Free Atomics: Hardware Atomic Operations without Fences

Ashkan Asgharzadeh
Computer Engineering Department
University of Murcia
30100 Murcia, Spain
ashkan.asgharzadeh@um.es

Juan M. Cebrian
Computer Engineering Department
University of Murcia
30100 Murcia, Spain
jcebrian@um.es

Arthur Perais
Univ. Grenoble Alpes, CNRS,
Grenoble INP*, TIMA
38000 Grenoble, France
arthur.perais@univ-grenoble-alpes.fr

Stefanos Kaxiras
Dept. of Information Technology
Uppsala University
Uppsala, Sweden
stefanos.kaxiras@it.uu.se

Alberto Ros
Computer Engineering Department
University of Murcia
30100 Murcia, Spain
aros@ditec.um.es

ABSTRACT

Atomic Read-Modify-Write (RMW) instructions are primitive synchronization operations implemented in hardware that provide the building blocks for higher-abstraction synchronization mechanisms to programmers. According to publicly available documentation, current x86 implementations serialize atomic RMW operations, i.e., the store buffer is drained before issuing atomic RMWs and subsequent memory operations are stalled until the atomic RMW commits. This serialization, carried out by memory fences, incurs a performance cost which is expected to increase with deeper pipelines.

This work proposes *Free atomics*, a lightweight, speculative, deadlock-free implementation of atomic operations that removes the need for memory fences, thus improving performance, while preserving atomicity and consistency. Free atomics is, to the best of our knowledge, the first proposal to enable store-to-load forwarding for atomic RMWs. Free atomics only requires simple modifications and incurs a small area overhead (15 bytes). Our evaluation using gem5-20 shows that, for a 32-core configuration, Free atomics improves performance by 12.5%, on average, for a large range of parallel workloads and 25.2%, on average, for atomic-intensive parallel workloads over a fenced atomic RMW implementation.

CCS CONCEPTS

• **Computer systems organization → Multicore architectures**;
• **Theory of computation → Parallel computing models**.

KEYWORDS

Multi-core architectures, microarchitecture, atomic Read-Modify-Write instructions, Total-Store-Order (TSO), store-to-load forwarding

*Institute of Engineering Univ. Grenoble Alpes

ISCA '22, June 18–22, 2022, New York, NY, USA

ACM ISBN 978-1-4503-8610-4/22/06…$15.00
https://doi.org/10.1145/3470496.3527385

ACM Reference Format:
Ashkan Asgharzadeh, Juan M. Cebrian, Arthur Perais, Stefanos Kaxiras, and Alberto Ros. 2022. Free Atomics: Hardware Atomic Operations without Fences. In *The 49th Annual International Symposium on Computer Architecture (ISCA '22), June 18–22, 2022, New York, NY, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3470496.3527385

1 INTRODUCTION

Atomic Read-Modify-Write (RMW) instructions are primitive synchronization operations provided by most instruction set architectures (ISA), such as x86 [26], IBM Power [25], ARMv8 [4] (since v8.1), and RISC-V [55]. Atomic RMWs are used either directly by programmers or by operating-system libraries to provide higher-abstraction synchronization mechanisms to programmers. These synchronization mechanisms include mutually exclusive locks, barriers, and other mechanisms used to negotiate mutual exclusion among threads in parallel applications [23].

According to public documentation, current x86 processors implement atomic RMWs by (1) acquiring exclusive permission for the corresponding cacheline, (2) *locking* that cacheline (*cache locking* in Intel terminology) thus preventing other cores from accessing the same cacheline by denying their coherence requests, (3) reading-modifying-writing a new value into the cacheline, and (4) *unlocking* the cacheline [26].

Still according to documentation, those atomic RMWs serialize all outstanding load and store operations, i.e., wait for them to commit [26]. This means that the store buffer (SB) is drained before an atomic RMW issues [39], and that subsequent memory operations cannot perform until the atomic RMW writes and releases the cacheline lock. This serialization, easily implemented by surrounding the atomic RMWs with memory fences [41], degrades performance.

Figure 1 quantifies that cost, split into cycles waiting to drain the SB (*Drain_SB*) and cycles waiting for the atomic RMW to commit (*Atomic*) as measured in our simulation infrastructure using gem5-20 [36] and running Splash-3 [47], Parsec-3 [8], and a modern suite of write-intensive benchmarks [20, 30] (see Section 5.1 for detailed information). The average cost of atomic RMWs, dominated by *Drain_SB* cycles, is generally more than 100 cycles for Skylake (224-entry reorder buffer –ROB–) and increases for Icelake (352-entry ROB). In some applications, the cost per atomic RMW can be much higher (e.g., almost 700 cycles for fft and radix and

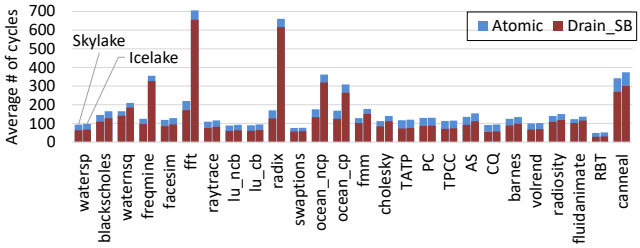


Figure 1: Cost in terms of performance of atomic RMWs

almost 400 cycles for ocean_ncp and canneal). A previous study [41] reported a cost of 67 cycles for a Sandy Bridge machine (168-entry ROB). This is consistent with our observation that the latency of a fenced implementation of atomic RMWs increases with the ROB size. Despite their importance [5], little effort has been dedicated in the literature to optimizing atomic RMWs.

A key observation of this work is that the main role of fences surrounding atomic RMWs as defined by the x86 architecture is *not* to enforce order between memory operations, but to disable any memory-related speculation mechanism by running the atomic RMWs in isolation. The reason is that the total store order model (TSO), supported by x86 (x86-TSO) [49], already enforces ordering across atomic RMWs, since both load→load and store→store orders are enforced and atomic RMWs guarantee the atomic execution of the load and store. Hence, no loads or stores would appear to be reordered across atomic operations, even in the absence of fences (see Section 3.2.3 for details).

This work explores the feasibility of executing atomic RMW instructions concurrently and out-of-order, that is, ignoring the memory fences introduced in the ISA specification, while enforcing x86-TSO semantics. To this end, we propose *Free atomics*, atomic RMWs without memory fences. On the one hand, Free atomics lock their target cacheline, and are therefore never cancelled due to incoming invalidations or other external events, thus preventing starvation scenarios. On the other hand, Free atomics execute speculatively, avoiding unnecessary costs such as draining the SB before issuing subsequent instructions, while still enforcing the memory consistency model and providing atomicity guarantees (Section 3.4). However, unfencing atomics while performing cache locking introduces deadlock scenarios. We elaborate on deadlocks that Free atomics can face and how to avoid them in Section 3.2.

This work focuses on the x86-TSO memory model. Indeed, in weaker memory models, such as the ARMv8 model, the fences applied to atomic RMWs (e.g., *acquire* and *release* fences can be added to *fetch-and-add* as needed in ARMv8.1) actually entail a stronger ordering. Therefore, removing the fences would be a more involved process, because by default, the implementation will not provide the "safety net" that is used to implement stronger models.

The main contributions of this work are:

- A mechanism to efficiently squash atomic RMW instructions that collectively hold multiple cacheline locks.
- An exhaustive description of possible livelock and deadlock scenarios that may appear when executing memory operations (including atomic RMWs) out of order, and simple solutions to recover from such deadlocks.
- Enabling store-to-load forwarding from/to atomic RMWs, while enforcing both atomicity and consistency. This allows Free atomics to be executed after each other without relinquishing the permission of the target cacheline, improving performance by increasing lock locality.

We evaluate Free atomics using the gem5-20 full-system simulator, showing performance improvements of 12.5%, on average, for a large range of parallel workloads and 25.2%, on average, for atomic-intensive parallel workloads, over a fenced atomic RMW implementation for a 32-core system. This is achieved with a simple implementation of just 15 bytes.

2 BACKGROUND

At the ISA level, there are two main alternatives when designing atomic memory operations, which are independent of the memory consistency model enforced by the system: atomic RMW instructions and load linked/store conditional (LL/SC) pairs. Atomic RMWs are single instructions that offer direct support for atomic operations such as *fetch-and-increment*, *test-and-set*, and *compare-and-swap* [37]. LL/SC [29] are pairs of instructions that can be used to implement, in software, the same atomic operations. The key difference is that since LL and SC are distinct architectural instructions, the primitive (usually LL–op–SC) is interruptible (e.g., a context switch may take place in between), whereas an atomic RMW instruction is not. Moreover, an LL/SC pair will fail due to relevant external events such as coherence invalidations and cache evictions. As a consequence, LL/SC pairs are commonly enclosed in a spin loop that exits when the store conditional succeeds. Conversely, from the programmer's perspective, atomic RMWs always succeed. This gives an advantage to atomic RMW instructions by simplifying the code but more importantly, by avoiding the need for any cleanup at the software level should an LL/SC pair be interrupted before succeeding or failing.

Atomic RMW instructions are supported by architectures such as x86 [26] ("locked atomic operations"), IBM Power [25], ARMv8 [4], and RISC-V [55] ("atomic memory operations" or AMO). A conservative way to implement atomic RMW instructions is to execute them in isolation with other memory operations, i.e., all load instructions must commit and all store instructions must be drained from the SB before the execution of the atomic RMW. Moreover, a cache locking mechanism is triggered once the read-write permission has been acquired for the corresponding cacheline in order to guarantee atomicity [11, 26, 39]. In this case, locking entails preventing any remote request from invalidating or modifying the cacheline.

Figure 2 describes the execution of a *fetch-and-increment* atomic RMW instruction (Fetch&Inc) according to the x86 architecture specification, that is decoded into five micro-operations (μops) [1, 48], explained below:

1. `Mem_Fence1`: This memory fence guarantees that the load μop of an atomic RMW (i.e., load_lock) does not issue until all previous memory operations commit and exit the SB. This has a high performance cost (see *Drain_SB* in Figure 1). Furthermore, this fence protects atomic RMWs against squashes,

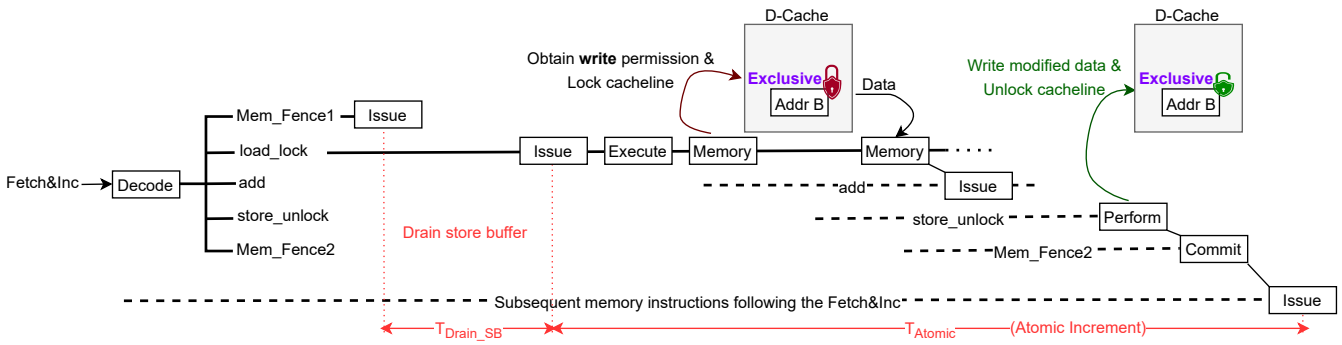


Figure 2: Implementation and execution of atomic RMWs. Instructions and µops follow the gem5-20 naming convention.

for example, due to mispeculation caused by a previous memory instruction. Finally, it also protects against deadlock scenarios that may arise when the cacheline lock is taken before emptying the SB, as first described by Rajaram et al. [41].

2. `Load_Lock`: This µop is an ordinary load for which some memory flags have been set to indicate its responsibilities and specific features [1, 36]. First, it *cannot execute speculatively*, that is, it is not issued until all previous instructions have committed. This guarantees that the load_lock cannot be squashed. Second, it *acquires read-write coherence permission* for the target cacheline (not just read permission), as it is always followed by a store operation [37]. Third, it *locks the cacheline*, preventing both local and external requests from accessing the cacheline.
3. `Add`: This is the arithmetic or logical µop (depends on the atomic RMW, e.g. atomic_fetch_add [27, 31]). Due to data dependencies, it cannot issue until the load_lock performs, i.e., reads the value from memory.
4. `Store_Unlock`: When it performs (after committing), this µop writes the data to the target cacheline, unlocks it, and leaves the SB. This makes the cacheline available to other memory requests (either local or external) [39].
5. `Mem_Fence2`: This fence prevents younger loads from issuing until the atomic RMW commits [26, 52]. There is a performance penalty associated with the wait time of subsequent loads (see *Atomic* in Figure 1).

3 FREE ATOMICS

Free atomics aim to improve the performance of atomic RMWs by removing the fences surrounding them and allowing them to execute partially out-of-order. We iteratively build three flavours of Free atomics: first, we allow out-of-order speculative execution, second, we remove the fences surrounding atomic RMWs, and third, we enable store-to-load forwarding to/from atomic RMW instructions.

3.1 Allowing out-of-order speculative execution

As memory fences only impose order among memory operations, atomic RMW instructions, and specifically load_lock µops, could in fact issue before becoming the oldest instruction in the pipeline, thus potentially executing out-of-order. Doing so, however, implies

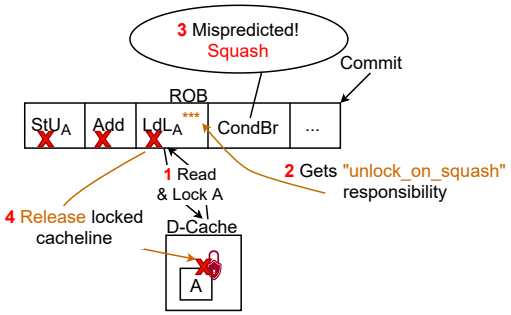


Figure 3: Unlock on squash

that the atomic RMW instruction may now be squashed due to branch mispredictions or exceptions.

When an instruction is squashed, the micro-architectural state updates performed by the squashed instruction have to be canceled [26]. The µops comprising an atomic RMW instruction are similar to other µops in this aspect. Squashing them is straightforward, except for the cache locking mechanism. With speculative execution of atomic RMW instructions, it is possible that the load_lock has (speculatively) locked the target cacheline prior to being squashed. At this time, the store_unlock cannot have performed. Therefore, we must provide a mechanism to unlock the cacheline if the atomic RMW instruction is squashed, otherwise, the cacheline would remain locked and inaccessible to other memory requests forever, and the system would eventually deadlock. To address this, each load_lock that successfully locked its target cacheline is assigned the *unlock_on_squash* responsibility. When the load_lock squashes, it unlocks the cacheline by carrying out the same action that would have been carried out by the store_unlock when performing. Figure 3 shows the squash of an atomic RMW operation to address *A*. The load_lock (*LdL*) that is responsible for unlocking its target cacheline is squashed when an older conditional branch (*CondBr*) is found to have been mispredicted. Then, the lock is lifted and all subsequent µops, including the store_unlock (*StU*), are squashed.

As we show in Section 5, issuing atomics (with fences) out-of-order from control-speculative paths does not provide significant performance gains. This is because branches tend to resolve fast

if they do not depend on loads, or if the load they depend on has already obtained data. Therefore, issuing a fenced atomic RMW speculatively offers little performance advantage. However, the ability to issue an atomic RMW speculatively and out-of-order paves the road for our next optimization: removing the fences.

3.2 Unfencing

Fences surrounding atomic RMWs do not aim to enforce x86-TSO consistency, since the load_lock and the store_unlock are performed atomically, which means that the load_lock cannot execute ahead of previous stores. The fences instead disable speculative re-ordering mechanisms for memory operations, such as speculative load→load reordering [19]. This section demonstrates that such fences can be completely removed and resolves the associated challenge: preventing livelocks and deadlocks.

3.2.1 Concurrency and correctness. Unfencing gives the opportunity to execute local memory requests concurrently with atomic RMWs. Some of these requests may target cachelines that are already locked. Preserving atomicity does not require to lock the cacheline against local requests, i.e., coming from the core that locked the cacheline. Hence, Free atomics enable local accesses to locked cachelines, which is not possible when using *fenced* atomic RMWs.

Local loads older than a Free atomic can freely read from a locked cacheline without jeopardizing atomicity, since they will always read the value before the RMW operation (stores perform after the loads). This is a safe behaviour since in an in-order execution the cacheline would not be locked at the time the load performs. In case the atomic is squashed, the loaded data remains correct data and the older load is unaffected.

Similarly, local loads younger than a Free atomic can safely read from a locked cacheline. If the load accesses the same (or overlapping) bytes as the Free atomic, it should get the data through conventional store-to-load forwarding from the previous store_unlock if the data is ready. Otherwise, if obtaining the data from memory, it will be squashed when the store_unlock resolves its address as a consequence of a memory dependence misprediction. If the load accesses different bytes than the Free atomic, reading the data does not compromise atomicity.

Local stores older than a Free atomic can also write into a locked cacheline. In this case, it is guaranteed that the store targets different bytes than the Free atomic. Otherwise, a memory dependence misprediction would have been detected when the store calculated its target address, thus squashing the Free atomic. If the memory dependence prediction is correct or the store address is known at the time the Free atomic performs, the Free atomic load_lock is re-scheduled,[1] preventing it from executing when a previous store to the same address is in flight. This condition is relaxed in Section 3.3.

Local stores younger than the Free atomic will always perform the write after the Free atomic, in order.

[1] A load_lock (or a normal load) is re-scheduled when it is not able to perform the first time (e.g., cache miss). Re-scheduling assigns the responsibility of returning data (and locking the cacheline for load_lock) to the load queue entry, through sending the instruction to the memory pipeline a second time. We handle two consecutive atomic RMWs to the same bytes in the same way.

3.2.2 Allowing concurrent execution of Free atomics. Free atomics can speculatively execute out-of-order. It is therefore possible that while a preceding Free atomic is outstanding (e.g., waiting to obtain read-write coherence permission for the target cacheline, or waiting for the store_unlock to perform), a younger Free atomic executes and attempts to lock a cacheline. We enumerate three possible scenarios, and highlight their implications:

- Multiple Free atomics target different cachelines. *Implication 1:* The cache locking mechanism must support multiple cachelines.
- Multiple Free atomics access the same cacheline, but different bytes: the cacheline is locked more than once. *Implication 2:* Precise information about the number of Free atomics locking a cacheline is required.
- Multiple Free atomics access the same (or overlapping) bytes, (which implies that the cacheline is locked more than once), but the cached data is stale from the point of view of the younger load_lock. *Implication 3:* Younger atomic RMWs must read the data stored by older atomic RMWs if their addresses overlap.

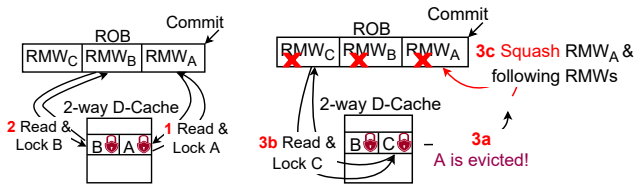
Section 4 presents a microarchitecture that implements Free atomics by building on those implications.

3.2.3 Enforcing store→AtomicRMW→load order. In TSO memory model, load→load, store→store, and load→store orders are preserved. Therefore, these orders are also enforced in the presence of atomic RMWs. Furthermore, atomic RMWs perform the load (i.e., load_lock) and store (i.e., store_unlock) atomically. As a consequence of these two implications, TSO should preserve store→AtomicRMW→load order. However, the omitted store→load order in TSO might jeopardize the consistency definitions regarding atomic RMWs when we blindly remove memory fences from the micro-operations of atomic RMWs.

To guarantee store→load order *across* atomic RMWs, Free atomics should enforce two sub-orders: a) store→AtomicRMW, and b) AtomicRMW→load. The key aspects that guarantee those sub-orders are that (i) Free atomics commit after draining older stores from SB, so when the store_unlock enters the SB, it finds the SB empty, and (ii) the target cacheline of a Free atomic is already locked by the load_lock and the required write permission has been obtained.

Store→AtomicRMW order is enforced by committing a Free atomic only when the SB is empty. Yet, load_lock can speculatively perform before an older store (that resides in SQ or SB) performs. Since load_lock ensures that no remote core writes on its cacheline from the time it performs, it is guaranteed that the value in memory at the time of commit is the same as when performed, so the order with previous stores is enforced.

For the AtomicRMW→load order it is important to note that the store_unlock can first commit and then stay in the SB until it writes to cache (which takes just the write latency as SB is empty). However, during this time the target cacheline remains locked, so no remote load or store can access it until the store_unlock writes in cache and unlocks the cacheline. Moreover, any remote write to the cacheline accessed by a load following the FreeAtomic, while the Free atomic is not committed, would squash the load [19]. AtomicRMW→load is hence enforced.



(a) All ways in a set locked (b) Squash due to eviction of locked cacheline

Figure 4: Livelock due to eviction of a locked cacheline

Yet, while the store_unlock waiting in SB to write in cache, the speculative loads can commit after FreeAtomics commit (i.e, in-order commit). However, these speculative loads can never be reordered over older stores since they left the SB before the load can commit, and the only store in the SB is the store_unblock. Consequently, store→load order across Free atomics is preserved as well.

3.2.4 Handling Livelocks by Preventing Evictions of Locked Cachelines. When executing memory requests concurrently with atomic RMWs, cache evictions can take place while locks are held. Selecting a locked cacheline as a victim is tantamount to lifting the lock and therefore squashing the atomic RMW. This can be a source of livelock, as the eviction may happen indefinitely. Figure 4 depicts one example of such livelock scenario. Two atomic RMWs lock their respective cachelines, A and B, which map to the same cache set (1&2). When a third atomic RMW aims to lock a cacheline C mapping to the same set (3b), A is selected for eviction first (3a). The eviction causes the atomic RMW, and consequently all subsequent instructions to be squashed (3c). This situation may repeat over and over, thus creating a livelock scenario. To enforce liveness, we guarantee that a locked cacheline is never selected as the victim by the replacement policy. This restriction introduces new deadlock scenarios in the presence of out-of-order RMWs execution, i.e., if all ways of a cache set are locked and an older instruction (atomic RMW or regular memory operation) needs to allocate a new cacheline in the cache to retire. Those are handled through the deadlock avoidance mechanism discussed next.

3.2.5 Squashing Atomic RMWs to Avoid Deadlocks. When executing atomic RMWs concurrently with other memory instructions, a number of deadlock scenarios may appear. Rajaram et al. [41] reports a deadlock scenario when atomic RMWs are executed before draining the SB. We extend the deadlock analysis by showing new deadlock scenarios introduced in Free atomics. We then present a general solution that does not require changes in the coherence protocol and that guarantees forward progress.

Deadlock analysis. We start our description with the arguably most obvious deadlock, which is caused by re-ordering atomic RMWs, similar to the problem of two threads trying to acquire two software locks in the opposite order [12]. Figure 5 depicts a *RMW-RMW deadlock scenario*, where younger load_locks execute speculatively locking a cacheline, earlier than an older (in program order) load_lock that still needs to acquire the lock. Core1 atomically updates A and then B, while core2 atomically updates B and then A.

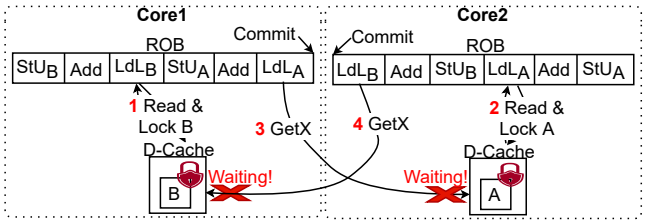


Figure 5: RWM-RMW deadlock scenario

Core1 executes first LdL_B and acquires the cache lock successfully (1). Core2 executes first the LdL_A and acquires the lock, too (2). The deadlock happens because the LdL of the older atomics cannot be performed as the cacheline lock is held in another core, and cannot commit; the StU of the younger atomic cannot commit and hence be performed (stores perform after committing); and the cache locks are never lifted (3&4).

We continue our analysis with the deadlock scenario reported by Rajaram et al. [41], which may happen when atomic RMWs acquire cacheline locks before the SB drains. Figure 6 depicts the *Store-RMW deadlock scenario*. First, LdL_B in core1 and LdL_A in core2 execute speculatively and lock two different cachelines (1&2). Then, ordinary stores in the core's respective SB try to obtain write permission for the cachelines, which are already locked in the other core (3&4). The GetX (get exclusive) coherence requests generated by the stores can only be granted when the requested cachelines are unlocked. The cachelines A and B will be unlocked when the respective StUs perform and write their data. However, since there is a stalled store at the head of each SB, both atomic RMWs in core1 and core2 cannot commit and therefore their StUs cannot perform (perform happens after commit) and unlock their respective cachelines. Eventually, the system will deadlock. Stores should ensure forward progress in order to avoid the deadlock.

Deadlock scenarios can also appear when atomic RMWs execute before previous ordinary loads, as there may be no guarantee for previous loads to perform and eventually commit. The only reason a load cannot perform is when it finds a cacheline locked in another core. Figure 7 depicts a *load-RMW deadlock scenario*. Core1 has locked cacheline B (1) and Core2 has locked cacheline A (2) as a consequence of two atomic RMW instructions. The loads to address A in Core1 (Ld_A) and to address B in Core2 (Ld_B) try to access the cacheline which has been already locked in a remote core (3&4). Therefore, the loads cannot perform and the subsequent atomic RMWs cannot commit, so the cachelines remain locked forever. Note that any combination of the previous scenarios, e.g., a Load-RMW reordering in one core and a Store-RMW reordering in another can similarly lead to a deadlock situation.

Finally, we discuss a deadlock situation caused by *cache inclusion* properties, similar to the scenario discussed by MAD atomics [21]. The concept of cacheline locking is only considered for the core's private data cache. Indeed, locking a cacheline in the whole hierarchy would incur significant latency and notably slow down the common case (uncontended cacheline) by tens of cycles. As a result, higher level caches such as the L2, the L3 and the directory (or snoop filter) remain unaware of the fact that a cacheline is locked in a private L1D. If any of those higher levels enforces inclusion

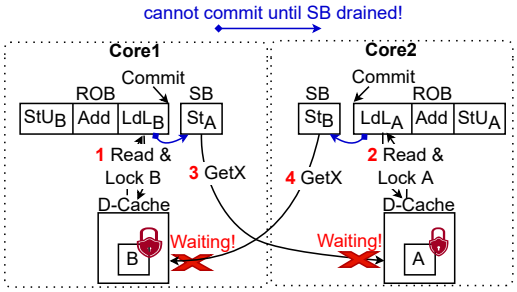


Figure 6: Store-RMW deadlock scenario

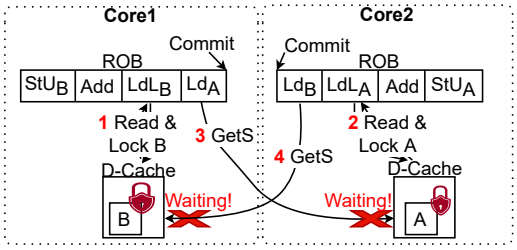


Figure 7: Load-RMW deadlock scenario

–which is always the case of the directory– then a deadlock can arise. Indeed, assume core0 has locked an L1D cacheline by speculatively performing a load_lock (*LdL*) operation. Also assume there exists an older unperformed store (*St*) in the SB of core0: the store is committed but waiting for write permission. The deadlock arises if, at the directory, allocating an entry for *St* leads to selecting the entry relating to the locked cacheline as the victim. Because the directory is inclusive, to evict the victim, all privately cached copies have to be invalidated, including the copy that is currently being locked in core0. Since locked cachelines cannot be evicted until the atomic RMW is performed (Section 3.2.4), core0 will block the invalidation request and the directory will not be able to allocate an entry for the new cacheline, which will prevent the atomic RMW from committing (and then unlocking the cacheline) due to the non-empty SB, leading to a deadlock. While core0 could inform the directory that it needs to pick another victim, this does not solve the deadlock as it is possible for all possible directory victims to actually be locked in private caches. Generally speaking, this deadlock can be triggered at any inclusive cache, shared or not.

Solution. We address the depicted deadlocks with a single watchdog mechanism. Specifically, we implement a cycle counter that is reset each time a load_lock performs (i.e., manages to lock a cacheline). If the counter reaches a certain threshold and no atomic RMW has committed, the watchdog triggers a pipeline flush starting from the oldest atomic RMW instruction that is holding a cacheline lock. This mechanism is similar to the one used by DIVA [6] in the context of –transiently– faulty hardware. This pipeline flush unlocks all locked cachelines in the core, letting incoming coherence request(s) and older unperformed memory operations left in the pipeline progress. As Free atomics do not commit until the SB drains, timeouts always find the older speculative Free atomic holding a cacheline lock in the pipeline. We found Free atomics-related deadlocks to be rare even with 32 cores. As a result, a large timeout value (10000 cycles) results in firing just a handful of times in specific applications (see Table 2).

Progress guarantees. Free atomics adhere to the following invariant: *only the core executing a Free atomic can squash it (thus lifting the cache lock); invalidation requests from other cores that find the cacheline locked have to wait until it is unlocked.* That is, the decision of squashing a Free atomic always comes from within the core executing the Free atomic. By squashing a Free atomic, a remote request waiting for the newly unlocked cacheline will progress. In other words, if a Free atomic is squashed and re-executed, it cannot re-enter a deadlock scenario with the *same* memory instruction that caused the initial deadlock that led to the squash.

3.3 Enabling store-to-load forwarding

Modern processors that execute loads out-of-order implement store-to-load forwarding, letting loads obtain data from the youngest previous store to the same address. This is a way to safeguard sequential semantics when previous stores have not written to cache yet that does not stall loads as long as the store data is available. Since fenced atomic RMWs execute in isolation, forwarding is not possible.

However, Free atomics execute the load_lock in the presence of previous unperformed stores, opening the possibility of forwarding. Disallowing forwarding, as done in some IBM microarchitectures [17, 24], is a sub-optimal alternative that prevents MLP [46]. To extract maximum performance, Free atomics allow a load_lock to read its data from previous unperformed stores, which may benefit specific software idioms.

3.3.1 Forwarding from a store_unlock. When forwarding from a store_unlock, correct execution is guaranteed by preserving atomicity between the forwarded (younger) Free atomic and the forwarding (older) Free atomic. That is, both Free atomics perform without releasing the cacheline lock. To keep the cacheline lock, the forwarding store_unlock should not unlock its cacheline when performing, but instead delegate that responsibility to the store_unlock of the forwarded Free atomic. Hence, a *do_not_unlock* responsibility is assigned to each store_unlock that forwards data to a load_lock, making the store_unlock perform as an ordinary store. The forwarded load_lock does not need to lock the cacheline as it is already locked. The cacheline is eventually unlocked by the store_unlock of the forwarded Free atomic.

Figure 8 depicts this scenario. First, $LdL1_A$ executes and locks the cacheline A (1). Then, $LdL2_A$ executes and accesses in parallel the cache and the store queue (SQ).[2] As $LdL2_A$ finds a match in the SQ (2b), its request to the cache is discarded (2a). When $StU1_A$ forwards data to $LdL2_A$, $StU1_A$ sets the *do_not_unlock* responsibility (3). When performing, $StU1_A$ just writes the new data to the cacheline, leaving the cacheline locked (4). When $StU2_A$ performs, writes and unlocks the cacheline A (5), making it available for external requests.

[2]The SQ contains all dispatched but not performed stores, while by SB we refer to the part of the SQ that contains the stores that already committed.

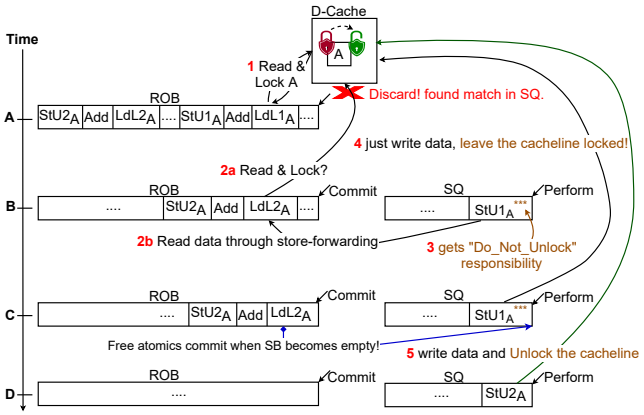


Figure 8: Forwarding from a store_unlock

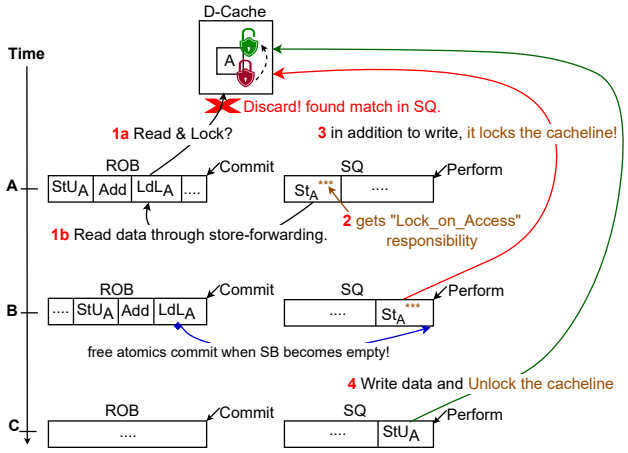


Figure 9: Forwarding from an ordinary store

3.3.2 Forwarding from an ordinary store. The key for correctness when forwarding from ordinary stores lies in the following observation made by Ros and Kaxiras [46]: When forwarding from a store, the load effectively performs when the store writes to cache. Free atomics should guarantee that when a forwarded load_lock performs (i.e., the forwarding store performs) the target cacheline is locked. Hence, a store that forwards data to a load_lock gets a responsibility called *lock_on_access*, which implies that the store must lock the cacheline when it performs, on behalf of the load_lock.

Figure 9 depicts the described scenario. First, LdL_A executes and accesses the SQ and the cache. The cache request is discarded (1a) on a SQ match (1b). When the store St_A forwards data to LdL_A, it gets *lock_on_access* responsibility (2). When performing, St_A locks the cacheline (3) and writes the data. Eventually, when StU_A performs, unlocks the cacheline (4).

3.3.3 Squashing a forwarded Free atomic. When a load_lock is squashed, it must release its target cacheline lock (Section 3.1). However, when a load_lock obtains its data through forwarding, the forwarding store (either a store_unlock or an ordinary store) becomes responsible for keeping the cacheline locked on behalf of the forwarded load_lock. Therefore, when squashing a forwarded load_lock, if the forwarding store still resides in SQ, the responsibility assigned to it (either *do_not_unlock* or *lock_on_access*) should be taken back. Otherwise, the load_lock can freely perform the *unlock_on_squash*, as the cacheline is guaranteed to be locked.

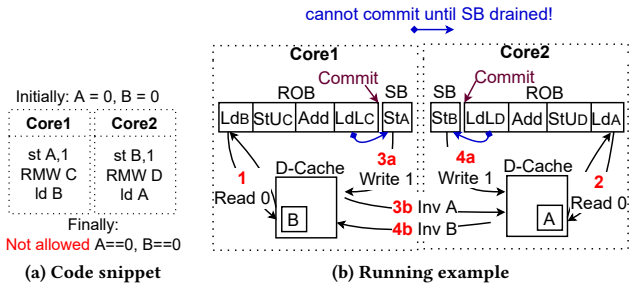


Figure 10: Proof of type-1 atomicity using Dekker's algorithm

3.3.4 Chain of Forwarding. A forwarded Free atomic could also forward the data to a subsequent Free atomic, thus creating a forwarding chain of arbitrary length. This forwarding chain improves lock locality by preventing other threads from *stealing* the cacheline and allowing a number of Free atomics to perform without releasing the cacheline lock. This, however, could lead to livelock if the chain length is not controlled. Therefore, Free atomics only permit a maximum number of consecutive forwarding (32 in our evaluation).

3.4 Free atomics are Type-1 atomics

Rajaram et al. [41] define three types of atomicity for RMWs based on the guarantees they provide. Type-1 is the strictest of the three types and it is equivalent to *fenced* atomic RMWs. Specifically, type-1 atomics *prevent writes of any address from appearing between the read and the write in the global memory order*. We claim that Free atomics adhere to the type-1 specification. A load_lock can perform while older stores have not performed yet. However, the execution is equivalent as executing the load_lock after all previous stores perform since i) reading the value implies that it cannot be modified by any write, as remote writes see the cacheline locked and local writes to the same memory location as the load_lock would squash the load_lock due to a memory-dependence violation and ii) the load_lock does not commit until the SB drains (see Section 3.2.3).

Let us consider the example provided by Rajaram et al. –Dekker's algorithm– using atomic RMWs as barriers, shown in Figure 10a (Figure 1 in Rajaram et al. [41]). In the example, atomic RMWs should prevent the loads from being reordered across the stores. Free atomics enforce that behaviour by not committing before the SB drains. Figure 10b shows a running example where both loads have executed and read an old value (1&2). Both stores are still in the respective SBs, preventing the Free atomics from committing. A write to either address A or B (3a&4a) would cause an invalidation in the other core cache (3b&4b) and consequently squash the corresponding load, therefore enforcing store→load order *across* Free atomics.

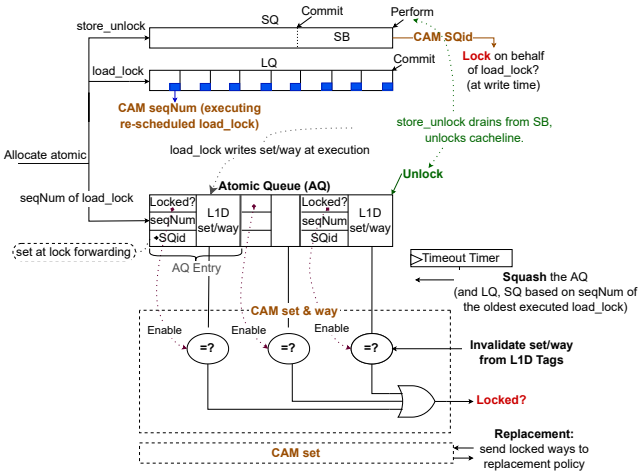


Figure 11: Proposed microarchitectural changes

4 HARDWARE IMPLEMENTATION

In a baseline implementation of atomic RMWs [26], a single cacheline can be locked at any given time. However, in Free atomics, multiple atomic RMWs may speculatively execute and lock multiple cachelines, or even the same cacheline multiple times. A hardware implementation should therefore provide i) a way to determine if a given cacheline is currently locked and ii) an efficient support for managing the responsibilities granted and revoked due to forwarding and squashes.

4.1 Determining locked cachelines

To determine the currently locked cachelines, we introduce the Atomic Queue (AQ), which is able to track multiple cachelines (*Implication 1* in Section 3.2.2). This queue is managed as a FIFO, and can conceptually be seen as a subset of the SQ. Each entry of the AQ relates to one Free atomic, that is, the load_lock and store_unlock share the same AQ entry. An entry is allocated in the AQ when a load_lock is dispatched to the LQ and the ROB, and deallocated when the corresponding store_unlock performs its write and leaves the SQ. A lack of free entry in the AQ when an atomic RMW has to dispatch stalls the front-end (much like when the SQ is full and a store needs to dispatch). Figure 11 depicts the AQ along with the LQ and the SQ. Each AQ entry consists of i) a *Locked* bit, ii) the L1D set/way where the locked cacheline resides, iii) a sequence number to handle flushes and load_lock re-scheduling, and iv) a pointer to the SQ (*SQid*) to handle forwarding.

4.1.1 Locking. When a load_lock obtains exclusive permissions and performs it locks the cacheline by writing the L1D set and way where the cacheline resides in its AQ entry and setting the *Locked* bit.[3] A cacheline locked multiple times will simply have its set and way information present in multiple AQ entries (*Implication 2* in Section 3.2.2).

[3]The *Locked* bit expresses that the set/way fields are valid. However, if the number of ways is not a power of two, a specific way encoding can be used to express set/way invalidity rather than implementing a dedicated *Locked* bit.

4.1.2 Unlocking. When a store_unlock at the head of the SQ performs, a signal is sent to the AQ to commit its head entry, which corresponds to the Free atomic of the performing store_unlock. This gracefully removes the lock imposed on the cacheline by this particular Free atomic. Note that the cacheline may remain locked if a younger load_lock targeting the same cacheline has already performed. This process is shown by the `Perform` (on the SQ) and `Unlock` (on the AQ) arrows on the right portion of Figure 11.

4.1.3 Searching. The AQ is searched associatively to determine the locked cachelines in the following two cases:

- External request (AQ searched by set/way): When a remote request performs its cache access and determines a hit, the AQ is associatively searched (unless it is empty). Only AQ entries with the *Locked* bit set participate in the search. If at least one set/way match is found in the AQ, the cacheline is already locked and the remote request must be blocked or retried. This search is performed via the first AQ `set & way` CAM port (bottom part of Figure 11.
- Cache replacement (AQ searched by set): To prevent livelocks, we must not replace a locked cacheline in the L1D. As a result, the AQ participates in the replacement policy by being searched using the L1D set as input and providing all ways stored in entries that have *Locked* bit set and whose set matches the set suffering the eviction. This search is performed via the AQ `set` CAM port shown at the bottom of Figure 11. A suitable victim may not always exist since all ways of the set can be locked, potentially leading to a deadlock. Since in our implementation, the number of entries of the AQ (4) is smaller than the associativity of the L1D (12), this situation is not possible. If the AQ size is strictly greater than the L1D associativity, a deadlock may arise as all ways can be locked by younger load_locks, but the timeout mechanism will eventually trigger and break this deadlock. The same applies if the AQ size is same as the L1D associativity and all ways are locked: If an older regular instruction needs to allocate in the L1D to retire, it will not be able to do so, and the timeout mechanism will also trigger.

Performing an associative search facilitates handling pipeline flushes compared to keeping track of locked cachelines through dedicated counters or L1D tag metadata: a squashed Free atomic is flushed from the AQ (the same way any instruction is flushed from ROB, LQ, SQ) and will therefore not participate in the search anymore, automatically unlocking the cacheline. Flushing an AQ entry only requires resetting the *Locked* bit. The sequence number is used to determine the AQ flush point and to let re-scheduled load_locks determine which AQ entry they relate to.

4.2 Managing store-to-load forwarding

We provide a generic implementation that supports both forwarding from a previous store and from a previous store_unlock, which covers *Implication 3* in Section 3.2.2. Specifically, the *SQid* field is populated when the corresponding load_lock forwarded from either an ordinary store or a store_unlock. In this case, any set/way information that it obtained from the cache if a hit took place is ignored, and the *Locked* bit is untouched. When a store leaves the SQ and writes to L1D, it broadcasts its SQid to the AQ, along with

L1D set/way information. Any AQ entry that matches on the SQid will capture the set/way information and the *Locked* bit, thereby locking the cacheline.[4] This mechanism, which relies on another associative search port (on SQid), is shown at the exit of the SQ in Figure 11 (perform time). The forwarding cases are detailed next:

- A regular store is performing, and it forwarded to a load_lock in the AQ: that store has the *lock_on_access* responsibility (Section 3.3.2). Having the AQ capture L1D set/way information based on the SQid will lock the cacheline, implementing *lock_on_access* for the store.
- A store_unlock is performing, and it forwarded to a load_lock in the AQ: the store_unlock retiring from the SQ is also retiring from the AQ, meaning that it will release its lock on the cacheline. However, since it forwarded to another Free atomic, the lock should not be released. This is achieved by having the AQ entry of the younger Free atomic capture the set/way information based on the SQid broadcast by the SQ. This is tantamount to having the older store_unlock perform both *unlock* and *lock_on_access* responsibility (in this order), which is equivalent to the store_unlock performing the *do_not_unlock* responsibility (Section 3.3.1).

4.3 Storage Overhead and Complexity

The area requirements of Free atomics are minimal compared to other structures in the core. Each AQ entry requires a total of 29 bits for an Icelake-like design: locked bit (1 bit), L1D set/way locator (6+4 bits, for a 48K 12-way L1D), sequence number (9+2 bits for a ROB below 512 entries with wrap around), and *SQid* (7 bits for a 72-entry SQ). According to our sensitivity analysis over AQ size, 4 entries is enough to provide the required concurrency for atomic RMWs in the analyzed benchmarks, amounting to just 116 bits (15 bytes) in total. The AQ has four CAM ports to implement the required functionalities: one for invalidation requests (10-bit comparators for set/way), one for replacement policy (6-bit comparators for set), one to handle locking in the presence of forwarding (7-bit comparators for SQid), and finally, for flushing and re-scheduling (11-bit comparators for seqNum). This amounts to a total of 136 bit comparisons. A 14-bit register to store the current timeout cycle count is also needed.

5 EVALUATION

5.1 Methodology

We simulate a multicore processor consisting of 32 out-of-order cores using the gem5-20 full-system simulator [36]. The simulated system runs Ubuntu 16.04 with Linux kernel 4.9.4. The processor parameters, mimicking an Intel Icelake processor, are shown in Table 1. We use Ruby and SLICC to model the memory hierarchy with a three-level MESI coherence protocol. The crossbar interconnect is modeled with GARNET [3]. Execution latencies are modeled as measured on real hardware by Fog [16]. We integrated a modified McPAT [32, 56] into gem5 to measure energy consumption using a process technology of 22nm, a voltage of 0.6V and the default clock gating scheme for the core. We do not measure energy for uncore (memory controller, network, etc).

We run parallel applications from the SPLASH-3 [47] (*simmedium* inputs), PARSEC 3.0 [9] (*simmedium* inputs) and a suite of write-intensive benchmarks [20, 30]. We pinned the threads to the cores to prevent the scheduler from generating an unbalanced load. We omit applications from PARSEC that did not finish execution on the baseline gem5 simulator when running with 32 cores. In addition we run Volrend from SPLASH-3 with *simsmall* input for the same aforementioned reason. We report statistics for the region of interest (ROI), that is, code after initialization and before output. We account for variability by running applications ten times. Each run is preceded by a randomized sleep timer to alter the architectural state. We then remove the three slowest runs (outliers) and compute the average of the other seven.

[4] A single AQ entry will generally match, but due to out-of-order execution, multiple entries may –incorrectly– match. If so, a memory dependency misprediction occurs and resolving it will gracefully restore the AQ state.

Table 1: System Configuration

Processor	
Width	Fetch/Decode 5 instr. Issue/Commit 10 μops
ROB, LQ, SQ	352, 128 entries, 72 entries
Predictors	StoreSet [10], L-TAGE [50]
Processor prefetch	At-commit store prefetch [54]
Memory	
Private L1I	32KB, 8 ways, 1 hit cycle
Private L1D	48KB, 12 ways, 4 hit cycles, pipelined, stride prefetcher [7]
Private L2 cache	256KB, 8 ways, 4 cycles tags, 10 cycles data
Shared L3 cache	16MB, 16 ways, 5 cycles tags, 45 cycles data
Directory	400% coverage, 16 ways
Memory	80ns access time

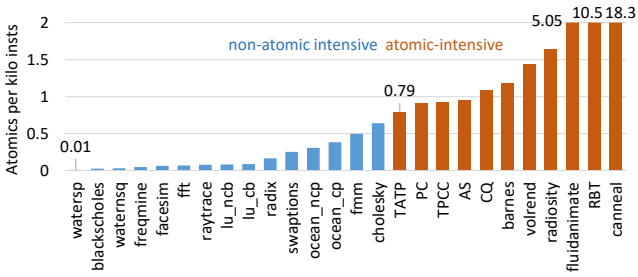


Figure 12: Frequency of atomic RMWs per kilo-instruction

5.2 Frequency of Atomic RMWs

First, we analyze the frequency of atomic RMWs in the evaluated applications. Figure 12 shows the number of committed atomic RMWs per kilo-instruction (APKI). The more an application uses atomic RMWs, the more opportunity is given to Free atomics to improve performance. Hence, in order to highlight the performance improvement achieved by Free atomics, we define as atomic-intensive applications those that show at least 0.75 APKI. This includes 11 applications:three from SPLASH (*radiosity*, *volrend*, and *barnes*); two from PARSEC (*canneal* and *fluidanimate*), and all write-intensive benchmarks. *Canneal* synchronizes purely with atomic operations.

Table 2: Characterization of Free atomics. MDV: Memory Dependency Violations as % of total squashes. FbA: Forwarded by Atomic and FbS: Forwarded by Store as % of total atomics

Benchmark	Omitted Fences (%)	Time-outs	MDV (% squashes)	FbA (% atomics)	FbS (% atomics)
watersp	96.73	0	4.18	2.66	0.11
blackscholes	95.98	0	2.37	3.88	0.28
waternsq	99.63	2	3.04	25.15	0.01
freqmine	99.11	1	2.30	1.74	6.62
facesim	96.94	10	8.15	6.55	1.81
fft	98.50	1	3.32	0.70	9.85
raytrace	97.39	0	2.92	20.98	0.03
lu_ncb	95.74	0	5.70	4.73	0.03
lu_cb	95.70	9	6.91	3.56	0.03
radix	99.53	4	2.55	1.59	10.66
swaptions	99.91	0	1.31	0.08	0.02
ocean_ncp	97.24	21	2.32	3.72	2.38
ocean_cp	97.05	19	2.63	3.17	3.11
fmm	99.15	3	1.62	28.19	0.67
cholesky	96.98	6	1.98	8.78	0.86
TATP	95.34	0	0.15	12.36	0.001
PC	96.27	1	0.27	21.79	0.001
TPCC	96.65	0	0.42	17.01	0.0005
AS	96.35	0	0.45	16.91	0.003
CQ	95.84	3	0.47	9.19	0.26
barnes	97.50	2	0.40	31.16	0.01
volrend	96.42	5	1.56	8.85	0.02
radiosity	97.94	1	1.09	34.01	0.001
fluidanimate	99.98	1	0.69	37.37	0.0005
RBT	99.11	0	0.23	0.14	0.002
canneal	99.98	1	0.04	2.77	0.001
Average	**97.58**	**3.46**	**2.19**	**11.81**	**1.41**

Fluidanimate has millions of non-contended locks. *Radiosity*, *volrend*, *barnes* and the write-intensive benchmarks utilize a considerable amount of locks and barriers for synchronization.

5.3 Characterization of Free Atomics

Table 2 characterizes Free atomics. The first column shows the evaluated applications. The second column, shows that Free atomics are able to omit virtually all the fences (97.58%, on average), as the only fences in x86, disregarding the ones using for atomic RMWs, are placed to ensure store→load order. The third column shows timeout counts, that trigger in 17 out of 26 applications. We employed a large timeout value to avoid *unnecessary* squashes due to long-latency requests that eventually succeed in locking a contended cacheline. Hence, the number of timeouts is extremely low, peaking in *ocean_cp* and *ocean_ncp*. The fourth column shows that only 2.19% of squashed Free atomics have violated a memory dependency. In other words, the main reasons of squashing Free atomics is due to branch misprediction. The fifth column presents the percentage of Free atomics which resolved by store-to-load forwarding from a preceding store_unlock. In some applications like *barnes*, *radiosity*, and *fluidanimate* this number exceeds 30%. For instance, in *barnes* the recursive function "walksub" is by far the most store-to-load forwarding intensive function, as this function uses a pair of "pthread_mutex_lock" and "pthread_mutex_unlock" for interactions between different nodes. Interestingly, store-forwarding to Free atomics is generally carried out by a preceding Free atomic. There are few cases (1.41%) where store-forwarding comes from an ordinary store (sixth column).

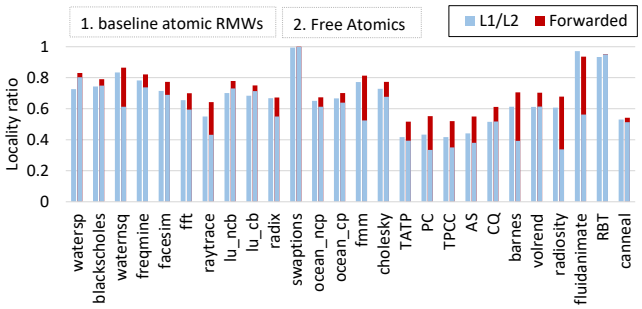


Figure 13: Locality of atomics

5.4 Locality of Atomics

Figure 13 compares the locality ratio of Free atomics against baseline atomic RMWs. Locality in this context relates to how many times a load_lock finds the data in the SQ or with write permission in the L1/L2. Free atomics increase *hardware* lock locality for all of the evaluated applications, except *fluidanimate*. Furthermore, Free atomics provide additional locality through store-to-load forwarding, while baseline atomic RMWs, on the other hand, rely merely on local cache(s) to obtain local lock. In applications, like *radiosity*, *barnes*, *fmm*, *PC*, and *AS* most of the lock locality is provided by enabling store-to-load forwarding for atomic RMWs. Consequently, the lock acquisition latency in Free atomics will be lower than in the baseline atomic RMWs case. In addition to unfencing, enhancing lock locality inevitably translates to a further reduction of the performance overhead of atomic RMWs, as shown next.

5.5 Performance Improvement

Figure 14 shows the execution time of Free atomics normalized to baseline atomic RMWs. Each bar represents the inclusion of additional features over the previous design, starting with out-of-order speculative execution (baseline+Spec), followed by removing fences surrounding atomic RMWs (FreeAtomics), and finally enabling store-to-load forwarding to/from Free atomics (FreeAtomics+Fwd). The shaded fraction of the bar represents active CPU time of the slowest thread, while unshaded represents its quiescent (i.e., sleep) cycles, which appear when halt instructions are inserted by the scheduler upon detecting idle cores.

The benefits of speculative execution are strongly application dependent and vary from 4.96% gain for *TATP* to 1.48% loss for *swaptions*. The potential gains are limited since uncommitted load(s) or unperformed store(s) preceding the atomic still prevent it from issuing. In addition, in case there is no memory operation before the atomic, the branches tend to resolve fast and therefore no significant performance gain is obtained from control-speculative paths (with the presence of fences). Nevertheless, allowing out-of-order

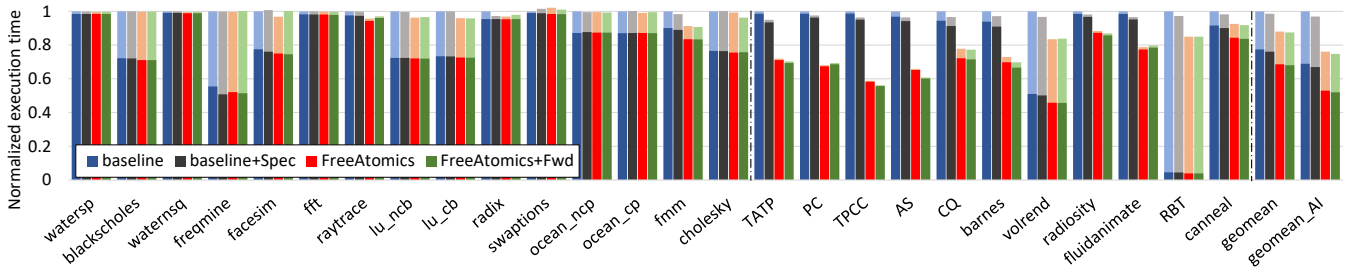


Figure 14: Normalized execution time. The bottom (shaded) part represents working time, while the top (light) part represents sleep time.

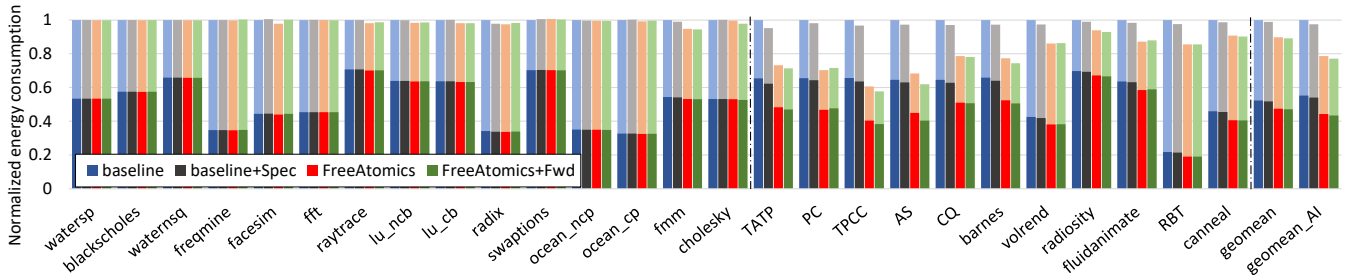


Figure 15: Normalized energy consumption. The bottom (shaded) part represents dynamic and the top (light) part represents static energy of the processor.

speculative execution is necessary to take full advantage of unfencing.

By unfencing atomic RMWs, Free atomics breaks the stall imposed by draining the SB before issuing atomic RMWs. Furthermore, Free atomics allows multiple atomic RMWs to concurrently lock cachelines and enables local access to locked cachelines. Unfencing provides the highest performance gain, with an average time reduction of 12.5% considering all applications and 25.2% when only considering atomic-intensive (i.e., AI) applications.

Finally, by enabling store-forwarding, Free atomics improves lock locality of atomic RMWs. This translates into a reduction in lock acquisition latency, further reducing the performance overhead of atomic RMWs, except for a few applications (e.g., *facesim*, *raytrace*, *PC*). Two applications, *TPCC* and *AS*, show a reduction of over 40% in their execution time, when using all Free atomics features. The hotspot of *AS* consists of a loop that selects two random data entries, locks both entries, swaps their values and unlocks. Meanwhile, *TPCC* creates a list of locks (randomized between 5 and 15 locks), acquires them and performs some computations before unlocking. A similar behavior is observed in *TATP* and *PC* (30% gain), but the hotspot loop only acquires one lock entry per iteration. In all cases, whenever there is contention for a lock between several cores, the pipeline of the waiting cores stalls due to fences. However, Free atomics allows for multiple atomic RMWs to run in parallel, improving memory-level parallelism for atomic RMWs.

5.6 Energy Efficiency

A breakdown of the energy savings in processor (the shaded part for dynamic and the light part for static) is presented in Figure 15. Static energy savings are directly proportional to performance gains. Dynamic energy is also improved due to two reasons. First, Free atomics significantly reduces wasted energy in spinning thanks to the reduction in the time atomics need to execute, mainly for those applications that show high lock contention (*TATP*, *PC*, *TPCC*, and *AS*). Second, to the increased locality of Free atomics due to its unique store-to-load forwarding feature. Indeed, we observed less committed instructions when using Free atomics with respect to the baseline atomic RMWs implementation. These reductions lead to average energy savings of 11% when considering all applications and 23% for atomic-intensive (i.e., AI) ones.

6 RELATED WORK

Moir [38] introduce restricted load-linked (RLL) and store-conditional (RSC), that are then used to implement compare-and-swap (CAS) operations. CAS operations are then used to implement regular LL/SC pairs with reduced costs, even though efficient implementations of multi-word LL/SC are proposed in [28]. These works discuss wait-free primitives at the algorithm level, and provide little information about hardware requirements for different consistency models.

Michael and Scott propose an efficient implementation of atomic RMWs [37], that relies on compare-and-swap, implemented with comparators in the caches, a write-invalidate coherence policy, and an auxiliary load exclusive instruction. Our baseline configuration is quite similar to this implementation. Designs from Intel [11, 26] include locked atomic operations whereas IBM Power [25] and RISC-V [55] include atomic memory operations (AMO). For x86 platforms, a detailed description of atomic RMWs can be found in Sewell et al. [49].

Speculative lock elision [42] can bypass the code (including atomic RMWs) guarding critical sections. Atomic RMWs, however, appear in other synchronization constructs (e.g., barriers, signal-wait) and also as standalone operations. Free atomics eliminates the cost of fences in all these cases.

Lin et al. [33] extend the ISA with Conditional Fences (C-Fences). C-Fences use compiler information to decide at runtime if the fence should be enforced or not. Conversely, we transparently remove all fences associated to atomic RMWs.

Speculative and selective memory fences [53] combine optimizations to reduce wait time due to memory fences. First, write-ahead and read-speculation let the execution window advance past a fence while the fence is active (based on [18, 19, 22]). Second, selective fences differentiate thread-local and shared data and constrain only the execution order among operations on shared data. WeeFence [14] and Address-Aware Fences [34] let post-fence loads commit before the fence. Deadlocks are avoided with significant coherence protocol modifications, additional broadcast messages, and the use of centralized structures.

Aga et al. [2] observes that the SB drain time is mainly dominated by stores that miss in cache, and propose Zfences, a mechanism that allows the on-chip directory cache to respond immediately by sending a message granting coherence permission (without data) for the requested memory write before a barrier. Fences are prevented from committing only when the SB contains stores without the coherence permission. Other proposals attempt to limit the scope of addresses affected by a fence [35], or combine different types of fences in fence groups, weak fences and strong fences, according to the criticality of a thread [13]. However, and contrary to Free atomics, these proposals are not transparent to the programmer/compiler and introduce programming challenges. Moreover, Asymmetric fences [13] require checkpointing to recover from deadlock scenarios. Asymmetric fences remain in the code, as their presence is necessary to initiate checkpointing. However, Free atomics completely remove the fence μ-ops without requiring programmer intervention and with minimal changes (no checkpointing, no coherence protocol modifications).

Deadlock avoidance has been also studied in proposals allow loads [43, 45] and stores [44] to be non-speculatively reordered while offering an ordered behaviour to the programmer. Writers-Block [43] resolves the deadlocks by guaranteeing the loads always to progress through tear-off copies. Non-speculative store coalescing [44] relies on a predetermined order of writes to solve deadlocks. Applying a predetermined order to Free atomics could lead to a reduction in timeouts, when a predetermined order in performing the stores older than the Free atomic can be guaranteed.

Shull et al. [51] propose Execution Dependencies Extensions (EDE) to eliminate fences. EDE requires extensions in the ISA to encode dependencies detected by the compiler/programmer and propagate them to the hardware. EDE targets fences used for persistent memories.

Finally, Rajaram et al. [41] propose two weaker atomicity definitions of RMWs with the goal of removing the cost of draining the SB. They avoid deadlocks by broadcasting the list of addresses to all cores and waiting for confirmation. Free atomics do not require broadcasts nor protocol modifications and address the problem of managing multiple cache locks (issuing multiple atomic RMWs) with a simple implementation (i.e., Atomic Queue). Free atomics further enable *local* accesses to locked cachelines and store-to-load forwarding to/from atomic RMWs (similar to [40] but in hardware) including from regular stores to atomics while other proposals only allowed forwarding from store unlock to regular loads [15]. All this is achieved without relaxing atomic semantics, which enables a seamless and transparent substitution in *all* synchronization idioms.

7 CONCLUSION

This work proposes and evaluates Free atomics, atomic RMW instructions implemented in hardware that are not surrounded by fences. We show that Free atomics can be executed speculatively while respecting consistency and atomicity, and discuss and address the implications of speculative and concurrent execution on correctness. Free atomics benefits from available memory-level parallelism and enables store-to-load forwarding, thus improving lock locality. Overall, Free atomics improves performance by 12.5%, on average, for a large range of parallel workloads and 25.2%, on average, for atomic-intensive parallel workloads over a fenced atomic RMW implementation.

ACKNOWLEDGMENTS

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 819134), the Swedish Research Council (VR) grant 2018-05254, the Vicerrectorado de Investigación e Internacionalización of the University of Murcia under the Talento 2021 programme, and the HiPEAC Collaboration Grants 2020. Also, we would like to thank Eduardo José Gómez Hernández for his help in part of the implementation.

REFERENCES

[1] Andreas Abel and Jan Reineke. 2019. uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In *24th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*. 673–686.

[2] Shaizeen Aga, Abhayendra Singh, and Satish Narayanasamy. 2015. zfence: Data-Less Coherence for Efficient Fences. In *29th Int'l Conf. on Supercomputing (ICS)*. 295–305.

[3] Niket Agarwal, Tushar Krishna, Li-Shiuan Peh, and Niraj K. Jha. 2009. GARNET: A Detailed On-Chip Network Model inside a Full-System Simulator. In *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*. 33–42.

[4] ARM. 2021. *ARM Architecture Reference Manual Armv8, for Armv8-A Architecture Profile*. ARM Holdings. https://developer.arm.com/documentation/102105/latest/

[5] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. 2011. Laws of Order: Expensive Synchronization in Concurrent Algorithms Cannot be Eliminated. In *38th Symp. on Principles of Programming Languages (POPL)*. 487–498.

[6] Todd M. Austin. 1999. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *32nd Int'l Symp. on Microarchitecture (MICRO)*. 196–207.

[7] Jean-Loup Baer. 2009. *Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors* (1st ed.). Cambridge University Press.

[8] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors*. Ph. D. Dissertation. Princeton University.

[9] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *17th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*. 72–81.

[10] George Z. Chrysos and Joel S. Emer. 1998. Memory Dependence Prediction using Store Sets. In *25th Int'l Symp. on Computer Architecture (ISCA)*. 142–153.

[11] Michael W. Chynoweth and Mary R. Lee. 2012. Implementing Scalable Atomic Locks for Multi-Core Intel® EM64T and IA32 Architectures. (July 2012).

[12] Edsger W. Dijkstra. 1971. Hierarchical ordering of sequential processes. *Acta Informatica* 1 (1971), 115–138.

[13] Yuelu Duan, Nima Honarmand, and Josep Torrellas. 2015. Asymmetric Memory Fences: Optimizing Both Performance and Implementability. In *15th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*. 531–543.

[14] Yuelu Duan, Abdullah Muzahid, and Josep Torrellas. 2013. WeeFence: Toward Making Fences Free in TSO. In *40th Int'l Symp. on Computer Architecture (ISCA)*. 213–224.

[15] Josué Feliu, Alberto Ros, Manuel E. Acacio, and Stefanos Kaxiras. 2021. ITSLF: Inter-Thread Store-to-Load Forwarding in Simultaneous Multithreading. In *54th Int'l Symp. on Microarchitecture (MICRO)*. 1296–1308.

[16] Agner Fog. 2020. The Microarchitecture of Intel, AMD and VIA CPUs: An Optimization Guide for Assembly Programmers and Compiler Makers. Available at https://www.agner.org/optimize/microarchitecture.pdf.

[17] Kourosh Gharachorloo. 1995. *Memory Consistency Models for Shared-Memory Multiprocessors.* Research report 95/9. Western Research Laboratory.

[18] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. 1991. Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors. In *4th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*. 245–257.

[19] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. 1991. Two Techniques to Enhance the Performance of Memory Consistency Models. In *20th Int'l Conf. on Parallel Processing (ICPP)*. 355–364.

[20] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. 2018. Persistency for Synchronization-Free Regions. In *45th Int'l Symp. on Computer Architecture (ISCA)*. 46–61.

[21] Eduardo José Gómez-Hernández, Juan M. Cebrian, J. Rubén Titos Gil, Stefanos Kaxiras, and Alberto Ros. 2021. Efficient, Distributed, and Non-Speculative Multi-Address Atomic Operations. In *54th Int'l Symp. on Microarchitecture (MICRO)*. 337–349.

[22] Lance Hammond, Mark Willey, and Kunle Olukotun. 1998. Data Speculation Support for a Chip Multiprocessor. In *8th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*. 58–69.

[23] Maurice Herlihy. 1991. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems (TPLS)* 13, 1 (Jan. 1991), 124–149.

[24] IBM. 1987. IBM System/370 Principles of Operation. *GA22-7000, IBM Data Processing Division, White Plains, NY.* (Sept. 1987).

[25] IBM Corporation. 2020. Power ISA Version 3.1. https://ibm.ent.box.com/s/hhjfw0x0lrbtyzmiaffnbxh2fuo0fog0

[26] Intel. 2021. Intel® 64 and IA-32 Architectures Software Developer's Manual. www.intel.com.

[27] ISO/IEC. 2018. *Information technology - Programming languages - C.* ISO/IEC 9899:2018.

[28] Prasad Jayanti and Srdjan Petrovic. 2005. Efficient Wait-Free Implementation of Multiword LL/SC Variables. In *25th Int'l Conf. on Distributed Computing Systems (ICDCS)*. 59–68.

[29] Eric H. Jensen, Gary W. Hagensen, and Jeffrey M. Broughton. 1987. *A New Approach to Exclusive Data Access in Shared Memory Multiprocessors.* Technical Report UCRL-97663. Lawrence Livermore National Laboratory.

[30] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2017. Language-Level Persistency. In *44th Int'l Symp. on Computer Architecture (ISCA)*. 481–493.

[31] Thomas Köppe. 2020. *Programming languages - C++.* ISO/IEC 14882:2020.

[32] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *42nd Int'l Symp. on Microarchitecture (MICRO)*. 469–480.

[33] Changhui Lin, Vijay Nagarajan, and Rajiv Gupta. 2010. Efficient Sequential Consistency Using Conditional Fences. In *19th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*. 295–306.

[34] Changhui Lin, Vijay Nagarajan, and Rajiv Gupta. 2013. Address-Aware Fences. In *27th Int'l Conf. on Supercomputing (ICS)*. 313–324.

[35] Changhui Lin, Vijay Nagarajan, and Rajiv Gupta. 2014. Fence Scoping. In *27th Conf. on Supercomputing (SC)*. 105–116.

[36] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Carlos Escuin, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Anthony Gutierrez, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Miquel Moreto, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Matthew D. Sinclair Boris Shingarov, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, William Wang, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. 2020. The Gem5 Simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152* (2020).

[37] Maged M. Michael and Michael L. Scott. 1995. Implementation of Atomic Primitives on Distributed Shared Memory Multiprocessors. In *1st Int'l Symp. on High-Performance Computer Architecture (HPCA)*. 222–231.

[38] Mark Moir. 1997. Practical Implementations of Non-Blocking Synchronization Primitives. In *16th Symp. on Principles of Distributed Computing (PODC)*. 219–228.

[39] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2020. *A Primer on Memory Consistency and Cache Coherence, Second Edition.* Morgan & Claypool Publishers.

[40] Takeshi Ogasawara, Hideaki Komatsu, and Toshio Nakatani. 2004. To-Lock: Removing Lock Overhead Using the Owners' Temporal Locality. In *13th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*. 255–266.

[41] Bharghava Rajaram, Vijay Nagarajan, Susmit Sarkar, and Marco Elver. 2013. Fast RMWs for TSO: Semantics and Implementation. In *34th Conf. on Programming Language Design and Implementation (PLDI)*. 61–72.

[42] Ravi Rajwar and James R. Goodman. 2001. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *34th Int'l Symp. on Microarchitecture (MICRO)*. 294–305.

[43] Alberto Ros, Trevor E. Carlson, Mehdi Alipour, and Stefanos Kaxiras. 2017. Non-Speculative Load-Load Reordering in TSO. In *44th Int'l Symp. on Computer Architecture (ISCA)*. 187–200.

[44] Alberto Ros and Stefanos Kaxiras. 2018. Non-Speculative Store Coalescing in Total Store Order. In *45th Int'l Symp. on Computer Architecture (ISCA)*. 221–234.

[45] Alberto Ros and Stefanos Kaxiras. 2018. The Superfluous Load Queue. In *51st Int'l Symp. on Microarchitecture (MICRO)*. 95–107.

[46] Alberto Ros and Stefanos Kaxiras. 2020. Speculative Enforcement of Store Atomicity. In *53rd Int'l Symp. on Microarchitecture (MICRO)*. 555–567.

[47] Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. 2016. Splash-3: A Properly Synchronized Benchmark Suite for Contemporary Research. In *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*. 101–111.

[48] Hermann Schweizer, Maciej Besta, and Torsten Hoefler. 2015. Evaluating the Cost of Atomic Operations on Modern Architectures. In *24th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*. 445–456.

[49] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. *Commun. ACM* 53, 7 (July 2010), 89–97.

[50] André Seznec. 2007. The L-TAGE Branch Predictor. *The Journal of Instruction-Level Parallelism* 9 (May 2007), 1–13.

[51] Thomas Shull, Ilias Vougioukas, Nikos Nikoleris, Wendy Elsasser, and Josep Torrellas. 2021. Execution Dependence Extension (EDE): ISA Support for Eliminating Fences. In *48th Int'l Symp. on Computer Architecture (ISCA)*. 456–469.

[52] CORPORATE SPARC International, Inc. 1994. *The SPARC Architecture Manual (Version 9).* Prentice-Hall, Inc.

[53] Oliver Trachsel, Christoph V. Praun, and Thomas R. Gross. 2006. On the Effectiveness of Speculative and Selective Memory Fences. In *20thInt'l Parallel and Distributed Processing Symp. (IPDPS)*. 9–pp.–.

[54] Thin-Fong Tsuei and Wayne Yamamoto. 2003. Queuing Simulation Model for Multiprocessor Systems. *IEEE Computer* 36, 2 (Feb. 2003), 58–64.

[55] Andrew Waterman and Krste Asanović. 2019. *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA.* RISC-V Foundation. https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf

[56] Sam Xi, Hans Jacobson, Pradip Bose, Gu-Yeon Wei, and David Brooks. 2015. Quantifying Sources of Error in McPAT and Potential Impacts on Architectural Studies. In *21st Int'l Symp. on High-Performance Computer Architecture (HPCA)*. 577–589.