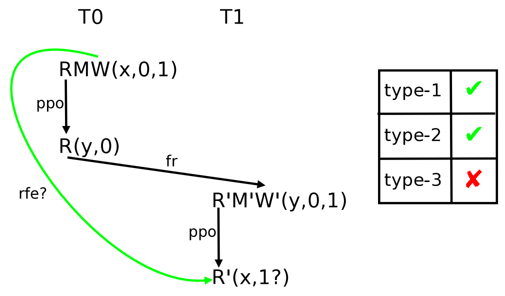


Figure 3. Dekker's with writes replaced by RMWs. In this and other examples that follow, $RMW(x, 0, 1)$ means that the RMW reads a value of 0 from location x and updates it to 1

Dekker's: write-replacement. One way to ensure that Dekker's algorithm works on TSO architectures is to replace the writes with type-1 RMWs as shown in Fig. 3 [16, 26] In the above example, we assume that the read $R(y)$ from thread 0 reads the initial value of 0. For Dekker's algorithm to work the read $R'(x)$ should read a value of 1. The following sequence of orderings ensure this: $W_a(x) \xrightarrow{ato} R(y) \xrightarrow{fr} W'_a(y) \xrightarrow{ato} R'(x)$ – where ato denotes the additional orderings induced by atomicity.

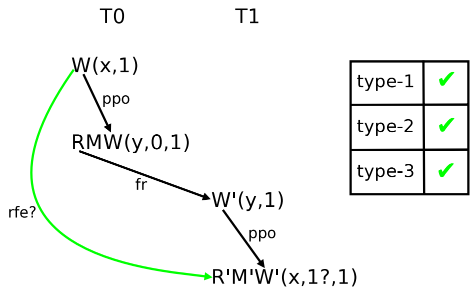


Figure 4. Dekker's with reads replaced by RMWs.

Dekker's: read-replacement. Using similar reasoning, it is easy to see that replacing reads with type-1 RMWs will also ensure that Dekker's algorithm works on TSO (Fig. 4).

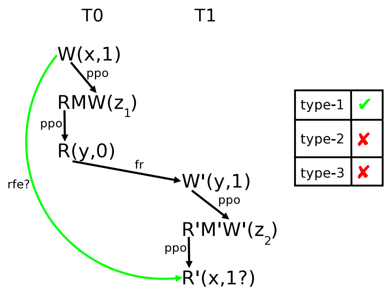


Figure 5. Dekker's with RMWs used as memory barriers. The two RMWs access different addresses z_1 and z_2.

Dekker's: RMWs as barriers. One simple way to make Dekker's algorithm work on TSO is to insert memory barriers between the writes and the reads, as the $W \rightarrow R$ ordering enforced by the memory barriers would ensure correctness. Since type-1 RMWs order memory operations before and after it, they can very well be used instead of the barriers. As shown in Fig. 5, the following sequence of ordering ensures correctness: $W(x) \xrightarrow{ato} R(y) \xrightarrow{fr} W'(y) \xrightarrow{ato} R'(x)$.

Implementing C/C++11 using type-1 RMWs. The C/C++11 concurrency model [7, 10] is an adaptation of *data-race-free-0* [1] which guarantees SC for data race free programs. It introduces a variety of atomic memory operations parameterized by different *memory order* parameters. Correct compilation depends (among other things) on mapping these atomic memory operations to hardware primitives. Batty et al. [6] recently proved that C/C++11 can be implemented on X86-TSO by mapping C/C++11 SC-atomic-reads and SC-atomic-writes to type-1 RMWs supported by x86 architectures (non-SC atomic reads and writes and non-atomic accesses can simply be mapped to ordinary TSO reads and writes). In fact, it is easy to adapt this proof and show that it is sufficient to map at least one of the SC-atomic-writes or the SC atomic reads to type-1 RMWs (see appendix). Informally, since TSO already preserves all program orders except the $W \rightarrow R$ order, we only need to ensure SC-atomic-writes are ordered with subsequent SC-atomic-reads; similarly to Dekker's algorithm, this can be accomplished by replacing either the reads or writes with type-1 RMWs.

2.4 Type-2 RMWs

We show that, unlike a type-1 RMW, a type-2 RMW placed between a write W_1 and a read R_2 does not explicitly enforce any of $W_1 \xrightarrow{ghb} R_a$, $W_a \xrightarrow{ghb} R_2$, or $W_1 \xrightarrow{ghb} R_2$. However, it disallows $R_a \xrightarrow{ghb} W_1$ and $R_2 \xrightarrow{ghb} W_a$ from being enforced [1] – in effect, a type-2 RMW is *implicitly ordered with respect to memory operations before and after it.*

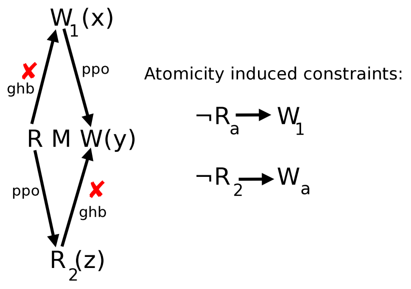


Figure 6. Memory ordering disallowed by a type-2 RMW

Lemma 2. *A type-2 RMW placed between two memory operations W_1 and R_2, disallows the enforcement of the following two orderings: $R_a \xrightarrow{ghb} W_1$ and $R_2 \xrightarrow{ghb} W_a$.*

Proof. Let us attempt to prove by contradiction by assuming the ordering $R_a \xrightarrow{ghb} W_1$ is enforced. Since there is no ppo edge directly connecting R_a and W_1, $R_a \xrightarrow{ghb} W_1$ we will need to be enforced via a sequence of edges as shown in Fig. 7. More specifically, there has to be a write $W'(y)$ which conflicts with $R_a(y)$ such that: $R_a(y) \xrightarrow{fr} W'(y) \xrightarrow{ghb} W_1(x)$. But, $R_a(y) \xrightarrow{fr} W'(y)$ implies $W_a(y) \xrightarrow{ato} W'(y)$, due to type-2 atomicity. This leads to a cycle: $W_a(y) \xrightarrow{ato} W'(y) \xrightarrow{ghb} W_1(x) \xrightarrow{ppo} W_a(y)$.

Similarly for the other part let us assume $R_2 \xrightarrow{ghb} W_a$. As shown in Fig. 7, this implies that there has to be a read $R''(y)$

[1] Disallowing an ordering $M_1 \xrightarrow{ghb} M_2$ (say) is not the same as enforcing $M_2 \xrightarrow{ghb} M_1$. The latter implies that M_2 will occur before M_1 in every valid global memory order, while the former implies that it is not necessary for M_1 to occur before M_2 in every valid global memory order

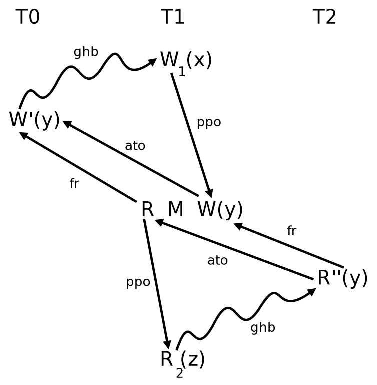


Figure 7. Scenario for proof of lemma 2.

which conflicts with $W_a(y)$ such that: $R_2(z) \xrightarrow{ghb} R''(y) \xrightarrow{fr} W_a(y)$. But, $R''(y) \xrightarrow{fr} W_a(y)$ implies $R''(y) \xrightarrow{ato} R_a(y)$, due to type-2 atomicity. This leads to a cycle: $R_a(y) \xrightarrow{ppo} R_2(z) \xrightarrow{ghb} R''(y) \xrightarrow{ato} R_a(y)$. □

Effect of implicitly ordered type-2 RMWs. Since a type-2 RMW neither enforces $W_1 \rightarrow R_a$ nor $W_a \rightarrow R_2$, it also does not transitively enforce $W_1 \rightarrow R_2$. Consequently, a type-2 RMW is not ordered like a memory barrier; in the next section we will propose an efficient implementation that does not incur the cost of a write-buffer drain. At the same time, a type-2 RMW appears to be strongly ordered with respect to any memory operation that synchronizes with the RMW i.e any memory operation from another thread that is to the same address as the RMW. As shown in Fig. 7, with respect to $W'(y)$ which synchronizes with R_a, W_1 appears to be ordered before the RMW. This is because, type-2 atomicity induces the ordering $W_a(y) \xrightarrow{ato} W'(y)$, which results in the sequence of orderings: $W_1(x) \xrightarrow{ppo} W_a(y) \xrightarrow{ato} W'(y)$, thereby ensuring $W_1(x) \rightarrow W'(y)$. Likewise, with respect to $R''(y)$ which synchronizes with W_a, $R_2(z)$ appears to perform after the RMW – the sequence of orderings $R_{\prime\prime}(y) \xrightarrow{ato} R_a(y) \xrightarrow{ppo} R_2(z)$ ensures this. Consequently, type-2 RMWs can seamlessly replace existing RMWs in synchronization idioms, as we will demonstrate next.

Dekker's: write-replacement. Similarly to type-1 RMWs, Dekker's algorithm will continue to work with writes replaced by type-2 RMWs as shown in in Fig. 3. Since $R(y) \xrightarrow{fr} W_a'(y)$, $R(y) \xrightarrow{ato} R_a'(y)$ (due to type-2 atomicity). Now, the sequence of orderings $R_a(x) \xrightarrow{ppo} R(y) \xrightarrow{ato} R_a'(y) \xrightarrow{ppo} R'(x)$ ensures that $R_a(x) \xrightarrow{ghb} R'(x)$. This in turn implies that $W_a(x) \xrightarrow{ato} R'(x)$, again due to type-2 atomicity.

Dekker's: read-replacement. Using a similar reasoning, replacing reads with type-2 RMWs will also ensure that Dekker's algorithm works on TSO (Fig. 4). Since $R_a(y) \xrightarrow{fr} W'(y)$, $W_a(y) \xrightarrow{ato} W'(y)$ (due to type-2 atomicity). Now, the sequence of orderings $W(x) \xrightarrow{ppo} W_a(y) \xrightarrow{ato} W'(y) \xrightarrow{ppo} W_a'(x)$ ensures that $W(x) \xrightarrow{ghb} W_a'(x)$. This in turn implies that $W(x) \xrightarrow{ato} R_a'(x)$, again due to type-2 atomicity.

Dekker's: RMWs as barriers (different addresses). A type-2 RMW cannot be used as a memory barrier in Dekker's algorithm if the RMWs used to replace the barriers access different addresses, since they would not appear strongly ordered with one another. As shown in Fig. 5, it can potentially allow the following sequence of operations – $R_a(z_1), R(y), R_a'(z_2),\ R'(x), W(x),$

$W_a(z_1), W'(x), W'_a(z_1)$ – which would lead to $R'(x)$ to read a value of 0.

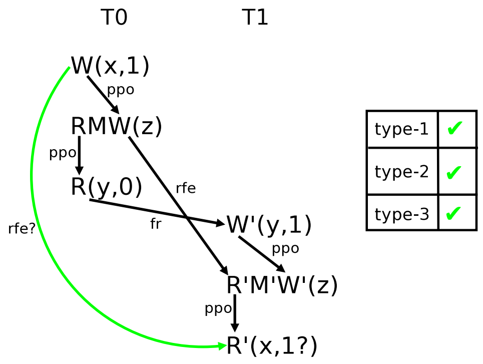


Figure 8. Dekker's with RMWs used as a memory barrier. The two RMWs access the same addresses z.

Dekker's: RMWs as barriers (same address). A type-2 RMW, however, can be used as a memory barrier in Dekker's algorithm if the inserted RMWs access the same address, since this ensures that the RMWs appear strongly ordered to one another. As shown in Fig. 8, type-2 RMWs used in the above fashion ensure that $R'(x)$ will read the correct value of 1. To see why, first recall that based on our assumption $R(y) \xrightarrow{fr} W'(y)$. This implies that $W_a(z) \xrightarrow{rfe} R'_a(z)$ (as the other possibility $W'_a(z) \xrightarrow{rfe} R_a(z)$ would result in the following cycle: $W'_a(z) \xrightarrow{rfe} R_a(z) \xrightarrow{ppo} R(y) \xrightarrow{fr} W'(y \xrightarrow{ppo} W'_a(z))$. This in turn leads to the sequence $W(x) \xrightarrow{ppo} W_a(z) \xrightarrow{rfe} R'_a(z) \xrightarrow{ppo} R'(x)$, ensuring that $R'(x)$ reads the correct value.

Implementing C/C++11 using type-2 RMWs. We formally show that, similarly to type-1 RMWs, C/C++11 can be implemented by mapping at least one of SC-atomic-writes or SC-atomic-reads to type-2 RMWs. Recall that, since TSO already preserves all program orders except the $W \rightarrow R$ order, we only need to ensure SC-atomic-writes are ordered with subsequent SC-atomic-reads. Intuitively, since type-2 RMWs appear strongly ordered when used in synchronization idioms, this can be accomplished by replacing either the SC-atomic-reads or SC-atomic-writes with RMWs. See appendix for the formal proof.

2.5 Type-3 RMWs

We show that, similarly to a type-2 RMW, a type-3 RMW placed between W_1 and R_2 does not explicitly enforce any of $W_1 \xrightarrow{ghb} R_a$, $W_a \xrightarrow{R}_2 ghb$, or $W_1 \xrightarrow{ghb} R_2$. However, unlike a type-2 RMW it disallows only $R_a \rightarrow W_1$ (but could allow $R_2 \rightarrow W_a$) – in effect, a type-3 RMW is implicitly ordered with respect to memory operations before it, but not with those after it.

Lemma 3. *A type-3 RMW placed between two memory operations W_1 and R_2, disallows $R_a \xrightarrow{ghb} W_1$ (but could allow $R_2 \xrightarrow{ghb} W_a$ to be enforced).*

Proof. Proof of $\neg R_a \xrightarrow{ghb} W_1$ is identical to the first part of the proof of lemma 2. To understand why $R_2 \xrightarrow{ghb} W_a$ is not disallowed, let us consider the second part of the proof of lemma 2, where we assumed $R_2 \xrightarrow{ghb} W_a$. As shown in Fig. 7, this implies that there has to be a read $R''(y)$ which conflicts with $W_a(y)$ such that: $R_2(z) \xrightarrow{ghb} R''(y) \xrightarrow{fr} W_a(y)$. Recall that type-2 atomicity

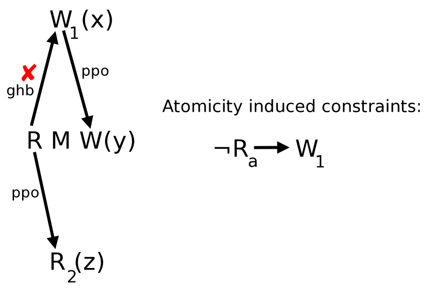


Figure 9. Memory ordering disallowed by a type-3 RMW

induced the ordering: $R''(y) \xrightarrow{ato} R_a(y)$, which led to a cycle. However, such an ordering is not induced by type-3 atomicity, which allows for reads to happen between the $R_a(y)$ and $W_a(y)$, and so there is no cycle. □

Effect of implicitly ordered type-3 RMWs. Since a type-3 RMW enforces neither $W_1 \rightarrow R_a$ nor $W_a \rightarrow R_2$, it also does not transitively enforce $W_1 \rightarrow R_2$. Consequently, a type-3 RMW is not ordered like a memory barrier. At the same time, a type-3 RMW appears to be strongly ordered with respect to any write/RMW that synchronizes with the RMW. As shown in Fig. 7, with respect to $W'(y)$ which synchronizes with R_a, W_1 appears to be ordered before the RMW. This is because type-3 atomicity induces $W_a(y) \xrightarrow{ato} W'(y)$, which in turn results in the sequence of orderings: $W_1(x) \xrightarrow{ppo} W_a(y) \xrightarrow{ato} W'(y)$ which ensures this. On the other hand, with respect to the read $R''(y)$ which synchronizes with W_a, $R_2(z)$ does not appear to be ordered after the RMW, since type-3 atomicity allows $R''(y)$ to occur between $R_a(y)$ and $W_a(y)$. Consequently, type-3 RMWs cannot seamlessly replace existing RMWs in synchronization idioms, as we will demonstrate next.

Dekker's: write-replacement. Unlike type-1 or type-2 RMWs, replacing writes with type-3 RMWs cannot guarantee correctness (Fig. 3). This is because type-3 atomicity is not able to induce $R(y) \xrightarrow{ato} R'_a(y)$. Hence, the following sequence is allowed: $R_a(x), R(y), R'_a(y), R'(x), W_a(x), W'_a(y)$ – which would lead to $R'(x)$ to read 0.

Other Dekker's scenarios. For the other Dekker's algorithm scenarios (Fig. 4, Fig. 5, and Fig. 8) a type-3 RMW behaves identically to a type-2 RMW.

Implementing C/C++11 using type-3 RMWs. We formally show that C/C++11 can be implemented by mapping SC-atomic-reads (and optionally SC-atomic-writes) to type-3 RMWs. However, it is not sufficient (unlike type-1 and type-2 RMWs) for only the SC-atomic-writes to be so mapped. Intuitively, since type-3 RMWs appear strongly ordered only when synchronizing with writes or RMWs, but not reads, all SC-atomic-reads need to replaced with RMWs. See appendix for the formal proof.

2.6 Summary

We show that type-2 RMWs can seamlessly replace type-1 RMWs in various synchronization idioms, except when a type-1 RMW is used purely as a memory barrier. Given that all modern TSO(-like) architectures have a dedicated memory barrier instruction, there is no need to use an RMW as a barrier. Furthermore, type-2 RMWs can still be used as a memory barrier provided such RMWs are forced to synchronize with each other (by forcing them to access the same address). Similarly to type-2 RMWs, type-3 RMWs also do not behave like memory barriers. However, unlike type-2 RMWs, type-3 RMWs only appear ordered with respect to writes/RMWs (but not reads) that synchronize with the RMW; thus type-3 RMWs cannot seamlessly replace type-1 RMWs. Nevertheless, we show that by replacing synchronization reads with type-3 RMWs, the above synchronization idioms can still be implemented using type-3 RMWs.

3. TSO RMWs: Implementation

In this section we first discuss how existing type-1 RMWs are implemented. We then describe our proposed type-2 and type-3 RMW implementations. For the following discussion we assume a chip multiprocessor with local L1 caches and a shared L2 cache; the local caches are kept coherent at the L2 cache using a distributed directory based coherence protocol.

3.1 Type-1 RMW

Recall that a type-1 RMW is strongly ordered with respect to memory operations before and after it: a type-1 RMW placed between write W_1 and read R_2 results in the enforcement of $W_1 \rightarrow R_a$ and $W_a \rightarrow R_2$, where R_a/W_a are the read/write of the RMW respectively. To enforce $W_1 \rightarrow R_a$, pending writes in the write-buffer (if any) must complete before R_a can retire.

Furthermore, type-1 atomicity mandates that there should not be any conflicting reads or writes (to the same address as the RMW) between R_a and W_a. To ensure this, existing RMW implementations use a cache-line locking mechanism [16, 20, 26]. The R_a obtains read/write permissions for the cache-line, and locks it before it retires, thereby denying coherence requests to the cache-line. Once W_a completes, the cache-line is unlocked.

To ensure that $W_a \rightarrow R_2$ is enforced, R_2 is allowed to retire only after W_a completes. In other words, reads that follow the RMW have to wait until: (a) all writes prior to the RMW are performed (the write-buffer is drained) and (b) R_a and W_a are performed. Thus, the type-1 RMW incurs the cost of a write-buffer drain and the cost of performing R_a and W_a.

Gharachorloo et al. [13] proposed two techniques to provide efficient memory ordering. Both these techniques can be used to improve the performance of type-1 RMWs. The first one involves issuing the read-exclusive request for all pending writes in parallel, to efficiently enforce the write-buffer drain. The actual writes, however, are completed in-order, keeping with TSO. Parallel issue of the read-exclusives will be serialized at the local L1 cache and at the directory, but will make full use of the interconnect and overlap invalidation and acknowledgement messages for all the pending writes. The second technique is to hide part of the write-buffer drain latency through in-window speculation. Here, the instructions following the RMW are speculatively executed, but are allowed to complete only after the RMW and all the pending writes before it complete.

3.2 Type-2 RMW

Recall that a type-2 RMW is not explicitly ordered with respect to memory operations before and after it in the program order. Since a type-2 RMW that is placed between memory operations W_1 and R_2, does not enforce $W_1 \rightarrow R_a$, R_a need not wait for the write-buffer to be drained. However, type-2 atomicity still disallows conflicting reads or writes from appearing between R_a and W_a in the global memory order. Similarly to a type-1 RMW, this is ensured using the cache-line locking mechanism. Like before, R_a obtains read/write permissions for the cache-line, locks the cache-line, and then retires. After this, W_a simply retires into the tail of the write-buffer. At this point the RMW effectively retires, and allows memory operations following it (e.g. R_2) to retire (since $W_a \rightarrow R_2$ is not enforced). Finally, when W_a reaches the head of the write-buffer and completes, the cache-line is unlocked.

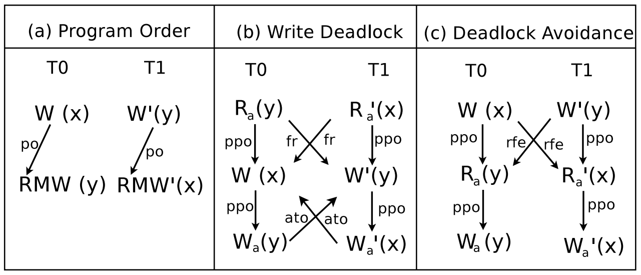


Figure 10. (a) shows a code segment that can cause a write-deadlock. (b) shows an execution order with a cyclic dependency of non-occurring events resulting in a write-deadlock. fr ordering

Write-deadlocks. The above implementation, while simple, can potentially result in a deadlock. To guarantee type-2 atomicity, coherence requests to the cache-line locked by an RMW are denied until W_a and the pending writes prior to it complete. If such a pending write W_1 is to a cache-line which has already been locked by another RMW$'$ from a different processor, then W_1 (and hence W_a) will have to wait until W'_a completes. If W'_a itself is stalled because of a similar write in its write-buffer, a deadlock manifests.

This is illustrated in the code segment shown in Fig. 10(a), where $W(x)$ occurs before RMW(y), and $W'(y)$ occurs before RMW$'(x)$ in program order. As shown in Fig. 10(b), let us assume that $R_a(y)$ and $R'_a(x)$ have retired after locking their respective cache-lines, while the writes ($W(x)$ and $W'(y)$) have retired into the write-buffer and are yet to complete. Cache-line locking ensures that $W(x)$ cannot complete until $W'_a(x)$ has completed, and $W'(y)$ cannot complete until $W_a(y)$ has completed. However, since writes are ordered in TSO, $W_a(y)$ cannot complete until $W(x)$ completes, and $W'_a(x)$ cannot complete until $W'(y)$ completes. This leads to a write-deadlock.

More formally, our assumptions can be represented by the two fr orderings: $R'_a(x) \xrightarrow{fr} W(x)$ and $R_a(y) \xrightarrow{fr} W'(y)$. Now, type-2 atomicity induces the two orderings: $W'_a(x) \xrightarrow{ato} W(x)$ and $W_a(y) \xrightarrow{ato} W'(y)$. This in turn results in a cycle: $W(x) \xrightarrow{ppo} W_a(y) \xrightarrow{ato} W'(y) \xrightarrow{ppo} W'_a(x) \xrightarrow{ato} W(x)$. Since each of the memory operations which are part of the cycle have not yet performed, a deadlock ensues.

Deadlock avoidance. In order to ensure that the deadlock scenario discussed above never occurs, we should guarantee that none of the pending writes before an RMW, are to cache-lines locked by other RMWs – the deadlock safety property. To ensure this, we propose a mechanism to dynamically maintain the set of unique RMW addresses accessed by RMWs from all processors – the *addr-list*. Furthermore, we make this addr-list available locally to each of the processors.

Now, when an RMW is performed, if none of the pending writes in the write-buffer conflict with the addr-list, we can safely say that these writes are not to locked cache-lines. On the other hand, if any of the pending writes conflicts with the addr-list, the deadlock safety property is not guaranteed. In such a case, we revert to type-1 implementation by draining the write-buffer before performing the RMW – thereby avoiding a deadlock.

There are two challenges to efficiently implementing this mechanism in hardware: (a) keeping track of the RMW addresses in the addr-list efficiently; (b) keeping the addr-list coherent across all processors. We implement the addr-list using a bloom filter [8], which is a well understood mechanism for maintaining sets and supporting membership queries. In order to keep the addr-list coherent we simply broadcast the address whenever a new RMW address is encountered by a processor. Our design exploits the fact that the number of unique RMW addresses is relatively small – our experiments show that typically around 1% of the number of dynamic RMWs are to unique addresses. This in turn means that the addresses of the RMWs can be stored efficiently in a relatively small-sized bloom filter, with a low probability of false positives. More importantly, the number of broadcasts required to keep the addr-list coherent is minimal.

We now explain the working of our mechanism in more detail. When an RMW is ready to perform, we first query the bloom filter for the RMW address. If the address is not found in the filter, we insert the RMW address into the local bloom filter. In addition to this, since the addr-list has changed, we broadcast the new address to all processors. Each of the other processors, upon receiving the address, inserts the address into its respective bloom filter and sends back an acknowledgement. Once all acknowledgements have been received (or if the RMW's address is found in the addr-list in the first place), we query the bloom filter with the pending writes' addresses. If any of these write addresses are found in the addr-list, this flags a potential deadlock. Consequently, the write-buffer is drained before performing the RMW like a type-1 RMW. On the other hand, if none of the pending writes' addresses are found, the RMW does not wait for the write-buffer to drain. It locks the cache-line and simply retires, while the write of the RMW is retired into the write-buffer.

To see why our scheme is correct note that an RMW can lock the cache-line and retire (with pending writes in the write-buffer) only when:

- c1: the RMW's address is made visible to all processors
- c2: none of the pending writes conflict with the addr-list.

Now, c1 implies that any write (W') that could be potentially involved in a deadlock with the original RMW will conflict with the local addr-list. c2 implies that an RMW with W' in its write-buffer will revert to type-1, thereby avoiding a deadlock. Consider the deadlock scenario shown in Fig. 10(c). Recall that in the deadlock scenario, $R_a(y)$ and $R'_a(x)$ have retired, but their respective pending writes $W(x)$ and $W'(y)$ are unable to complete (inducing the two fr orderings: $R'_a(x) \xrightarrow{fr} W(x)$ and $R_a(y) \xrightarrow{fr} W'(y)$). The fact that $R_a(y)$ and $R'_a(x)$ have retired implies that both x and y must be present in the bloom filter (from c1). In addition to this, $W(x)$ and $W'(y)$ should have checked the filter for conflicts (from c2). The assumed fr orderings imply that neither of the writes conflicted with the bloom filter. This in turn implies that neither x nor y are in the bloom filter leading to a contradiction.

False Positives. Bloom filters suffer from false positives. The correctness of our scheme, however, is not compromised. The false positive may result from either an RMW or a pending write checking the bloom filter. When an RMW, whose address has not been encountered before, queries the bloom filter and the bloom filter returns a false positive, the RMW address ends up not being broadcast. This is safe, however, because any write which conflicts with this address will also similarly return a false positive. It is worth noting that false positives in this case may reduce the number of RMW broadcasts. Likewise, When a pending write queries the bloom filter and the filter returns a false positive, the write-buffer is unnecessarily drained. The correctness of the mechanism, however, is not affected.

Finally, in our design, the bloom filter keeps track of RMW addresses of all contexts. In other words, each bloom filter is independent of the thread context. While this may increase the probability of false positives, it again does not present any correctness issues.

It is worth noting that, the probability of false positives in the filter increases with the number of elements inserted into it, leading to a performance degradation over time. To handle this, we reset the bloom filters of all processors when the number of RMW addresses inserted into the filter exceeds a certain threshold, which is a function of the bloom filter configuration. To ensure correctness, when a processor receives a reset request, it waits until all in-flight RMWs have completed, and responds subsequently.

3.3 Type-3 RMW

Recall that a type-3 RMW, like a type-2 RMW, is not explicitly ordered with respect to memory operations before and after it. R_a need not wait for the write-buffer to be drained – it can retire even if there are pending entries in the write-buffer. However, type-3 atomicity still disallows conflicting writes and other RMWs from appearing between R_a and W_a in the global memory order. Since reads to the same memory address can appear between R_a and W_a, it is sufficient for R_a to get read permissions for the cache-line, unlike type-1/type-2 RMW which require read/write permission.

If the RMW is to a cache-line owned by the local cache, then it is locked in the cache itself before retiring R_a, similar to type-1/type-2 RMWs. If the RMW is to cache-line in *shared* state, however, locking the cache-line locally cannot prevent an RMW from another processor, which also has the cache-line in its local cache, from performing. To resolve this, we propose a *directory locking* protocol, wherein R_a to a cache-line in shared state is locked in the directory by transitioning the cache-line to a *locked* state. When W_a is issued from the write-buffer, the cache-line is transitioned out of the locked state allowing subsequent coherence requests to the cache-line to be serviced. This optimization removes any invalidation delay, incurred by the RMW, from the critical execution path.

Once R_a obtains a lock and retires, W_a simply retires into the tail of the write-buffer. At this point the RMW effectively retires, and allows memory operations following it to retire. Thus reads that follow a type-3 RMW will only have to wait until R_a obtains read permission for the cache-line and locks it. Finally, when W_a reaches the head of the write-buffer and completes, the lock on the cache-line is released. Similarly to type-2 RMWs, the implementation of type-3 RMWs also makes use of the bloom filter mechanism to avoid deadlocks.

4. Experimental Evaluation

The primary goal of our experiments was to compare the cost of type-1, type-2, and type-3 RMWs. Furthermore, we evaluated the impact of the different types of RMWs on the overall execution time of the benchmark programs. Since RMWs are also used to implement C/C++11 SC-atomic-reads and/or SC-atomic-writes, we also investigated the performance of supporting C/C++11 concurrency model with type-1, type-2 and type-3 RMWs. We briefly describe our implementation before discussing the results.

4.1 Implementation

Table 2. Architectural Parameters

Processor	32 core CMP, inorder
Write Buffer	32-entry deep
L1 Cache	private, 32 KB 4-way 2-cycle latency
L2 Cache	shared, 1 MB per-core, 16-way 6-cycle latency
Memory	300 cycle latency
Coherence	MOESI distributed directory
Interconnect	2D Mesh, 1-cycle link, 4-cycle router latency

Simulator. We use the GEM5 simulator to implement our baseline system, which is an x86-based CMP composed of inorder processors, with local L1 caches and a shared-distributed L2 cache.

Table 3. Benchmark Characteristics

Code	Suite	Problem Size	Ratio of RMWs per 1000 memops	% Unique RMWs	% write-buffer drains for type-2/type-3 RMW	RMW broadcasts per 100 RMW ops
radiosity	SPLASH-2	room	15.56	0.28	0.06	0.26
raytrace	SPLASH-2	car	13.83	0.02	0.12	0.02
fluidanimate	PARSEC	simmedium	17.43	0.46	0.09	0.46
dedup	PARSEC	simmedium	8.10	3.31	0.20	3.12
bayes	STAMP	bayes+	34.15	0.91	0.01	0.80
genome	STAMP	genome+	6.19	0.64	0.10	0.52
wsq-mst	Lockfree	10000 nodes	23.41	3.80	0.07	3.71

Cache latencies were obtained from CACTI [21]. The baseline uses type-1 RMWs. The local caches are kept coherent using a distributed directory based on the MOESI coherence protocol. We chose inorder cores for our simulation as the GEM5's out-of-order processor model is unstable for full system simulation of the x86 processor architecture. The choice of inorder cores, however, is a valid design point owing to the fact that several present and future many-core processors, like the Intel Xeon Phi, Sun Niagara T2, and NVIDIA GPUs, make use of inorder cores as opposed to out-of-order cores to achieve better performance to power ratios. As mentioned in the previous section, we implemented a parallel write-buffer drain mechanism. This improves the baseline significantly over the serial write-buffer drain. We did not implement in-window speculation as it is not applicable to inorder processors. The architectural parameters for our implementation are presented in Table 2.

We modified the simulator to implement type-2 and type-3 RMWs with deadlock avoidance. In our implementation, we used a 128B bloom filter with 3 hash functions. It is worth noting that the only hardware overhead for type-2/type-3 RMWs is the 128B bloom filter and a RMW threshold counter per processor. Also, we did not make use of the threshold counter in our simulations as we ran only a single context which did not require a bloom filter reset for good performance.

Benchmarks. We evaluate our technique using benchmarks in Table 3, which includes both lock-based and a lock-free program. *radiosity* and *raytrace* are benchmarks from the Splash-2 suite which primarily use RMWs in lock/unlock primitives. Similarly, *fluidanimate* and *dedup* (from PARSEC) are also lock-based benchmarks. It is worth noting here that we chose only the top two benchmarks from each suite, in terms of the ratio of RMW instructions to other memory operations. We do this as traditional lock-based algorithms are highly scalable and do not communicate (or synchronize) very much and thus do not benefit from reducing the cost of RMWs. On the other hand, lock-free programs use more RMWs taking advantage of low-latency communication on multicores. *wsq-mst* is a lock-free parallel spanning tree algorithm [4] using Chase-Lev work stealing queue. *bayes* and *genome*, from the STAMP (using TL2 [11]), use RMWs for locking writes in transactions and to commit transactions. We ran the benchmarks in their regions of interest, with the input sizes mentioned in Table 3.

C/C++11 concurrency. Because of the recency of the C/C++11 concurrency model, there is no corpus of C/C++11 code to test our ideas on. We therefore modified the *wsq-mst* program to make use of atomic reads/writes as prescribed by the C/C++11 model. *wsq-mst* uses Dekker-like synchronization to update the task queue pointers while removing tasks from the queue; thus the read and write of this synchronization primitive corresponds to an SC-atomic-read and SC-atomic-write respectively. As mentioned earlier, the C/C++11 concurrency model can be realized by replacing SC-atomic-writes and/or SC-atomic-reads with RMWs. We compare the performance of the different types of RMWs by replacing either the SC-atomic-reads (*wsq-mst_rr*) or SC-atomic-writes (wsq-mst_wr) with RMWs. We do not consider type-3 RMWs for write replacement here as that cannot guarantee correctness (as described in §2.5).

4.2 Cost of RMWs

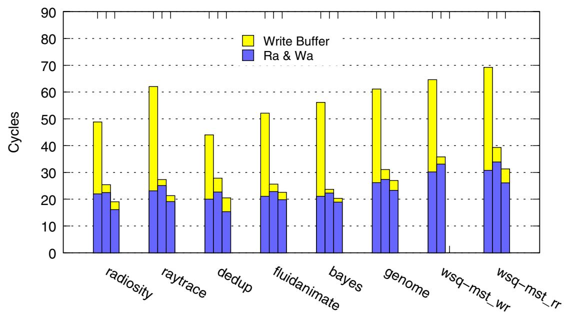
We split the cost of an RMW in two parts: the cost of performing the read and write (R_a/W_a); and the cost of handling the writes in the write-buffer. The average cost of an RMW across the chosen benchmarks for type-1, type-2, and type-3 RMWs is presented in Fig. 11(a). As we can see, RMWs are expensive – the average cost of type-1 RMWs is as high as 69 cycles. We also observe that the write-buffer drain significantly contributes to the overall cost of an RMW (58.0% on average). We can infer from this that a significant number of RMWs have at least one write in the write-buffer which needs to send out invalidation requests. Also, a significant number of RMWs are to shared cache-lines which explains the cost contributed by R_a/W_a.

Using type-2 RMWs, the cost of an RMW reduces by 38.6%-58.9% when compared to type-1 RMWs across the benchmarks. As seen from Fig. 11(a), a significant portion of the performance improvement is by avoiding the write-buffer drain in the general case. Recall that we revert to a write-buffer drain, when a write hits in the bloom filter. As seen from Table 3, the average number of hits of pending writes in the bloom filter is negligible for each benchmark, and is sometimes zero. This explains the low write-buffer drain cost for type-2 and type-3 RMWs. It is worth noting that the cost of R_a/W_a itself slightly increases when compared with type-1 RMWs as a portion of the RMWs require broadcasts in addition to the invalidation request. The number of such RMW broadcasts depends on the accuracy of the bloom filter. As shown in the table, the percentage of RMWs that require a broadcast is less than 1.0% for most lock-based benchmarks except for *dedup* (3.1%), which has a higher ratio of unique RMWs to begin with. We have not presented the increase in network traffic due to RMW broadcasts, as this number is negligible across all chosen benchmarks (<0.5%).

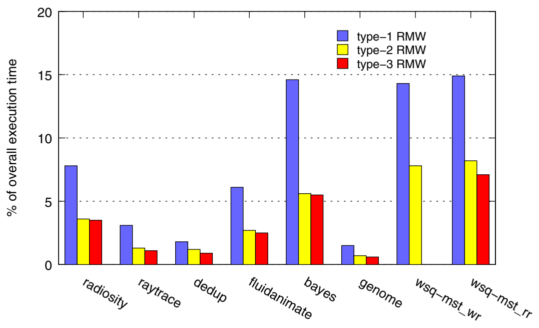
Type-3 RMWs reduce the cost of the RMW even further. The average cost of a type-3 RMWs is lower than type-1 RMWs by up to 64.3%. Type-3 RMWs reduce the cost of R_a/W_a but incur a similar write-buffer drain delay as type-2 RMWs.

C/C++11 concurrency. Similarly to lock-based benchmarks, we observe that using type-2 RMWs reduces the average cost of RMWs by 44.6% (write-replacement), and 43.2% (write-replacement) respectively, over type-1 RMWs. As mentioned earlier, type-3 RMWs cannot be used for write-replacement. For read-replacement, type-3 RMWs provide an additional 11.6% improvement over type-1 RMWs.

It is worth noting that the cost of RMWs in read-replacement (*wsq-mst_rr*) is higher than in write-replacement (*wsq-mst_wr*) for all types of RMWs; with read replacement, there are more entries



(a) Cost of type-1(left), type-2(center), and type-3(right) RMWs



(b) Execution time overhead of RMWs

Figure 11. Experimental Results

in the write-buffer per-RMW, which increases draining cost. The cost of R_a/W_a, however, is oblivious to whether SC-atomic-read or SC-atomic-write were replaced. In case of type-2 RMWs, we observe that the number of writes conflicting with the bloom filter increases, thereby increasing the cost of an RMW. Also note that this lock-free program, unlike traditional benchmarks, have more RMW broadcasts (3.7%) owing to a relatively larger number of unique RMWs. This affects the performance of type-2 and type-3 RMWs. However, the write-buffer drain cost eclipses the broadcast overhead.

4.3 Execution time overhead

Although we achieve a significant reduction in the cost of an RMW in all chosen benchmarks, its impact on the overall execution time depends on the ratio of RMW operations to other memory operations. We call this the *density* of RMWs. Thus, benchmarks with a larger RMW density benefit more from cheaper RMWs. Table 3 shows the ratio of the number of RMWs to the number of other memory operations in each of the benchmarks. From Fig. 11(b) shows the impact of RMWs on the overall execution time for all the chosen benchmarks. As expected, lock-free algorithms suffer more from expensive RMWs than lock-based algorithms. Similarly, *bayes* and *wsq-mst* also spend a lot of time performing RMWs. Although *genome* is a lock-free benchmark, the impact of RMWs on the overall execution time is less owing to a lower RMW density. This is because genome performs a lot more operations per transaction. As for lock-based benchmarks, *radiosity* and *fluidanimate* spend more than 5.0% of their execution time on RMWs. This, however, is not the case with *raytrace* and *dedup*. This is a result of the effort put into optimizing traditional lock-based benchmarks. We can extrapolate that other benchmarks from Splash-2 and Parsec will show an even lesser impact of RMWs.

With type-2 RMWs, we get up to 9.0% reduction for *bayes*, where the write-buffer drain almost but eliminated, as seen from Table 3. We also observe a significant reduction in the contribution of RMWs to the overall execution time in all other lock-free benchmarks as well. Even *radiosity* and *fluidanimate* show a reduction in overall execution time albeit lesser than 4%. Type-3 RMWs further improve the overall performance over type-2 RMWs, but only by a minimal amount ($<$0.5%).

C/C++11 concurrency. As for the C/C++11 concurrency model, replacing read atomics with RMWs results in a slightly higher overhead of RMWs as can be seen from the figure. The best performance can be obtained by replacing read atomics with type-3 RMWs (7.7% improvement over type-1 RMWs).

In summary, type-2 and type-3 RMWs are significantly cheaper than type-1 RMWs across all chosen benchmarks. This translates to a significant reduction in the overall execution time for the lock-free work stealing queue program which exhibits a higher RMW density. Traditional lock-based programs also show an improvement in performance. This improvement, however, is only visible in programs with a high RMW density. Other benchmarks show a negligible improvement in performance.

5. Related Work

Memory ordering. Over the years, researchers have proposed a number of techniques for achieving memory ordering efficiently [9, 14, 17, 18, 23]. While any of the above techniques can be used to efficiently implement the barrier-like ordering of a type-1 RMW, the goal of our work, however, is orthogonal. Instead of striving to implement the barrier-like ordering, we ask the question as to why a TSO RMW should be ordered like a memory barrier in the first place. Indeed, as we have shown through our weaker type-2/type-3 RMWs, implementing a barrier-like ordering is not necessary.

Weaker atomicity RMWs. Gharachorloo et al. [12] were the first to observe that it is sufficient for RMWs to use type-3 atomicity in the context of various memory consistency models. However, in order for their TSO specification to be compliant with the original TSO specification, they then added additional program order edges to RMWs, making the RMWs strongly ordered – hence equivalent to type-1 RMWs.

The load-reserve/store-conditional instruction is a classic example of an RMW in weaker models such as Power [19] which uses type-3 atomicity semantics. None of the mainstream TSO architectures, however, provide such an RMW. However, even if a TSO architecture were to support such an RMW, it would be ordered like a type-1 RMW. Because of its speculative nature, memory operations following such an RMW can only be retired after the store-conditional succeeds, and thus, such memory operations will have to wait for pending writes in the write-buffer, making the store-conditional act as a full barrier.

Hardware locking mechanisms. There have been several proposals (e.g. [28]) which address issues related to hardware based locking mechanisms. It is worth noting that these locks refer to the synchronization primitive as a whole and not the RMW instructions used in these primitives. These proposals primarily deal with lock contention and fairness. Our proposal is orthogonal to such work as we deal with the overhead added by the RMW to the local thread.

6. Conclusion

We observed that the atomicity semantics of an RMW is the key factor which affects the RMW's ordering semantics, its programmability, and its implementation cost. Existing TSO RMWs use a strict definition of atomicity (type-1) which results in the RMW being strongly ordered like a memory barrier. Whereas type-1 RMWs are costly to implement, they can be easily used in synchronization idioms on TSO without requiring additional memory barriers. In this paper, we proposed two weaker atomicity definitions: type-2 and type-3 atomicity; we formally derived how type-2 and type-3 RMWs would be ordered, and demonstrated that the resultant ordering is strong enough to implement various synchronization idioms using the weaker RMWs. We then proposed efficient architectural implementations of the weaker RMWs – experimental results show that our proposed type-2 RMW (type-3 RMW) is 58.9% (64.3%) cheaper than an existing type-1 RMW on average.

Based on our analysis and experimental evidence, type-2 RMWs, while performing almost as well as type-3 RMWs, are also able to seamlessly replace existing type-1 RMWs in common synchronization idioms – except in situations where an RMW is used as a memory barrier. Thus, they appear to be a promising alternative to existing type-1 RMWs. We also show how the proposed type-2 and type-3 RMWs can be used to implement C/C++11 atomics – thus making it possible for the compiler to transparently utilize the proposed RMWs to realize C/C++11 more efficiently.

A. C/C++11 implementation proofs

Recall that the C/C++11 concurrency model [7, 10] has marked memory accesses of various kinds (only SC is important on TSO, the properties of the others are automatically satisfied by normal reads and writes on TSO). We work with the formal description in Batty et al [6]. For a particular execution of a program, various relations among the actions corresponding to these operations are defined, including a happens-before relation; modification order mo, a total order per atomic location on writes to that location; and SC order sc, a total order on all SC atomic actions in the execution. There are several consistency conditions which these relations must satisfy for the execution to be consistent (briefly, both mo and sc must be consistent with happens-before; the ithb part of happens-before must be acyclic; certain shapes contradicting coherence must not occur within happens-before; and reads must read from a happens-before consistent write). Furthermore, if any consistent execution in the sense above has a data race, then the program as a whole has no defined semantics.

Correct compilation to TSO depends (among other things) on mapping the atomic accesses to TSO hardware primitives. Batty et al [6] prove correctness for a few variant mappings on X86-TSO; specifically, the *read-write-mapping* of Table 4(a) (from a prototype by Terekhov [27]), which maps SC-atomic-reads and SC-atomic-writes to X86-TSO RMWs. It is easy to adapt their proof and weaken the mapping, making only the SC-atomic-reads RMW's as in Table 4(b): *read-mapping*, or only the SC-atomic-writes RMW's as in Table 4(c): *write-mapping*. We now show that each mapping above would suffice for correctly implementing C/C++11 using type-2 RMWs (and reprove for type-1), while for type-3 RMWs, the read-write mapping and the read-mapping work. The write-mapping would not work for type-3 RMWs, by Dekker's counterexample in the paper (Fig. 3).

A.1 A generic outline of the proof strategy

The proof is fairly standard, following the proofs in [5, 6]. In particular, the way of constructing SC orders is derived from the earlier paper.

Mapping read-from maps, and mo First, the events occurring in the hardware models are related to the C/C++ actions from the corresponding program. For everything except the C/C++11 SC atomics, this is straightforward, as ordinary reads and writes correspond to C/C++11 reads and writes. For the SC actions, we assume that there is a unique mapping that can be derived. Then the hardware rf relation corresponds to the reads-from map of C/C++11, and the hardware ws relation (restricted to *atomic* locations) corresponds to mo of C/C++11.

ghb contains the C/C++11 ithb Here we notice that under any mapping (and any kind of RMW), each of the components of C/C++11 inter-thread-happens-before are part of ghb, by the construction via release sequences. Thus the ghb is a greater relation than the C/C++11 ithb.

Constructing the C/C++11 SC order. This part of the proof crucially depends on the mapping, so we will have to parametrize the proof by the mapping. We consider, as in the proof of SC actions on Power [5], an arbitrary linearization of the union of $\mathsf{po}_{\mathsf{SC}}$, program-order on SC actions; $\mathsf{ws}_{\mathsf{SC}}$, ws restricted to SC actions; $\mathsf{fr}_{\mathsf{SC}}$, which relates SC reads to all SC writes to the same location coherence-after the write the read reads from; and $\mathsf{erf}_{\mathsf{SC}}$, which relates a SC read and the last SC write in coherence before the write, or that write if a SC write, that the read reads-from.

We will then show that these relations are included in the ghb relation, and thus their union is consistent with ghb. As a corollary, by the acyclicity of ghb, we get that the union is acyclic and thus can be extended to a linear SC order.

C/C++11 concurrency. Assuming we can construct the SC order as above, we are now in a position to verify the consistency in C/C++11 of all behaviors permitted by TSO (with the variant RMWs) for race-free C/C++11 programs:

- Acyclicity of ithb: First, the ithb is contained within ghb, which is acyclic.
- Consistency of happens-before and mo: Second, mo should be consistent with C/C++ happens-before (which we get by ws being included in ghb, and the uniproc condition).
- Coherence diagrams: Third, the coherence diagrams [6] CoRR, CoRW, CoWR, and CoWW, must not be contradicted by the happens-before, which we get by the construction of ghb.
- Consistency of SC order: Fourth, sc should be consistent with happens-before and mo, which we get by our construction of sc.
- Reads read from a consistent write: Fifth, SC reads must read-from a write not happens-after the sc-last SC write, which we get by construction of sc. Other reads must read from a happens-before consistent write, where we note that all reads read from the last write to the same location in ghb. It is possible, however, that there is no C/C++11 happens-before relating the read and write (hb is smaller than ghb). Then, we find a race in the original C/C++ program, contradicting the race-free assumption.
- Constructing a race: Suppose we have found a read and a write that it reads-from that are not C/C++11 happens-before related. We find the minimal such pair in ghb (we know ghb is acyclic, so this is well-founded). Cut off the program without this read, and anything program-order after that write. Now we add back the read, but read from a C/C++11 allowed write; and it races with the original write. We complete the program execution in any consistent way, to get a racy consistent execution. Note that without speculative execution as in Power, this proof is much simpler than the corresponding proof for Power [5].

A.2 Instantiating the generic proof

Now we fill in the pieces above for each atomicity definition and each mapping. The remaining obligation is finding events in the TSO execution corresponding to the C/C++11 SC atomics, and proving that $\mathsf{po}_{\mathsf{SC}}$, $\mathsf{ws}_{\mathsf{SC}}$, $\mathsf{fr}_{\mathsf{SC}}$, and $\mathsf{erf}_{\mathsf{SC}}$ are contained within ghb.

Read-write-mapping and read-mapping. For these mappings, we consider the write W_a of the RMW for the SC read, and the write

Table 4. Mapping from C/C++11 to X86

(a) read-write-mapping

Operation	x86 Impl.
non-SC read	mov
SC read	lock xadd(0)
non-SC write	mov
SC write	lock xchg

(b) read-mapping

Operation	x86 Impl.
non-SC read	mov
SC read	lock xadd(0)
non-SC write	mov
SC write	mov

(c) write-mapping

Operation	x86 Impl.
non-SC read	mov
SC read	mov
non-SC write	mov
SC write	lock xchg

(either by itself in the read-mapping, or from the RMW for the read-write-mapping) for the SC write. Then $\mathsf{po}_{\mathsf{SC}}$is a part of ghb (they are same-thread writes). $\mathsf{ws}_{\mathsf{SC}}$is a part of ghb by definition of write-serialization. Every $\mathsf{fr}_{\mathsf{SC}}$edge must be consistent with ghb, since the subsequent write cannot be in ghb between R_a and W_a of the RMW, using any atomicity definition. Every $\mathsf{erf}_{\mathsf{SC}}$edge must be consistent with ghb, since the write read-from must be coherence-before the W_a of the SC read, and cannot come between R_a and W_a in any atomicity definition.

Write-mapping Here SC reads are mapped to plain reads, and thus there is no write to use as above. Instead, we use the read as is for SC reads, and the read R_a of the RMW for the SC write. Using this mapping, $\mathsf{po}_{\mathsf{SC}}$is a part of ghb (they are same-thread reads). For write-serialization, $\mathsf{ws}_{\mathsf{SC}}$is a part of the ghb, since R_a of each write must be before that write in fr. Likewise, $\mathsf{erf}_{\mathsf{SC}}$is a part of ghb, but the proof has two cases. For same threads, R_a of the write is ghb-before the read (same-thread reads). For different threads, R_a from the write is ghb before W_a in fr, and W_a is before the SC read in rfe. The last piece required is $\mathsf{fr}_{\mathsf{SC}}$. The SC read is certainly before in fr W_a of the RMW, but we are now considering R_a as representing the SC action. For Type-1 and Type-2 RMWs, it is consistent to impose that the SC read is before R_a, since they are to the same location, and no same-location actions can be in ghb between R_a and W_a. Then we get the required result.

For Type-3 RMWs, since a *read* can be in between R_a and W_a of a RMW, this strategy will not work. This is the point where the proof fails for Type-3 RMWs.

Acknowledgements

We would like to thank Peter Sewell and the anonymous reviewers for their helpful comments and advice for improving this paper. This work is supported by the Centre for Numerical Algorithms and Intelligent Software, funded by EPSRC grant EP/G036136/1 and the Scottish Funding Council to the University of Edinburgh. Susmit Sarkar was supported by EPSRC grant EP/H027351.

References

[1] S. V. Adve. *Designing memory consistency models for shared-memory multiprocessors*. PhD thesis, Madison, WI, USA, 1993. UMI Order No. GAX94-07354.

[2] J. Alglave. *A Shared Memory Poetics*. PhD thesis, 2010.

[3] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. T. Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *POPL*, pages 487–498, 2011.

[4] D. A. Bader and G. Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors (smps). *J. Parallel Distrib. Comput.*, 65(9):994–1006, 2005.

[5] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell. Clarifying and compiling C/C++ concurrency: from C++11 to POWER. In *Proc. POPL*, 2012.

[6] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *POPL*, pages 55–66, 2011.

[7] P. Becker, editor. *Programming Languages — C++*. 2011. ISO/IEC 14882:2011. A non-final recent version is available at `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf`.

[8] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[9] C. Blundell, M. M. K. Martin, and T. F. Wenisch. Invisifence: performance-transparent memory ordering in conventional multiprocessors. In *ISCA*, 2009.

[10] *Programming Languages — C*. 2011. ISO/IEC 9899:2011. A non-final recent version is available at `http://www.open-std.org/jtc1/sc22/wg14/docs/n1539.pdf`.

[11] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *DISC*, pages 194–208, 2006.

[12] K. Gharachorloo, S. Adve, A. Gupta, J. Hennessy, and M. Hill. *Specifying system requirements for memory consistency models*. Computer Systems Laboratory, Stanford University, 1993.

[13] K. Gharachorloo, A. Gupta, and J. L. Hennessy. Two techniques to enhance the performance of memory consistency models. In *ICPP (1)*, pages 355–364, 1991.

[14] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is sc + ilp=rc? In *ISCA*, pages 162–171, 1999.

[15] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13:124–149, January 1991.

[16] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Number 253669-033US. December 2009.

[17] E. Ladan-Mozes, I.-T. A. Lee, and D. Vyukov. Location-based memory fences. In *SPAA*, pages 75–84, 2011.

[18] C. Lin, V. Nagarajan, R. Gupta, and B. Rajaram. Efficient sequential consistency via conflict ordering. In *ASPLOS*, pages 273–286, 2012.

[19] I. B. Machine and A. C. I. Staff. *PowerPC Microprocessor Common Hardware Reference Platform: A System Architecture*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1995.

[20] M. Michael and M. Scott. Implementation of atomic primitives on distributed shared memory multiprocessors. In *Proc. HPCA*, 1995.

[21] N. Muralimanohar and R. Balasubramonian. Cacti 6.0: A tool to understand large caches.

[22] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *Proc. TPHOLs*, 2009.

[23] A. Singh, S. Narayanasamy, D. Marino, T. D. Millstein, and M. Musuvathi. End-to-end sequential consistency. In *ISCA*, pages 524–535, 2012.

[24] D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan and ClayPool Publishers, 2011.

[25] C. SPARC International, Inc. *The SPARC architecture manual (version 8)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[26] C. SPARC International, Inc. *The SPARC architecture manual (version 9)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[27] A. Terekhov. Brief tentative example x86 implementation for C/C++ memory model. `cpp-threads` mailing list, `http://www.decadent.org.uk/pipermail/cpp-threads/2008-December/001933.html`, Dec. 2008.

[28] E. Vallejo, R. Beivide, A. Cristal, T. Harris, F. Vallejo, O. Unsal, and M. Valero. Architectural support for fair reader-writer locking. In *Proc. MICRO*, 2010.