2025 IEEE International Symposium on High-Performance Computer Architecture (HPCA)

No Rush in Executing Atomic Instructions

Ashkan Asgharzadeh†, Josué Feliu‡, Manuel E. Acacio†, Stefanos Kaxiras§, and Alberto Ros†
†University of Murcia, ‡Universitat Politècnica de València, §Uppsala University
ashkan.asgharzadehdonighi@bsc.es, jofepre@gap.upv.es, meacacio@um.es, stefanos.kaxiras@it.uu.se, aros@ditec.um.es

***Abstract*—Hardware atomic instructions are the building blocks of the synchronization algorithms. Historically, to guarantee atomicity and consistency, they were implemented using memory fences, committing older memory instructions, and draining the store buffer before initiating the execution of atomics. Unfortunately, the use of such memory fences entails huge performance penalties as it implies execution serialization, thus impeding instruction- and memory-level parallelism.**

The situation, however, seems to have changed recently. Through experiments on x86 machines, we discovered that current x86 processors manage to comply with the x86-TSO requirements while avoiding the performance overhead introduced by fences (fence-free or unfenced implementation). This paves the way to new potential optimizations to atomic instruction execution. In particular, our simulation experiments modeling unfenced atomics reveal that executing atomic instructions as soon as their operands are ready does not always lead to optimal performance. In fact, this increases the time that other threads should wait to obtain the cacheline. In contended scenarios, delaying the execution of the atomic instruction to minimize the time the cacheline is locked provides superior performance.

Based on this observation, we present Rush or Wait (RoW), a hardware mechanism to decide when to execute an atomic instruction. The mechanism is based on a contention predictor that estimates if an atomic will access a contended cacheline. Non-contended atomics execute once their operands are ready. Contended atomics, on the contrary, wait to become the oldest memory instruction and to drain the store buffer to execute, minimizing the contention on the accessed cacheline. Our experimental evaluation shows that RoW reduces execution time on average by 9.2% (and up to 43%) compared to a baseline that executes atomics as soon as the operands are ready, and yet it requires a small area overhead (64 bytes).

I. Introduction

Hardware atomic Read-Modify-Write (RMW) instructions are essential building blocks for writing parallel applications. They enable different cores (or any processing unit) in a shared-memory architecture to communicate correctly, as the application programmer expects. These atomic RMW instructions appear in the application code explicitly used by the programmer (e.g., `fetch-and-increment`), or employed by operating system libraries to build higher abstraction synchronization mechanisms such as software locks or barriers [14].

According to public Intel documentation, for performing an x86 atomic RMW [18], first, the core requires to acquire exclusive permission for the corresponding cacheline. Then, the core locks the cacheline in its local data cache, which is called *cache locking* in Intel terminology. Cache locking guarantees the atomicity of the cacheline in a multicore processor by preventing coherence requests coming from other cores from succeeding, and thus enabling correct synchronization. In cache-locking mode, the core performs its arbitrary *modify* operation (any arithmetic, exchange, or logical operation). Finally, the core updates the memory location and unlocks the cacheline, allowing the pending coherence requests from other cores to progress. In addition to this functionality, the execution of any atomic RMW requires to comply with the memory consistency model enforced by the corresponding instruction set architecture (ISA).

In x86-TSO memory model specifically, it is expected that atomic RMWs have a total order [33]. This means that there is a consensus among all processing cores on the singular execution sequence of any atomic RMW. Furthermore, reordering of either younger or older memory instructions (load or store) with respect to the atomic RMWs is prohibited. To meet these consistency requirements, atomic RMWs had been implemented so cautiously that they were considered a performance hindrance [10], [30], [32]. More precisely, the main contributors to their entailed performance overhead were the memory fences at the surroundings of the micro-operations of atomic RMWs [30]. These memory fence micro-operations not only imposed serialization by disabling instruction- and memory-level parallelism (ILP and MLP, respectively) with respect to atomics, but also delayed the *start* of the execution of atomics until they reached the head of the load queue (LQ) and the store buffer (SB) was drained [26].

Recent x86 microarchitectures manage to improve the performance of atomic RMWs, practically achieving the same latency as the non-atomic RMWs (see Section II for further details). Despite no further documented information explaining thoroughly how this performance improvement is achieved by Intel (except mentioning that "Locked instructions do not wait for all previous instructions to complete execution" [18]), our experiments on Intel machines suggest that the key performance improvement comes from the *removal of memory fences* surrounding the micro-operations of x86 atomic RMWs. Recently, in academia, Free Atomics [4] has disclosed a fence-free implementation of x86 atomic RMWs, which constitutes, as far as we know, the most detailed implementation of unfenced atomic RMWs.

Even though unfencing atomic RMWs enables ILP and MLP, starting the execution of the atomics as soon as their dependencies are resolved can increase cache-locking time considerably and prevent other threads (running on other cores in a multicore processor) from progressing for a longer period, thus resulting in significant performance loss for some applications. This is illustrated in Fig. 1, which shows for parallel applications (see Section V for further details) the normalized

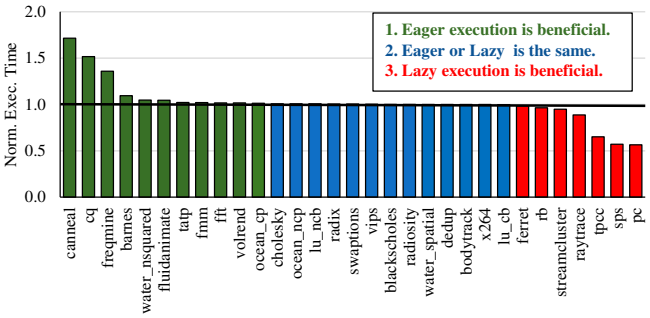


Fig. 1. Normalized execution time of a system that executes unfenced atomic RMWs *lazy* with respect to executing them *eager*. Applications are sorted from best to worst speedup of eager versus lazy execution of atomics.

execution time of a configuration that performs *lazy* execution of atomic RMWs with respect to *eager* execution. We refer to lazy execution when the core waits for the atomic RMW to reach the head of the LQ and for the SB to be drained before executing it. In contrast, eager execution executes the atomic RMWs as soon as their operands are available. Applications with green bars benefit from executing atomics eager, applications with red bars achieve better performance when executing them lazy, and finally, the ones in blue obtain the same performance either with lazy or eager execution modes.

In this work, for the first time to the best of our knowledge, we analyze the trade-off of executing early but having a large cache-locking time versus executing later and reduce cache-locking time for modern (unfenced) x86 atomic RMW instructions. We find that, in general, atomic RMWs accessing non-contended cachelines benefit from eager execution, with workloads such as *canneal* and *freqmine* reducing execution time by 42% and 26% compared to lazy execution. On the contrary, atomic RMWs accessing contended cachelines favor lazy execution. In applications such as *pc* and *sps*, which feature contended atomics, lazy execution of atomic RMWs reduces eager's execution time by 44% and 43%, respectively. Taking any of the two approaches statically yields poor performance when applications with contended atomics execute them eager or when applications with non-contended atomics execute them lazy.

To leverage this finding, we propose Rush or Wait (RoW), a hardware mechanism to dynamically decide whether a given atomic RMW should be executed eager or lazy. RoW implements a small predictor that estimates whether the execution of an atomic RMW will access a contended cacheline. If it predicts that the atomic will not face contention, RoW executes the atomic as soon as possible. Conversely, if it estimates that the atomic will face contention, RoW delays its execution until it reaches the head of the LQ and the SB is drained. When enabling forwarding from older stores to atomics, RoW executes contended atomics eager if a matching forwarding store is found to improve atomic locality.

Our simulation results, modeling unfenced x86 atomic RMWs in a cycle-accurate multicore simulator and using different parallel benchmark suites, show a 9.2% (and up to 43%) reduction in execution time, on average, when atomic RMWs execute as predicted by our contention predictor compared to a baseline that always executes atomics as soon as their operands are ready. This performance benefit is achieved with just 64 bytes of storage, a 14-bit subtractor, a 14-bit comparator, and minor hardware changes.

The main contributions of this work are:

- Report experimental results suggesting that current x86 microarchitectures may implement unfenced atomic instructions.
- Analyze the trade-off between executing atomics eager but having a larger cache-locking time versus executing them lazy, reducing cache-locking time.
- Observe that, in general, atomics accessing contended cachelines benefit from lazy execution, while those accessing non-contended cachelines benefit from eager execution.
- Propose and examine different predictors to estimate contention and determine when an atomic should be issued to get the best of eager and lazy execution.

II. Background

A. Experimental analysis on modern x86 atomic RMWs

To analyze if current x86 processors use unfenced atomic RMWs, we design a microbenchmark that allocates a large array of elements and performs RMW instructions on randomly selected elements. We study three different RMW instructions: `Fetch-and-Add` (FAA), `Compare-and-Swap` (CAS), and `Swap`. The atomic version of these RMW instructions can be used to implement multiple synchronization mechanisms [15], [32]. We devise four variants of the microbenchmark depending on how the RMW instruction is performed: non-atomically (without the `lock` prefix), atomically (with the `lock` prefix), and each of them without and with explicit memory fences (`mfences`) inserted before and after the RMW instruction. Because the size of the array allocated by the microbenchmark exceeds the cache capacity and the elements are accessed randomly, the memory access latency is high. Hence, the performance impact of memory fences is highly noticeable. When a memory fence is executed, the memory accesses retrieving the elements on which the operations are performed get serialized. When the memory fence is removed, the MLP grows and execution overlaps the memory latency of different accesses, achieving higher performance. The microbenchmark runs a single thread that performs 100M iterations. Each iteration consists of picking a random element and performing the corresponding operation on it.

Fig 2 shows the number of cycles per iteration for each microbenchmark on two x86 processors with different microarchitectures. The Intel i5-9400F is a recent x86 processor. It belongs to the Coffee Lake microarchitecture and was launched in 2019. On the contrary, the Intel Xeon X3210 is a relatively old x86 processor, which belongs to the Kentsfield microarchitecture and was launched in 2007. The experimental

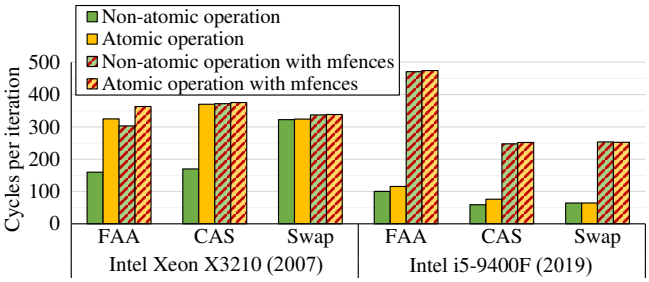


Fig. 2. Cycles per iteration of microbenchmarks using different atomic RMW instructions with/without the lock prefix/explicit `mfences`.

results show that, in *old* x86 processors, atomic RMW instructions incur the cost of fences. This observation is backed by the fact that the cycles per iteration when adding the lock prefix to perform the RMW instruction atomically approximately double.[1] In addition to that, manually adding an *mfence* instruction before and after the atomic RMW instruction does not have any impact on performance. We believe that these results can only be explained by the fact that, in former x86 processors, atomic RMW instructions internally enforce the behavior of a memory fence. In contrast, the experimental results reveal that in the recent x86 processor atomics RMW instructions do not incur the cost of fences. In this case, adding the lock prefix to the operation does not significantly increase the number of cycles per operation. However, when manually inserting the memory fences before and after the RMW instruction (either non-atomic or atomic), performance drops to roughly a fourth. We think that such performance degradation can only be explained by the fact that recent x86 processors do not internally enforce the behavior of a memory fence to implement atomic RMW instructions.

Summing up, our experimental results show that recent x86 processors manage to comply with the x86-TSO requirements without suffering the performance overhead introduced by fences, which suggests that they may implement unfenced atomic RMW instructions. Although we only report results for a recent processor from Intel, the same behavior was observed with different recent x86 processors both from Intel and AMD. In the next sub-section we elaborate on Free Atomics [4], a proposal from the academia to implement unfenced atomic RMWs, which is perhaps the closest description of the current (undisclosed) implementation of atomic RMWs in x86 processors.

B. Free Atomics

Free Atomics [4] is a lightweight hardware mechanism that removes the implicit memory fences surrounding the atomic instructions, thus improving performance while preserving atomicity and x86-TSO consistency. Removing fences allows the concurrent execution of multiple atomics. In-flight atomics

[1]The Swap operation is implemented with the `xchgl` instruction. In this case, when a memory operand is referenced, the processor's locking protocol is automatically implemented for the duration of the exchange operation, regardless of the presence or absence of the lock prefix [18]. This explains the fact that roughly the same performance is observed for the non-atomic and atomic cases with this operation.

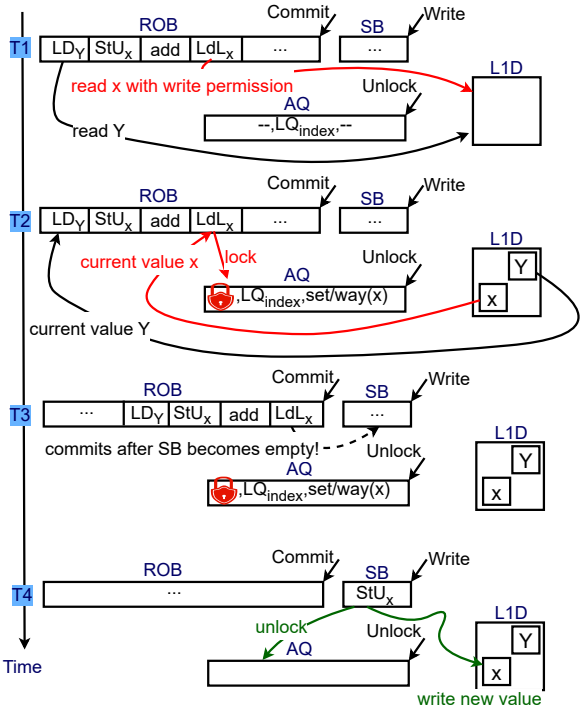


Fig. 3. Implementation and execution of unfenced atomic RMW fetch-and-increment in an x86 processor according to Free Atomics.

are tracked by a microarchitectural unit called Atomic Queue (AQ). The AQ is a FIFO queue that resides near the SB.

Through a simple example in Fig. 3, we illustrate how Free Atomics provides cache locking, compliance with the x86-TSO memory model, and enables ILP and MLP in an unfenced implementation of atomic RMWs. Note that in this figure, the time labels (T1, T2, T3, and T4) only represent the functionality of the atomic RMWs, but not the latency that each action needs to perform.

The figure illustrates the execution of an atomic fetch-and-increment instruction on a cacheline (x), consisting of three different micro-operations: 1) load micro-operation (called load_lock in Free Atomics terminology and depicted as LDL_x); 2) addition (ADD); and 3) store micro-operation (called store_unlock in Free Atomics terminology and depicted as STU_x), that reside in the reorder buffer (ROB) of a x86 core, followed by a regular load instruction (LD_Y) on a different cacheline (Y). Once the micro-operations are dispatched to the pipeline, in addition to taking entries in the ROB, and an entry in LQ and SQ for its load and store micro-operations, respectively, an entry in the AQ is reserved for the atomic instruction (e.g., represented by the store micro-operation). In the designated AQ entry, among other information, each atomic keeps the index of the corresponding LQ entry that is allocated at dispatch time, enabling the load micro-operation to recognize its matching entry in the AQ. In this figure, an arrow at the head of ROB, SQ, and AQ points to from which side of these units an atomic instruction commits, writes to L1 data cache (L1D), and unlocks the cacheline, respectively.

Free Atomics allows the atomic increment to execute as soon as its operands are ready, even if there are older un-

committed memory instructions or the SB is not empty (Time T1). Therefore, the LDL_x asks for exclusive permission for the cacheline (x), and once granted and when the cacheline is located in the L1D, it marks the cacheline locked by setting the locked bit in the corresponding AQ entry. Moreover, LDL_x annotates the set and way of the cacheline's physical address in the AQ entry (time T2). This enables the core to stall any invalidation or downgrade message that arrives at the core involving cacheline (x) or a potential eviction of this cacheline from the L1D. By snooping the AQ, all those messages will find the cacheline locked, thus preserving atomicity. Around time T1 (either a few cycles before or after it), the younger regular load instruction (LD_Y) can also initiate its execution *speculatively*, improving ILP and MLP.

After reading and locking the cacheline (Time T2), the addition is performed on the corresponding value of x. Once the atomic increment reaches the head of the ROB, it has to wait to drain the SB before leaving the ROB (Time T3). This guarantees total memory ordering with x86 atomic RMW instructions (see [4] for a thorough explanation). After committing the atomic increment, all its micro-operations leave the ROB. However, note that the STU_x still resides in the SB and remains there until it performs the write operation and releases the lock. Once the atomic instruction commits, all younger instructions (such as the LD_Y) are allowed to commit once they reach the head of the ROB without waiting for the STU_x to leave the SB. In this way, unfencing can bring significant performance improvements.

Finally, at time T4, the STU_x writes the new value and unlocks the cacheline by clearing the corresponding AQ entry (and leaving the SB). Note that the stores in TSO memory model happen in order and the SB and AQ entries were both allocated simultaneously at the dispatch time. Therefore, at unlock time, the head of the SB corresponds with the head of AQ. Also, note that the write by the STU_x can happen almost immediately, as the SB was ensured to be empty before committing the atomic, and the cacheline was already locked and located in the L1D.

Although unfenced atomic RMWs have the potential to significantly improve ILP and MLP by overlapping the execution of the atomic instructions with older and younger instructions, we next demonstrate that the time to initiate the execution of the atomic instructions is critical to achieve optimal performance in some relevant scenarios.

III. Motivation

As demonstrated in Section II-A, current x86 processors implement unfenced atomics.[2] This allows atomic instructions to start executing as soon as their operands are ready, without waiting until they become the oldest memory instruction in the pipeline and the SB is drained. We refer to this way of executing atomics as *eager* execution. Executing atomics as soon as possible should theoretically improve performance as it eases hiding their execution latency under the shadow of older instructions. However, each atomic operation locks a cacheline when it executes, even with older unresolved memory accesses and long dependency chains, and the lock is only released after the store micro-operation writes and leaves the SB. Before, it needs the SB drains in order to commit without jeopardizing any consistency guarantee provided by TSO. This means that atomic operations that are executed eager can hold cachelines locked for long periods, which can hurt performance when operating on contended cachelines that are also requested by other threads.

An alternative option to reduce the time cachelines are locked is to delay issuing atomics until certain conditions are met while still allowing younger instructions to execute ahead of the atomic. Suppose that we want to minimize the time the cachelines are locked. In that case, we can restrict issuing atomics until they become the oldest memory instruction in the pipeline and the SB is empty. We use the term *lazy* execution to refer to the execution of atomics with these constraints. Notice that it allows instructions younger than the atomic to execute speculatively before the atomic does, and so it differs from a fenced implementation of atomics. While lazy execution of atomics minimizes the time cachelines are locked, when the atomic misses in the cache, the entire access latency is exposed in the critical path.

In this work, we make the observation that unfenced atomics bring in the need to decide which is the optimal time to issue an atomic. Fig. 4 shows that the backward and forward dependency chains of atomics do not prevent the eager and lazy execution of them from being relevant[3]. The first bar of the figure presents the number of instructions older than an atomic that are not executed yet when the atomic issues eager. This metric shows that the execution of an atomic can effectively start *soon* in most cases because when its register dependencies are resolved, there are still, on average, 48 older instructions than the atomic to be executed. The second bar depicts the number of instructions younger than an atomic that have already started their execution when the atomic issues lazy. This metric shows that, in many workloads, the lazy execution of atomics does not prevent younger instructions from starting execution. This is the case with workloads that strongly benefit from lazy execution, such as *tpcc*, *sps*, and *pc*, which start speculatively executing more than 50 instructions younger than the atomic before it executes. Few instructions can start executing before the atomic in other workloads such as *streamcluster* and *raytrace*. This does not prevent the lazy execution to improve performance as it still reduces the cacheline locking time but reduces its potential benefits. Summing up, as Fig. 1 illustrates, choosing between eager and lazy execution of atomics has an overwhelming impact on the performance of several applications. Although eager execution of atomics roughly reduces the execution time of *canneal* and *freqmine* to half compared to lazy execution, it nearly doubles

[2]From now on, for concise writing, we simply use "atomic RMWs" (or "atomics") rather than "unfenced atomic RMWs".

[3]In this figure and the followings, only the atomic-intensive applications are depicted (see Section V).

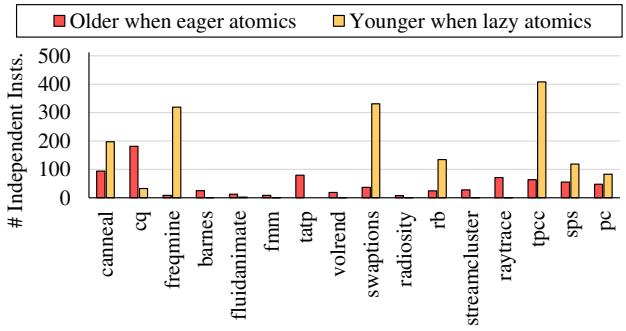


Fig. 4. Number of independent instructions with respect to eager and lazy atomics.

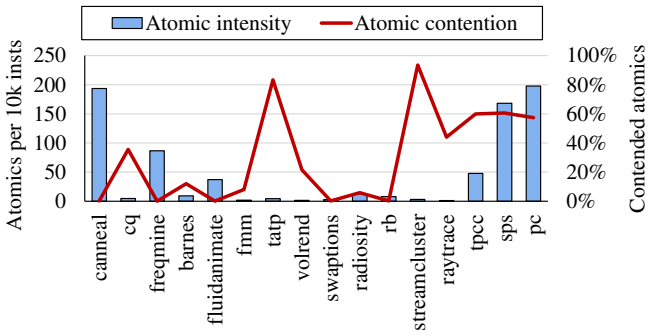


Fig. 5. Atomics per 10 kilo-instructions and percentage of atomics that face contention.

the execution time of *pc* and *sps*.

To shed some light about the reasons for this divergent performance, we show two statistics. First, Fig. 5 depicts the atomic intensity and contentiousness of the applications, which helps understand the performance trend of the eager and lazy execution of atomics presented in Fig. 1. Second, in Fig. 6, we demonstrate how the eager and lazy execution of atomics impacts the latency of atomic instruction's themselves.

Fig. 5 shows the atomics per 10 kilo-instructions executed by the applications (blue bars and left y-axis) and the percentage of the atomics that face contention with eager execution (red line and right y-axis). An atomic is considered contended when it accesses a cacheline concurrently used or requested by another thread. The higher atomic intensity of the applications in the left and right parts of the figure explains their higher performance sensitivity to the way atomics are executed. The higher contentiousness of the atomics in *tpcc*, *sps*, and *pc*, along with their high atomic intensity, explains their superior performance with lazy execution, as we discuss next.

Fig. 6 presents the atomic instruction's latency, which is broken down into three parts: 1) dispatch to issue, 2) issue to lock, and 3) lock to unlock. The first and second latency bars for each application represent eager and lazy execution, respectively. Essentially, lazy execution increases the waiting time a ready-to-execute atomic spends to become the oldest memory instruction in the pipeline and for the SB to become empty to get issued (larger *blue* segments). Nevertheless, this minimizes the time a cacheline remains locked by the atomic (*yellow* segments) and reduces the time to acquire the cacheline (*orange* segments) as fewer threads (cores),

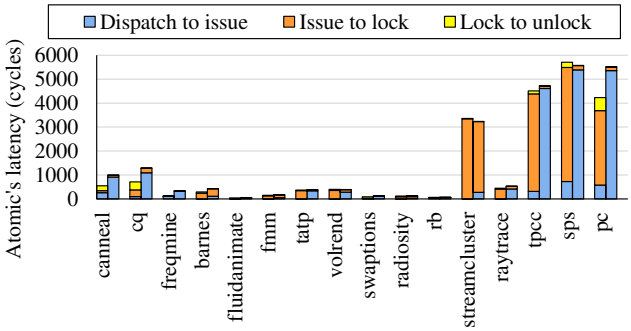


Fig. 6. Comparison of time needed from dispatch until write between eager (1st bar) and lazy (2nd bar) execution of atomic RMWs.

compared to eager execution, request the cacheline (only the threads having the atomic RMW at the head of the LQ and SB empty) and the cacheline is released quicker after executing the atomic instruction. This behavior benefits applications with contended atomics (e.g., *tpcc*, *sps*, and *pc*), on which the issue-to-lock latency explodes with eager execution due to the longer time that cachelines remain locked.

In contrast, if the cachelines used by the atomics are not contended, lazy execution decreases performance because the application favors executing atomics (and acquiring the cacheline) as soon as the atomics are ready to execute (e.g., *canneal*, *freqmine*). In this way, atomic instructions hide part of their memory latency while executing older memory instructions.

Following the previous discussion, the behavior of *cq* is slightly contradictory. It shows low atomic intensity, but atomics strongly impact its performance. Furthermore, it is relatively contended, but favors the eager execution of atomics. The reason for this behavior is atomic locality. In *cq*, the eager execution of atomics allows keeping a cacheline accessed by previous instructions in L1D to execute the atomic concurrently through store-to-load forwarding while, with lazy execution, the cacheline is invalidated and should be fetched again by the atomic, exposing its access latency in the critical path. RoW takes these observations into account and also performs well under this scenario (see Section VI).

Summing up, the previous observations make it clear that there is a tradeoff between eager and lazy atomics. Locking the cacheline earlier is productive as long as other cores do not need such a cacheline while it is locked. Otherwise, when a core holds a cacheline locked for a long period and makes other cores wait, eager atomics can quickly turn detrimental to performance.

IV. Rush or Wait: Predicting Contention

For atomic-intensive workloads, issuing atomics in an eager or lazy way has an unyielding impact on performance. Therefore, detecting contention is the key to leveraging this enormous potential impact. As discussed in the previous section, non-contended atomics should be issued eager to hide the access latency of the atomic as much as possible. On the contrary, contended atomics should be issued in a lazy way to minimize the time a cacheline is locked and hence reduce the average time that threads wait to acquire such a cacheline.

However, detecting contention is not straightforward. The next subsections discuss how the contention estimation mechanism of Rush or Wait (RoW) evolves from a basic approach that fails to capture contentiousness efficiently to an advanced version that more precisely captures it.

A. Contention detection based on execution window

When the load micro-operation of the atomic executes, it computes its address and requests the accessed cacheline with exclusive permissions. Once the cacheline arrives at the private cache, it is locked until the atomic operation is performed, and the updated data is written back to the cache.

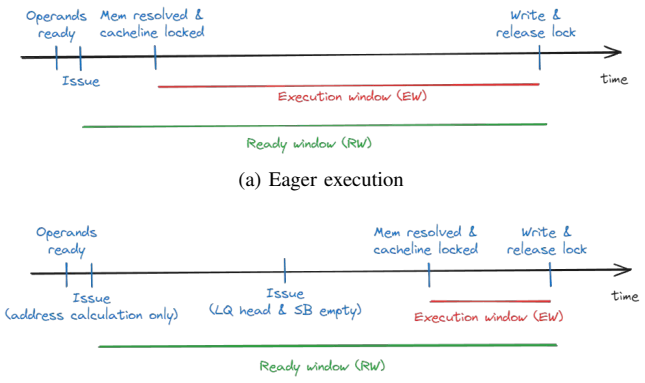
The simplest approach to identify contention is to monitor the invalidation and downgrade messages (we will refer to them as *external requests*) from the coherence protocol that reach a core for locked cachelines. When one of these messages arrives, the core marks the atomics that access the corresponding cacheline as contended. (Needless to say, the core delays the coherence response until the atomic writes and unlocks the cacheline.)

Implementation. Our baseline core already models unfenced atomics as in Free Atomics (see Section II-B). Thus, it includes an AQ that keeps track of the atomic instructions in flight. When the core receives an external request, it searches the AQ to determine if the cacheline involved in the operation is locked, in which case it needs to stall the response to that message. Using this search in the AQ, RoW simply marks a matching atomic as contended, setting an additional bit (*contended* bit) added to each AQ entry. This bit will be used to train a contention predictor (see Section IV-D) when the atomic completes. The overhead of this contention detection mechanism is just 16 bits, since we model a 16-entry AQ.

B. Extending the execution window: Ready window

The previous mechanism only detects contention when the cacheline is locked. This approach works well with eager execution and when the memory request hits in the L1D, as it locks the cacheline immediately. Fig. 7a (red window) illustrates this case. Notice, however, that the longer it takes to resolve the memory request of the load (moving the cacheline locking closer to the atomic write), the shorter the execution window for detecting contention.

This effect exacerbates when atomics execute lazy, as illustrated in Fig. 7b (red window), making the mechanism less effective in detecting contention. With lazy execution, atomics are only issued once they become the oldest instruction in the LQ and the SB is empty. This minimizes the window during which a cacheline is locked, reducing it to fewer than 10 cycles on average. Even for highly contended cachelines, such a short window reduces the probability that a core receives an external request from the coherence protocol while the cacheline is locked. Note that if a core holds a cacheline and an external request reaches the core before an atomic starts its execution and locks the cacheline, the core will simply invalidate/downgrade the cacheline, but it will not identify it as contended.



(a) Eager execution

(b) Lazy execution

Fig. 7. Contention-detection windows with eager and lazy execution.

To enable a core to track contention in a larger window and both in eager and lazy modes, we propose that an atomic starts monitoring external requests as soon as its operands are ready. Fig. 7a and Fig. 7b (green windows) show the extended contention-tracking window (called *ready window*) approach. In particular, with lazy execution, the ready window allows identifying requests from other cores that would be delayed in case the atomic had been executed eager.

Implementation. To track external requests for cachelines accessed by atomics in the ready window, RoW needs to calculate the address of atomics as soon as their operands are ready. Thus, we propose that regardless of the atomics execution policy (eager or lazy), atomics get issued once their operands become available. However, if the atomic should execute lazy because it is predicted that it will access a contended cacheline, we record that situation using a *only-calculate-address* bit. The atomic then goes through the address calculation stage and accesses the TLB to obtain its physical address. This address is stored in the AQ entry assigned to the atomic. At this point, if the *only-calculate-address* bit is set, the instruction is brought back to the issue stage, waiting to issue a second time once the conditions to execute lazy are met. The *only-calculate-address* flag is also used to avoid releasing the issue queue and LQ entries and to avoid issuing any dependent instruction in the following cycles, simplifying the actions needed to take the atomic back to the issue stage. Note that contemporary cores already delay the execution of instructions depending on a load that is expected to miss (or bank-conflict) in the cache [18]; thus, no additional hardware is needed to perform this task. Also, calculating the effective address for an atomic when the operands are ready, but not issuing the memory request to the L1D until the execution policy dictates it, is similar to what happens to regular store instructions at the execution stage (calculate the effective address but wait to write to the L1D after committing and reaching the head of SB [18]). When the atomic meets the conditions to execute lazy, it is issued again. Now, it can simply copy its address from the AQ entry to the LQ entry, skipping the address calculation and TLB access, since the address it computed previously is still valid (assuming that the address is invalidated on TLB shutdowns).

In addition, as we discussed in Section IV-A, RoW should

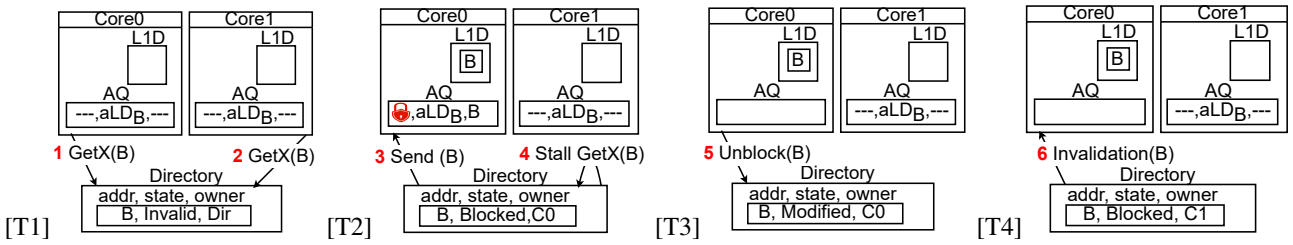


Fig. 8. External requests for contended cachelines might reach the core after it has lifted the cacheline lock, which makes only tracking these requests an insufficient approach to detect contention in all cases. [T1] Two atomics request the cacheline, [T2] Core0 acquires the cacheline, and locks it in AQ, [T3] Atomic in Core0 commits and unlocks the cacheline fast, before Unblock message reaches the directory, [T4] Invalidation reaches Core0 while the atomic has already completed execution; the atomic in Core0 could not see the contention at the directory.

search the AQ on an external request to find if the accessed cacheline is locked and consequently stall the coherence message. We propose that during this search, if the cacheline is not locked but any atomic in the AQ matches the address, RoW marks the *contended* bit for those atomics, effectively extending the contention-tracking window. This can be done in parallel with snooping the LQ and squashing the loads that match the address of the external request. Extending the contention detection window from the execution window to the ready window only requires 16 additional bits.

C. Tracking contention in the coherence directory

The previous approach only detects contention when an external request reaches the L1D during the ready window. In some cases, the external requests for contended cachelines will come after the cache lock is lifted. The reason lies in how cache coherence protocols function.

Fig. 8 illustrates how coherence transactions affect the cores tracking contention, yielding the previous approach insufficient to identify contention in all cases. Initially, a cacheline (address `B`) is stored in the directory (i.e., `Invalid` state in the directory, [T1] in Fig. 8 - directory state). Atomics from different cores request the cacheline (transactions 1 and 2). The directory sends the cacheline to the first request arriving to it (core0, transaction 3) and stalls the request of core1 until core0 acknowledges receipt of the cacheline (step 4). Until the directory receives an acknowledgment from core0, the cacheline remains in a transient state, called `Blocked`, in the directory ([T2] - directory state) and no invalidation requests are sent to core0. Once core0 receives the cacheline in its L1D, it locks the cacheline in the AQ, initiates execution, and replies back to the directory with an `Unblock` message to confirm the reception of the cacheline (message 5). Now, the directory marks core0 as the owner of the cacheline with new value (`Modified` directory state at [T3]). Then the request from core1 is processed by the directory sending an invalidation request to core0 (transaction 6). Meanwhile, the directory state is changed to Blocked state again, but for core1 ([T4] - directory state). Finally, the unblock/invalidation round trip typically takes long time, meaning that the atomic may have finished execution, unlocked the cacheline, and drained the AQ when the invalidation is seen (empty AQ at core0 - [T4]).

The described problem can be solved in different ways. One solution would be to detect contention at the directory and use the directory response messages to inform cores that the cacheline is being accessed by other cores. To keep the coherence protocol intact, we opt for an alternative approach in RoW. In particular, contention can be detected when the sender of the cacheline is a remote private cache and the latency of the transaction acquiring the cacheline is larger than the typical transaction round trip without contention. Such a simple approach works well because the latency for contended cachelines is higher than that for non-contended ones and because capturing the sender being a remote cache filters long latency accesses to the last level cache or memory. Furthermore, the more atomics from different cores request the cacheline, the higher the latency becomes.

Implementation. We consider that a cacheline is contended *when it arrives at the core from another core's private cache and its latency exceeds a threshold*. Coherence messages commonly include the sender identifier (or at least, a bit to indicate if the response comes from private or shared caches) [9]. To obtain the latency that the request took to be resolved, RoW augments each AQ entry with a field (*request issued cycle*) that stores the cycle on which the `getX` request to acquire the cacheline was sent. When the core receives the cacheline in the L1D and locks it in the AQ, the *request issued cycle* is subtracted from the current cycle to obtain the request latency. If the cacheline arrives from a private cache and its latency exceeds the threshold, the corresponding atomic is considered contended and marked accordingly, setting the *contended* bit in the AQ entry. As previously discussed, this bit will be used to train the contention predictor.

Our experimental results show that a threshold of 400 cycles is optimal to differentiate between accesses to contended and non-contended cachelines, even though contended accesses typically greatly exceed that value. To record the issue time of the `getX` request, we add a 14-bit field to each AQ entry. When the load micro-operation of an atomic is issued, this field is initialized with the 14-least significant bits of the current cycle. No additional hardware is needed to keep track of the processor cycles as they are already accounted for in current processors and used, for example, in performance counters [18]. When the response arrives, the 14 less significant bits of the current cycle minus the bits stored in the *request issued cycle* field using unsigned arithmetic gives the

latency of the request. [4] Therefore, a 14-bit field per AQ entry, a 14-bit unsigned subtractor, and a 14-bit comparator for the AQ are enough to detect if the latency of an atomic acquiring the cacheline exceeds 400 cycles.

D. A simple predictor structure

The previous subsections discuss different approaches to identify when an atomic faces contention (i.e., when it has accessed a cacheline concurrently used or requested by at least one additional core) during its execution. Now, we briefly discuss the implementation of the predictor to anticipate if an atomic will face contention during its execution. The predictor is checked in the allocation stage using the program counter (PC) of the atomic. If the prediction states that the atomic will not face contention, RoW will execute it eager, seeking to hide its execution latency in the shadow of older instructions. On the contrary, if the prediction states that the atomic will face contention, RoW will execute it lazy, with the goal of locking the cacheline for a period as short as possible to allow other cores to access the contended cacheline earlier. Later, when the atomic releases the cacheline lock, it updates the predictor using the *contended* bit of its AQ entry.

More precisely, the goal of the contention predictor is to estimate the probability of facing contention that a core has during the execution of an atomic when it is ready to execute. However, the contention matching a particular instance of an atomic is hard to predict as it depends highly on timeliness and thread synchronization. Hence, we just aim to predict the probability that any instance of that atomic, considering its PC, has of facing contention based on past events.

We explore two simple N-bit saturating-counter predictors ($N = 4$ in our experimental evaluation) with 64 entries in RoW. The predictor uses XOR-mapping [13] and is indexed with the 6-least significant bits of the PC XORed with the following 6 bits. PC-locality is not relevant for the workloads whose atomics do not face contention (e.g., *canneal* or *freqmine*). However, when the workloads combine contended and non-contended atomics, the fewer the number of entries in the predictor, the higher the aliasing. If contended and non-contended atomics share the same predictor entry, some of them will not be executed with the optimal policy, which will degrade performance. This is the case, particularly, of the applications that benefit from lazy execution such as *tpcc*, *sps*, and *pc*, which fail to execute contended atomics lazy due to aliasing when the number of entries of the predictor is reduced. Using a single predictor entry for all atomics, for instance, causes a performance degradation by 0.3% on average compared to eager execution.

We explore two options to update the saturating counters:

1) UpDown: Upon seeing contention on the cacheline being used by an atomic, the counter assigned to that atomic is incremented. On the contrary, seeing no contention will decrement the counter. If the value of its counter is below or equal a threshold ($= 1$ in our experiments), it will execute eager; otherwise, it will execute lazy.
2) Saturate on Contention: Upon seeing contention, the counter with any current value will be maximized (i.e., 2^N -1). Similar to the previous method, if no contention is seen, the value of the counter will be decremented. The decision and manner to execute eager or lazy is the same as with the previous predictor (for this predictor, threshold's value equals 0).

The predictors, with their update policies and thresholds to estimate contention, move the execution of an atomic aggressively towards lazy when it faces contention, as contended atomics usually benefit from it. They also favor recent contention behavior rather than historical behavior to predict contention. Finally, we also evaluated other predictors, such as a predictor that performs +2/-1 when detecting/not detecting contention, but observed that the up/down and saturate predictors reach higher performance benefits.

E. Favoring atomic locality

Enabling atomic instructions to receive their data through store-to-load forwarding, either from older regular stores or from older atomics, potentially accelerates the execution of the forwarded atomic as it guarantees that the core does not invalidate or evict the cacheline between the forwarding instruction writes and the forwarded atomic executes. Free Atomics [4] allows chains of forwarding between atomics, on which an older atomic forwards its data to a younger atomic without releasing the cacheline lock. The authors, however, acknowledge that these chains of atomics can lead to livelocks, and thus, they limit the forwarding chains to a length of 32. The hardware necessary to implement this constraint is not discussed, though.

We also analyze the effect of forwarding in this work. In our implementation, we allow an atomic to be forwarded from an older matching regular store. This simple mechanism offers average performance benefits by improving atomic locality, it is simple to implement, and prevents unlimited forwarding chains of atomics that may restrict thread-level parallelism when involving contended cachelines.

To improve atomic locality, when an atomic predicted to issue lazy finds potential of forwarding from an older regular store (done when computing its address eager), the atomic continues its execution eager. The fact that the older store needs to execute anyway and will contend for the cacheline, already invalidating other cores' cacheline, mitigates the negative impact of executing the atomic eager. When the forwarding chance exists and we execute the atomic eager, we avoid losing the cacheline between the store and atomic instructions, improving the atomic locality and the performance.

Implementation. Favoring atomic locality when executing atomics requires turning the lazy execution of an atomic into eager. This implies that atomics that are predicted contended, and thus, should execute lazy, not only compute their address (to extend the contention-tracking window as explained in Section IV-B) on their *only-calculate-address* issue but also

[4] With this approach, a latency between 16,384 (2^{14}) and 16,784 cycles would be misinterpreted as lower than the threshold. However, reaching such a large latency is highly unlikely.

need to search the SB for a matching store. Fortunately, the number of atomics is small compared to the number of loads, and the additional searches to the SB do not significantly increase the pressure in its search ports. If the atomic finds a matching (non-atomic) store, RoW turns its execution into eager. To this end, it needs to clear the *only-calculate-address* bit in its corresponding AQ entry and to free its issue queue entry. Otherwise, if there are no chances of forwarding and the atomic remains lazy, it is moved back to the issue stage to start its execution when it becomes the oldest memory instruction and the SB drains.

F. Memory requirements

The memory overhead of RoW is divided into two components: 1) the memory overhead of the contention predictor, and 2) the memory overhead of augmenting the AQ entries to keep track of contended atomics and timestamps.

The contention predictor is implemented as a 64-entry table of 4-bit saturating counters, used to track contention and is incremented/decremented as discussed in Section IV-D. The table requires 256 bits (i.e., 64 entries $\times$ 4 bits).

To detect contended cachelines, we rely on both the ready window mechanism and the coherence directory detection, so we augment each AQ entry with the following fields: *contended* bit, *only-calculate-address* bit, and a 14-bit *issued-cycle* timestamp. Given that the AQ only implements 16 entries, it is augmented with 256 bits (i.e., 16 entries $\times$ (1 + 1 + 14) bits). Overall, the memory requirements of RoW amount to only 64 bytes.

V. Methodology

To assess the proposed mechanism, we use a detailed in-house, out-of-order processor model that incorporates Free Atomics [4] as a state-of-the-art implementation of unfenced atomic RMWs. As discussed in Section II, and despite the implementation details have not been disclosed, recent x86 processors also implement atomic RMWs without fences. Our processor model is driven by a Sniper [7] front-end, which feeds the processor with the instructions and their micro-operations. The memory hierarchy and cache coherence protocol are modeled with the cycle-accurate GEMS simulator [24], and the interconnect is modeled with GARNET [1].

We simulate a multicore processor consisting of 32 cores. The processor parameters are based on the characteristics of the performance cores of the Intel Alder Lake microarchitecture [31]. The memory hierarchy parameters resemble the same microarchitecture as well. Table I presents the most relevant configuration parameters. To support unfenced atomic RMWs, each processor core incorporates a 16-entry AQ.

We run parallel workloads from Splash-4 [12] and PARSEC 3.0 [6] benchmark suites, which are relatively synchronization poor even though some of them are still atomic intensive, and a suite of six fine-grain synchronization-intensive workloads [11], [20]. This set of workloads provides a wide range of behaviors with different degrees of atomics per instruction and different sensitivities to the performance of the atomic instructions. From these three suites, we select and evaluate the results for the workloads that have at least one atomic per 10 kilo-instructions (considered as atomic-intensive) as the execution of atomics impact noticeably their performance. Nevertheless, we also report the overall execution time of RoW when considering all applications. All workloads run 32-threads. We report statistics for the parallel phase of the applications. In all the evaluations, reported execution times are normalized with respect to eager execution of atomics without forwarding.

TABLE I
System parameters

Processor	
Cores	32
Fetch / Issue / Commit Width	6 / 12 / 12 instructions
ROB / LQ / SB	512 / 192 / 128 entries
Atomic queue	16 entries
Branch predictor	TAGE-SC-L [34]
Mem. dep. predictor	StoreSet [8]
Memory	
Private L1I cache	32KB, 8 ways, 4 hit cycles
Private L1D cache	48KB, 12 ways, 5 hit cycles, IP-stride prefetcher
Private L2 cache	1MB, 8 ways, 12 hit cycles
Shared L3 cache	4MB per bank, 16 ways, 35 hit cycles
Memory access time	160 cycles

VI. Experimental evaluation

Fig. 9 shows the normalized execution time of applications when executing atomics eager, lazy, or with different variants of RoW, using the three contention-detection mechanisms proposed paired with the two predictors analyzed: UpDown (U/D) and Saturate on Contention (Sat). For these experiments, forwarding to atomics is disabled. With RoW, atomics are only issued eager if no contention is predicted. The lower the normalized execution time the better. As discussed in Section III, the applications on both extremes have higher atomic intensity (see Fig. 5), yielding higher performance variation depending on when atomics are executed. Furthermore, atomic-intensive yet not-contended applications such as *canneal* and *freqmine* favor the eager execution of atomics while the contended ones (e.g, *pc* and *sps*) favor the lazy execution. *Cq* presents a slightly different behavior that will be discussed later.

We start analyzing the performance of the execution window (EW) contention-detection mechanism against the lazy and eager execution of atomics. With the non-contended applications, the EW performs as well as the eager execution of atomics, which means that it does not predict contention when there is not. With the contended applications, it improves performance in *sps*, getting close to the lazy execution, but performs significantly worse than either lazy or eager executions in *tpcc* and *pc*. Overall, these results demonstrate that the EW mechanism is an insufficient approach to detect contention.

The ready window (RW) extends the window on which external requests are tracked to better capture contention for the cachelines accessed by atomics. For the non-contended workloads, the RW performs similarly to both the eager execu-

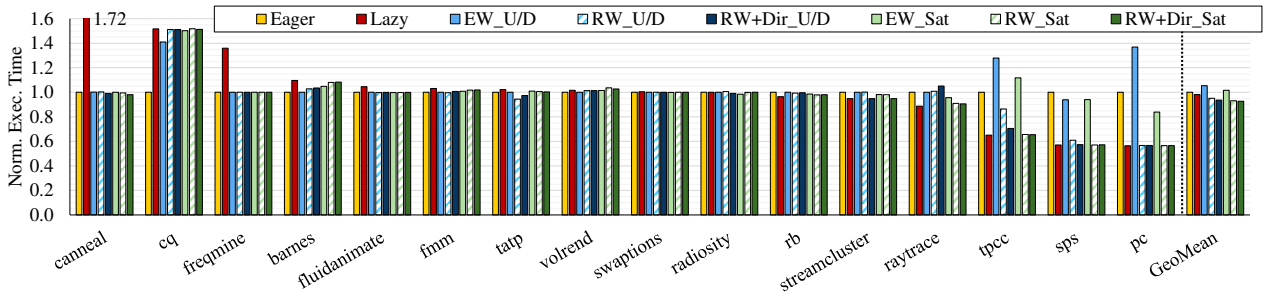


Fig. 9. Normalized execution time for different variants of RoW (EW, RW, and RW+Dir contention detection mechanisms paired with the up/down (_U/D) and saturating (_Sat) predictors) compared to eager and lazy execution. No store-to-load forwarding for atomics.

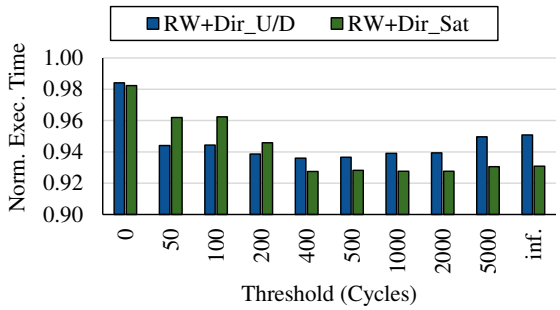


Fig. 10. Sensitivity of RoW to the latency threshold when using RW+Dir contention detection mechanism.

tion and the EW predictors, outperforming the lazy execution of atomics. For the contended workloads, the RW predictors are able to estimate contention accurately and perform close to the lazy execution of atomics in *pc*, *sps*, *tpcc*, and *raytrace*.

Finally, the ready window enhanced with the directory contention mechanism (RW+Dir) impacts performance more noticeably when paired with the up/down predictor (RW+Dir_U/D – dark blue bar). In this case, it improves the performance of *tpcc*, *streamcluster*, and *sps* non-negligibly and provides minor benefits across several other applications. Its impact with the saturating predictor (RW+Dir_Sat – dark green bar) is lighter because, for a contended atomic, the saturating predictor needs not to face contention fifteen consecutive times before the prediction for the atomic moves to not contended.

Fig. 10 show the sensitivity of the RW+Dir contention detection mechanism to the latency threshold. The performance sensitivity to this parameter is very limited as this mechanism works on top of the RW. A very large threshold (i.e., inf.) prevents detecting atomics that face large latency due to accessing contended cachelines acquired from other cores as contended and behaves like RW. A very small threshold, like 0 cycles, marks any atomic receiving the cacheline from another core as contended. This imposes overhead to atomic-intensive applications that infrequently face contention (e.g., *canneal*) by issuing its atomics lazy. The optimal threshold is achieved at 400 cycles. However, note that the latency to access contended cachelines is much larger than to access non-contended cachelines, and therefore simplifies the tuning of this parameter. In fact, the performance variation of increasing the threshold from 400 cycles to 2000 cycles is minimal.

Overall, the RW+Dir_Sat predictor achieves the best performance, reducing the execution time of both the eager and lazy execution of atomics by 7.3% and 5.5%, respectively. It also reduces the execution time of the RW+Dir_U/D predictor by 0.9%, and that of the EW and RW contention detection mechanisms paired with the saturating counter predictor by 8.8% and 0.4%. Despite the RW+Dir predictors provide the highest performance on average, they impose a small slowdown in applications such as *tatp*, *barnes*, and *raytrace* compared to the other contention-detection mechanisms. To a lower extent, but similarly to *cq*, these benchmarks feature contended atomics that benefit from eager execution. Thus, with more accurate contention detection mechanisms, more contended atomics are detected and executed lazy, which degrades performance in this case. We will later analyze how RoW deals with these applications to provide the best performance.

To support our claim that executing contended atomics eager results in longer cacheline locks and leads to greater access latency for other threads accessing the contended cachelines, we look at the miss latency of the memory instructions. Fig. 11 shows the average miss latency for the eager and lazy execution of atomics, and for RoW with the RW+Dir_U/D and RW+Dir_Sat predictors. For applications with more contended atomics (e.g., *pc*, *sps*, and *tpcc*; see Fig. 5), the miss latency with eager execution nearly doubles the latency with lazy execution, which confirms our claim. By executing contended atomics lazy, the miss latency achieved by RoW with both predictors closely resembles that of the lazy execution. For applications with non-contended atomics, the miss latency difference is minimal, as whether the atomic is executed earlier or later does not impact the miss latency directly. However, the timing of atomic execution does impact performance, as executing the atomics earlier helps hide their latency under the shadow of older memory accesses.

Prediction accuracy. Fig. 12 shows the accuracy of the up/down (U/D) and saturating (Sat) predictors to detect contention. That is, how often they predict that an atomic will or will not face contention and it is detected accordingly with the corresponding RW+Dir contention detection mechanism. The U/D predictor shows better accuracy across all benchmarks, reaching an average accuracy of 86% in the contention predictions. With the saturating counter, the prediction accuracy drops to 73% because it moves too aggressively towards predicting contention in atomics whose contentiousness is not completely byassed to contention. The lower accuracy of

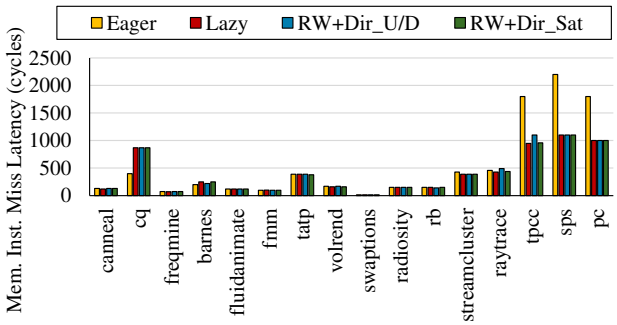


Fig. 11. Comparison of L1D miss latency for all memory instructions with eager and lazy execution of atomics and RoW using the RW+Dir contention detection mechanism.

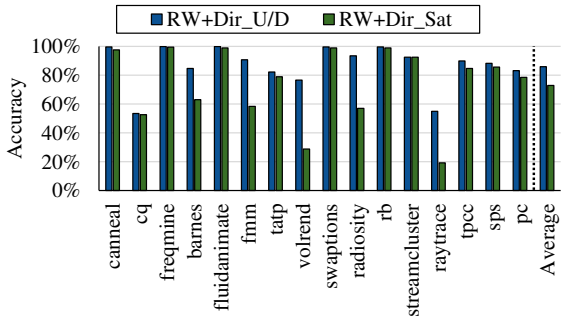


Fig. 12. Accuracy of contention detection by RoW.

this predictor, however, does not directly translate into lower performance because the accuracy difference mainly comes from benchmarks such as *fmm*, *voldrend*, and *radiosity*, which have no performance sensitivity to the eager/lazy execution of atomics (see Fig. 1) caused, in part, by their low atomic intensity (see Fig. 5). For *barnes*, the lower accuracy of the satuaring predictor compared to the U/D predictor results in lower performance. On the contrary, for *raytrace*, the saturating predictor performs better than the U/D despite its lower accuracy because it predicts contention more frequently and *raytrace* benefits from executing atomics lazy (see Fig. 1).

Enabling forwarding to atomics. In Fig. 9, *cq* clearly stands out as an application that noticeably differs from the others since it has contended atomics but benefits from eager execution. Albeit less evident, *barnes* and *tatp*, as previously discussed, also feature contended atomics that benefit from eager execution. The reason that explains this behavior is atomic locality. These applications feature atomics that execute after a store to the same cacheline. Despite these atomics are contended, they benefit from eager execution because, even when forwarding to atomics is disabled, executing the atomics earlier allows locking the cacheline before it is invalidated after the store writes. However, when RoW predicts that the atomic is contended and executes it lazy, the cacheline may be taken by another core, the atomic misses in the cache, and a longer access latency is exposed. This issue is also illustrated in Fig. 11, which shows the average miss latency. In *cq* or *barnes*, the lazy execution of atomics and the execution with RoW results in higher miss latency due to the worse atomic locality. In *tatp*, the average miss latency is similar in all configurations because even with the lazy execution of atomics, the atomic locality is frequently preserved.

To improve atomic locality, we enable forwarding from stores to atomics and extend RoW to execute a contended lazy atomic eager when it detects a matching store in the SB. Fig. 13 depicts the execution time, normalized to eager execution, for lazy execution, eager execution with forwarding (Eager+Fwd), and different variants of RoW, without and with forwarding enabled. Note that the lazy execution of atomics cannot benefit from forwarding to atomics as it requires draining the SB before executing an atomic, hence it is not shown in the figure. The eager execution of atomics slightly benefits from forwarding to atomics when there is an opportunity. This is mainly visible in *cq* and *tatp*, which have the highest number of forwarded atomics. The benefit is small, though, because in both cases, the eager execution of atomics already achieves good atomic locality as it manages to keep the cacheline in the private cache between the store write and the atomic execution.

Unlike the eager execution of atomics, RoW executed all contended atomics lazy, which was negatively impacting the atomic locality. Consequently, with the optimization to promote atomic locality, RoW reaches the highest benefit when forwarding is enabled. In particular, *cq*'s execution time reduces by 35% with the RW+Dir_U/D predictor and the forwarding optimimzation compared to the execution with the same predictor but without forwarding. As mentioned earlier, other applications such as *barnes* and *tatp* also benefit from the higher atomic locality and reduce their execution time by 5.2% and 1.5%, respectively. Summing up, with forwarding from stores to atomics enabled, RoW reaches the best performance with the RW+Dir_U/D combination, reducing the execution time compared to the eager and lazy execution of atomics by 9.2% and 8.5%, respectively. Considering all the applications (atomic-intensive and non-atomic intensive), RoW achieves a 4.0% performance improvement compared to a baseline that executes all atomics eager.

VII. Related work

Comparison with branch prediction. Our contention predictor is similar to the well-known bimodal branch predictor [35] regarding the policy for updating the counters (although we examined both up/down and saturate on contention predictors). However, to improve the accuracy of contention prediction, increasing the counter width, recording a history [37], or using complex algorithms (e.g., [19]) is not required. Instead, we proposed simple yet effective approaches to detect contended atomics. Detecting contention for atomics should be local (having individual entries per atomic) as they are uncorrelated as opposed to branches that might correlate in some scenarios [27].

Efficient implementation of atomic RMWs. The performance overhead imposed by fenced atomic implementations has been widely analyzed in the related work [10], [30], [32]. Our experimental analysis suggests that x86 processors may already implement unfenced atomics to avoid it (see Section II-A). Recently, Free Atomics [4] presented a hardware mechanism to implement unfenced atomics in

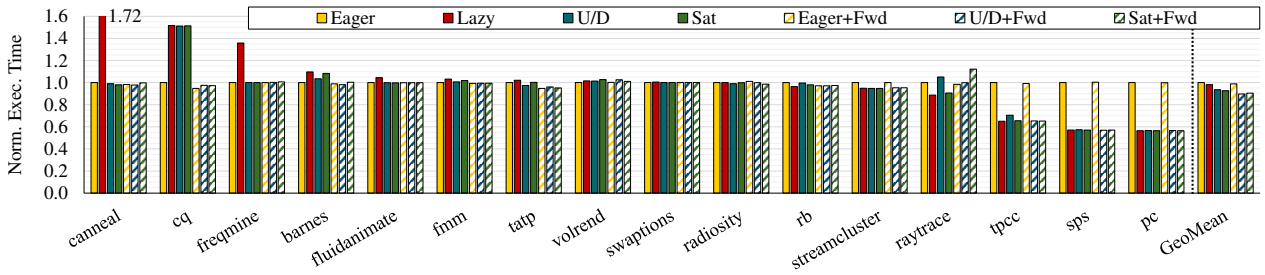


Fig. 13. Normalized execution time of eager and lazy execution of atomics and different variants of RoW when enabling forwarding from stores to atomics compared to the eager execution of atomics without forwarding. RoW variants use the RW+Dir contention detection mechanism and include the optimization to improve atomic locality.

x86 processors while preserving the atomicity and x86-TSO consistency. Regarding fenced atomics, Kurth et al. [21] proposed hardware support for RISC-V processors to execute atomic RMWs (called AMO in RISC-V ISA) based on the LoadLinked/StoreConditional (LL/SC) approach, which are interruptible peers to atomics. Mestan et al. [25] seeks to favor locality for regular loads with a predictor that estimates when LL/SCs will be successful and hence can forward its data to a following load instruction [25]. Finally, CLAU [5] suggests adopting hardware cache locking to improve the performance of *non-atomic* RMWs against contention stemmed from false-sharing. In this work, we study the trade-off between executing unfenced atomics eager or lazy and devise a mechanism to execute them at the best time by predicting whether they will face contention or not.

Near or far execution of atomics. Although this work focuses on when is the suitable time to execute unfenced atomics, there is an orthogonal decision on where is the right location to perform atomic memory updates. This decision varies between local caches (namely *near* atomics) and the last level shared cache (namely *far* atomics). In x86 machines, only near atomic RMWs are offered, while IBM offers only far atomic RMWs [16]. ARM processors provide both near and far atomics, with an emphasis on near atomics for better performance [2]. In recent work, the locality among atomics [28] and the contention over the accessed cacheline [17] were key factors to decide between near and far atomics.

Conflict management in Hardware Transactional Memory. In Hardware Transactional Memory (HTM), many works have been proposed to handle the conflict between transactions running on different cores [3], [23], [29]. Lupon et al. [23] propose using different version and conflict management policies for transactions depending on whether they face contention or not. Even though we share the idea of taking a different execution approach depending on contention, HTM systems are greatly different. They identify contention from transaction conflicts, which completely differs from our contention detection mechanism, and their version and conflict management policies are unrelated to the eager and lazy execution of atomics we propose. DeTras is the latest work that achieves this goal efficiently with low hardware cost via delaying conflicted stores within a transaction [36]. DeTras uses a predictor (SCH) similar to our *Sat* version, but tracks conflicted stores, which are detected by adding one bit to the coherence protocol messages.

Self invalidation to reduce shared cachelines access latency. Lai and Falsafi proposed *Last-Touch-Predictor (LTP)* to enable each core to invalidate its shared cachelines at a suitable time before being asked by other cores [22]. Their proposal relies on collecting different traces for each shared cacheline, starting from an L1D miss, until another core sends an invalidation. Similar to our work, LTP predicts over instructions (yet in our case only atomic instructions) rather than cachelines.

VIII. Conclusion

Our experimental analysis on x86 processors suggests that they may have recently adopted unfenced atomics, avoiding the performance overhead of the traditional fenced atomic implementation while still complying with the consistency model guarantees. In our work, we raise the concern that when to execute unfenced atomics is critical for performance. Executing atomics as soon as possible helps hide their access latency but also leads to longer cacheline locks, which can negatively impact performance when the accessed cacheline is contended.

To decide when to execute atomics, we propose Rush or Wait (RoW). RoW implements a contention predictor to anticipate if an atomic will not face contention, in which case it is executed eager, or will face contention, in which case it is executed lazy. In addition, to improve atomic locality, RoW moves to eager execution of contended atomics when a matching forwarding store is available. The experimental results show that RoW reduces the execution time on average by 9.2% (and up to 43%) compared to a baseline that executes atomics eager with a hardware overhead of only 64 bytes.

Acknowledgment

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), from the MCIN/AEI/10.13039/501100011033/ and the "ERDF A way of making Europe", EU (grants PID2021-123627OB-C51 and PID2022-136315OB-I00), from the MCIN/AEI/10.13039/501100011033/ and the European Union NextGenerationEU/PRTR (grants RYC2021-030862-I and TED2021-130233B-C33), from the Swedish Research Council (VR grant 2022-04959), and from the Swedish Foundation for Strategic Research (SSF grant FUS21-0067).

REFERENCES

[1] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2009, pp. 33–42.

[2] ARM, *ARM Architecture Reference Manual, for A-profile architecture*, ARM Holdings, 2024. [Online]. Available: https://developer.arm.com/documentation/ddi0487/latest/

[3] A. Armejach, J. R. T. Gil, A. Negi, O. S. Unsal, and A. Cristal, "Techniques to improve performance in requester-wins hardware transactional memory," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 10, no. 4, pp. 42:1–42:25, Feb. 2013.

[4] A. Asgharzadeh, J. M. Cebrian, A. Perais, S. Kaxiras, and A. Ros, "Free atomics: hardware atomic operations without fences," in *49th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2022, pp. 14–26.

[5] A. Asgharzadeh, E. J. Gómez-Hernández, J. M. Cebrian, S. Kaxiras, and A. Ros, "Hardware cache locking for all memory updates," in *42th IEEE International Conference on Computer Design (ICCD)*, Nov. 2024, pp. 566–574.

[6] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, Jan. 2011.

[7] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations," in *Conf. on Supercomputing (SC)*, Nov. 2011, pp. 52:1–52:12.

[8] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," in *25th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 1998, pp. 142–153.

[9] D. E. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, Inc., 1999.

[10] M. K. Elteir, H. Lin, and W. Feng, "Performance characterization and optimization of atomic operations on AMD gpus," in *IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2011, pp. 234–243.

[11] V. Gogte, S. Diestelhorst, W. Wang, S. Narayanasamy, P. M. Chen, and T. F. Wenisch, "Persistency for synchronization-free regions," in *39th Conf. on Programming Language Design and Implementation (PLDI)*, Jun. 2018, pp. 46–61.

[12] E. J. Gómez-Hernández, J. M. Cebrian, S. Kaxiras, and A. Ros, "Splash-4: A modern benchmark suite with lock-free constructs," in *Int'l Symp. on Workload Characterization (IISWC)*, Nov. 2022, pp. 51–64.

[13] A. González, M. Valero, N. Topham, and J. M. Parcerisa, "Eliminating cache conflict misses through XOR-based placement functions," in *11th Int'l Conf. on Supercomputing (ICS)*, Jun. 1997, pp. 76–83.

[14] M. Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems (TPLS)*, vol. 13, no. 1, pp. 124–149, Jan. 1991.

[15] M. Herlihy, N. Shavit, V. Luchangco, and M. Spear, *The art of multiprocessor programming*. Elsevier, 2020.

[16] IBM Corporation, "OpenPOWER Foundation Power Instruction Set Architecture," May 2024. [Online]. Available: https://www.ibm.com/docs/ar/aix/7.2?topic=reference-instruction-set

[17] J. Ingalls, W. W. Terpstra, H. Cook, and L. Kou, "Method for executing atomic memory operations when contested," U.S. Patent US11467962, Oct. 2022.

[18] Intel, "Intel® 64 and ia-32 architectures software developer's manual," www.intel.com, Mar. 2024.

[19] D. A. Jiménez, "Fast path-based neural branch prediction," in *36th Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2003, pp. 243–252.

[20] A. Kolli, V. Gogte, A. Saidi, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Language-level persistency," in *44th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2017, pp. 481–493.

[21] A. Kurth, S. Riedel, F. Zaruba, T. Hoefler, and L. Benini, "Atuns: Modular and scalable support for atomic operations in a shared memory multiprocessor," in *57th Design Automation Conference (DAC)*, Jul. 2020, pp. 1–6.

[22] A. C. Lai and B. Falsafi, "Selective, accurate, and timely self-invalidation using last-touch prediction," in *27th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2000, pp. 139–148.

[23] M. Lupon, G. Magklis, and A. González, "A dynamically adaptable hardware transactional memory," in *43rd Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2010, pp. 27–38.

[24] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, Sep. 2005.

[25] B. R. Mestan, G. N. Levinsky, and M. L. Karm, "Atomic operation predictor," U.S. Patent US11119767, Mar. 2022.

[26] V. Nagarajan, D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence, Second Edition*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2020.

[27] S. Pan, K. So, and J. T. Rahmeh, "Improving the accuracy of dynamic branch prediction using branch correlation," in *5th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Oct. 1992, pp. 76–84.

[28] V. S. Pardos, A. Armejach, T. Mück, D. S. Gracia, J. A. Joao, A. Rico, and M. Moretó, "Dynamo: Improving parallelism through dynamic placement of atomic memory operations," in *50th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2023, pp. 30:1–30:13.

[29] S. Park, C. J. Hughes, and M. Prvulovic, "Forgive-tm: Supporting lazy conflict detection in eager hardware transactional memory," in *28th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2019, pp. 192–204.

[30] B. Rajaram, V. Nagarajan, S. Sarkar, and M. Elver, "Fast rmws for tso: Semantics and implementation," in *34th Conf. on Programming Language Design and Implementation (PLDI)*, Jun. 2013, pp. 61–72.

[31] E. Rotem, A. Yoaz, L. Rappoport, S. J. Robinson, J. Y. Mandelblat, A. Gihon, E. Weissmann, R. Chabukswar, V. Basin, R. Fenger, M. Gupta, and A. Yasin, "Intel Alder Lake CPU architectures," *IEEE Micro*, vol. 42, no. 3, pp. 13–19, Mar. 2022.

[32] H. Schweizer, M. Besta, and T. Hoefler, "Evaluating the cost of atomic operations on modern architectures," in *24th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2015, pp. 445–456.

[33] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors," *Communications of the ACM*, vol. 53, no. 7, pp. 89–97, Jul. 2010.

[34] A. Seznec, "TAGE-SC-L branch predictors," in *JILP – Championship Branch Prediction*, Jun. 2014, pp. 1–8.

[35] J. E. Smith, "A study of branch prediction strategies," in *8th Int'l Symp. on Computer Architecture (ISCA)*, May 1981, pp. 135–148.

[36] R. Titos-Gil, R. Fernández-Pascual, M. E. Acacio, and A. Ros, "Detras: Delaying stores for friendly-fire mitigation in hardware transactional memory," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 33, no. 1, pp. 1–13, Jan. 2022.

[37] T. Yeh and Y. N. Patt, "Alternative implementations of two-level adaptive branch prediction," in *19th Int'l Symp. on Computer Architecture (ISCA)*, May 1992, pp. 124–134.