Hardware Cache Locking for All Memory Updates

Ashkan Asgharzadeh[1], Eduardo José Gómez-Hernández[1], Juan M. Cebrian[1], Stefanos Kaxiras[2], Alberto Ros[1]
[1]Computer Engineering Department, University of Murcia, Murcia, Spain,
[2]Department of Information Technology, Uppsala University, Uppsala, Sweden,
{ashkan.asgharzadeh, eduardojose.gomez, jcebrian}@um.es, stefanos.kaxiras@it.uu.se, aros@ditec.um.es

***Abstract*—Many applications need to perform operations that involve reading a value from memory, modifying it, and then writing it back. Multiple architectures provide hardware support for these operations via read-modify-write (RMW) instructions. The primary benefit is that the read can request a cacheline with write permissions, reducing coherence protocol overhead since the write will find the cacheline with appropriate permissions. RMWs can be either atomic or non-atomic. Atomic RMWs, used for synchronization, commonly require (i) locking the cacheline to guarantee atomicity by preventing invalidations and (ii) enforcing serialization of instructions in the program (e.g., via memory fences), which may cause performance degradation based on the implemented memory consistency model. Non-atomic RMWs, while not requiring such strict measures, should only be used in data-race free code sections. However, other cores may invalidate a cacheline during a non-atomic RMW (e.g., due to false sharing), flushing the pipeline and causing the loss of write permissions obtained by the read, which is detrimental to performance.**

In this work, we propose a microarchitectural mechanism that enables non-atomic RMWs to fetch the cacheline locking it, thus preventing other cores from "stealing" the cacheline while allowing them to run concurrently with other instructions in the same core. Our proposal enables concurrent hardware cache locking for multiple non-atomic RMWs while guaranteeing deadlock freedom and no programmer/compiler intervention. We also propose a *lock-chaining* mechanism to allow multiple consecutive memory updates to the same cacheline up to a predefined maximum (to prevent starvation and load imbalance). Our evaluation using gem5 full-system simulator shows that for an eight-core configuration, our proposal improves performance by up to 5.36% (2.05% on average), requiring just 45 bytes of storage per core.

***Index Terms*—Multi-core architectures, micro-architecture, non-atomic Read-Modify-Write, false sharing, hardware cache locking.**

I. Introduction

Read-Modify-Write (RMW) instructions, either atomic or non-atomic, perform the functionality of three individual operations within one: 1) read (load current value from memory), 2) modify (update current value with new value), and 3) write (store new value to memory). As the data being read will be modified and written by the same instruction, a well-known optimization is to request the data to the cache hierarchy with write permissions (also known as read-for-ownership [16], [22]). This optimization enables the store micro-operation to write the data to the L1 data cache (L1D) sooner since it is found in the correct coherence state. In addition, regular (non-atomic) RMW instructions,[1] which are prevalent in CISC architectures, do not have any atomicity guarantees, that is, the data being operated on can potentially be accessed and modified by other cores while the instruction is not yet completed (has not written its data into the L1D). This is why RMWs should only be used in data-race free (DRF) code sections [1].

When RMW instructions are contended, i.e., two or more cores are accessing simultaneously the cacheline updated by the RMW instruction, their performance can be sub-optimal. Being RMW expected to be used in DRF regions, this contention is primarily due to the well-known issue of false sharing [23], which occurs when different data (i.e., non-overlapping memory bytes) are located in the same cacheline. This can happen for various reasons, such as decisions made by the compiler or the memory allocator. While software can mitigate false sharing by optimizing data layouts and employing padding, complete elimination is difficult due to several inherent challenges [5], [8], [17], [20]. These include dynamic memory allocation, unpredictable runtime access patterns, complex data structures, reliance on legacy code and libraries, and performance trade-offs with data alignment.

When false sharing occurs and multiple cores attempt to update the same cacheline using RMWs simultaneously, the cacheline is frequently ping-ponged between the cores. This back-and-forth transfer results in two significant performance drawbacks. First, the invalidation, in most memory consistency models, squashes the RMW [11] and forces a pipeline flush that becomes increasingly costly with deeper pipelines. Second, the store micro-operation loses its exclusive permission before writing to the L1D, generating additional network traffic to retrieve the required permissions.

In this paper, we propose to adopt hardware *cache locking*, a familiar concept used by atomic RMWs [14], to improve the performance of (non-atomic) RMWs by making them resistant to contention during their execution. We name our proposal CLAU (Cache Locking for All Updates). Cache locking enables a RMW to hold the cacheline during its execution; from the moment that cacheline has been located in L1D and read by the load micro-operation until when the new value is written by the store micro-operation. Cache locking

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 819134), from the MCIN/AEI/10.13039/501100011033/ and the "ERDF A way of making Europe", EU (grant PID2022-136315OB-I00), and from the MCIN/AEI/10.13039/501100011033/ and the European Union NextGenerationEU/PRTR (grant TED2021-130233BC33).

[1]We refer to non-atomic RMW instructions as RMW instructions or RMWs

does not delay the cache access and brings two advantages. First, the invalidation will wait instead of squashing the RMW. Second, the store micro-operation will find the cacheline with exclusive permission at L1D, irrespective of contention. Cache locking is not intended to convert (non-atomic) RMWs into atomic ones, and can be disabled by the hardware. We offer an efficient hardware implementation that enables multiple RMWs to perform cache locking concurrently while executing speculatively, without facing deadlocks or livelocks in multi-core processors.

In addition to cache locking, we provide an optimization to exploit cache locality that enables cores to perform several RMWs to the same cacheline without lifting the cache locking. We call this feature *lock chaining*. This further improves performance by making a RMW find the cacheline already present and locked in L1D without initiating additional coherence protocol requests. However, a threshold in the maximum number of chained RMWs needs to be added to avoid load imbalance or starvation in other cores requesting the same cacheline.

We evaluate CLAU in the full-system simulator gem5 [21] with x86 cores using a large set of parallel workloads (see VI). Our simulation results show that RMW-induced pipeline flushes vanish (e.g., from 62 in a million instructions to 0), when cache locking and lock chaining are applied to all RMWs compared to a baseline in which RMWs are exposed to contention. As a result, performance improves up to 5.36% (2.05% on average). Our proposal entails a small area overhead of just 45 bytes per core.

The main contributions of this work are:

- Identifying the performance penalty of RMWs caused by contention resulting from false sharing.
- Using cache locking mechanism to enhance the performance of RMWs by making them resistant to contention.
- Presenting an efficient and deadlock-free hardware solution that supports concurrent cache locking for multiple RMWs.
- Exploiting locality by executing multiple RMWs without losing write permission (lock chaining).

II. Background

In x86-TSO [28] (total store order) consistency model, (non-atomic) RMWs have no special considerations compared to regular loads and stores. The load micro-operation (load micro-op) of the RMW can initiate its execution speculatively and request the cacheline with write permission. Speculative RMWs should be squashed when the accessed data is invalidated by another core (or evicted from L1D) [11]. This is to guarantee the TSO memory consistency model, by avoiding a possible load-load reordering observed by other cores. Commit is performed in order when they reach the head of the reorder buffer (ROB). Store micro-operation (store micro-op) remains in the store buffer (SB) to write into L1D complying with TSO.

Hardware cache locking is a mechanism used by atomic RMWs to help guarantee atomicity [14]. This feature locks the cacheline in a core upon reading it (with write permission), either in L1D or another structure inside the core, until an atomic RMW updates the cacheline by writing a new value to it, thus releasing the lock. This enables the core to deny invalidation or downgrade messages to that cacheline while locked. Moreover, cache locking avoids L1D replacement policy to choose a locked cacheline as a victim for eviction. Lastly, cache locking only stalls external requests; local operations can access the locked cacheline and read the value [11].

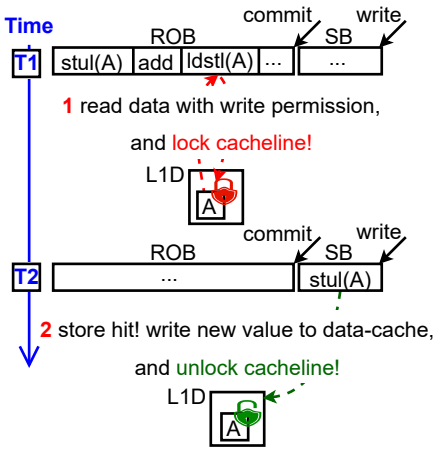


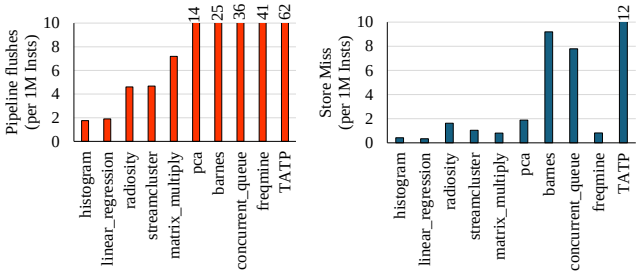
Fig. 1: Functionality of cache-locking RMWs

Fig. 1 illustrates the cache-locking mechanism for an atomic RMW that performs an addition to the data in address A. We use the same terminology as the x86-TSO gem5 model, and disregard fences. atomic RWMs are split into three micro-operations. `ldstl` loads the data (`ld`), while asking for the write permission (`st`) of the cacheline and locks it (`l`) in the core (Time T1 - 1). The data is passed to the `add` micro-operation to perform the addition. Later when the atomic RMW commits, the store micro-op, `stul`, enters the SB. Finally, when `stul` reaches the head of SB, it writes the new value (`st`) to the L1D and unlocks (`ul`) the cacheline (Time T2 - 2).

III. Motivation

As previously mentioned, RMWs are squashed when the core faces an invalidation (or loss of write permission in general, e.g., downgrade or eviction) to the cacheline they access. Unless the code is racy, these invalidation messages should be related to false sharing, but still cause performance degradation.

Fig. 2 shows a quantitative study on the impact of such scenario, which motivates CLAU. The left chart (Fig. 2a) shows the number of pipeline flushes per million instructions triggered by the load micro-op of a RMW. This happens when a RMW has not committed yet, and an invalidation matches the cacheline accessed by the load micro-op, residing in the load queue (LQ). We show, only applications that face more than one flush caused by a RMW per million instructions (see section VI for details on the methodology). Out of the ten shown applications, five face more than 10 flushes per



(a) Pipeline flushes by invalidation (b) Store Miss

Fig. 2: Drawbacks of contention between non-atomic RMWs

million instructions. Fig. 2b shows the second reason for the slowdown: the number of times the store micro-op lacks exclusive permission when it is about to write to the L1D (i.e., store miss), per million instructions. This happens when an invalidation or downgrade message arrives when the RMW has already committed (and therefore can not be squashed), but steals the write permission from the store micro-op residing in the store queue (SQ) or SB. Three of the shown applications have around 10 misses per million instructions. Since pipeline flushes are commonly more costly than cache misses, our key source of improvement lies in avoiding RMW squashes.

IV. CLAU: Cache Locking for All Updates

CLAU is based on two insights: i) we can minimize the performance overheads of false sharing by enabling cache locking for multiple speculative RMWs and ii) if several of these speculative RMWs target the same cacheline, we can hold the lock across their execution to reduce coherence traffic. Therefore, we implement two components: *cache-locking RMWs*, and *lock-chaining RMWs*.

A. Cache-Locking RMWs

To avoid pipeline flushes caused by RMWs, CLAU attempts to initiate cache locking for all RMWs. Therefore, several cacheline locks may be active at a particular time. This case happens as non-atomic RMWs do not disable instruction or memory level parallelism (ILP/MLP) with respect to them even when adopting cache locking (opposed to atomic RMWs that serialize execution [3]). As mentioned in related work [3], [9], [13], [24], when using multiple locks, attention needs to be paid to deadlocks, mainly because a cacheline may remain locked forever. Some deadlocks are related to the lock acquisition order on speculative cache locking and the limitations of the x86-TSO consistency model [3], [9], [24]. For instance, a load or store in a core cannot complete execution as the cacheline is locked in another core by a cache-locking RMW, and the cache-locking RMW itself cannot unlock the cacheline as there is a pending load or store ahead of it in the pipeline for the same reason (a cyclic dependency between two cores) [3]. Other deadlocks are related to cache inclusion property (this includes any cache level hierarchy and any shared micro-architectural units like Miss Status Holding Register) or to the replacement policy not being able to find not-locked victims [3], [13]. As an example, suppose there is a pending store ahead of an already executed cache-locking RMW in the pipeline that needs to be mapped in the same directory entry assigned to the cache locking RMW. The directory sends an invalidation to all the memory hierarchy but the cache locking at L1D refuses eviction. Hence, the store remains pending, and cache-locking RMW cannot unlock the cacheline either (as stores write in-order in x86-TSO), and eventually, the system will end in a deadlock [3].

Since cache locking is just a performance optimization and does not affect the correctness of RMWs, we can release the locks at any time using a watchdog mechanism without the need to flush the pipeline. RMWs just behave as in the baseline when that happens. The timeout threshold for the watchdog depends on the application characteristics and network contention, and therefore we perform a design space exploration in section VII-B. Still, deadlocks are rare, so a large enough threshold will practically allow all RMWs to use cache locking until the release of lock.

In addition, CLAU needs to guarantee that a cacheline will be unlocked either when the RMW is completed (the store writes to L1D) or squashed, such that the cacheline does not remain locked forever. Although cache-locking RMWs are immune to invalidations from external requests, they can be squashed inside the core due to their speculative execution (e.g., branch misprediction or memory dependency violation). If a cache-locking RMW has already locked the cacheline and is squashed it should unlock the cacheline. Releasing the lock at squash is consistent with canceling any micro-architectural state updates a squashed instruction performs.

In some cases cache-locking RMWs can still face LQ snoop matches carried out by invalidation requests. This scenario happens when an invalidation finds the target cacheline unlocked. The expected behavior would be to squash the RMW. However, we noticed that cache locking offers RMWs an extra guarantee: cache-locking RMWs do not need to be squashed on matching LQ snoops if they have not executed yet. The reason is that in this case the cache-locking RMW has not loaded the data yet (otherwise the invalidation would have been stalled by cache locking), and therefore no need to squash it. On the other hand, If a cache-locking RMW has already executed, but the cacheline is unlocked (e.g., watchdog mechanism), then invalidation should squash the cache-locking RMW; guarantee the memory consistency.

A final aspect is that the load micro-op of RMWs can obtain its data through store-to-load forwarding, similar to any regular load instruction. Once RMWs are devised with cache locking, this well-known optimization is still valid, but with a small consideration for simplicity: *a forwarded cache-locking RMW only locks a cacheline if it has already acquired write permission.* Otherwise, if forwarding happens but the cacheline does not have write permission, the cache-locking RMW is reverted to the baseline behavior. A further consideration is whether to disable cache locking when forwarding takes place. In section VII-B, we show that this alternative improves

performance, so it is better to skip cache locking even if the cacheline has write permission, in the case of store-to-load forwarding.

B. Lock-Chaining RMWs

The same cacheline can be *accessed* by multiple cache-locking RMWs in the same core. If these cache-locking RMWs fit inside the instruction window of the core, they can benefit from locality by not relinquishing the cacheline lock. Therefore we propose to maintain the cacheline locked for multiple RMWs. We call this technique *lock chaining* and it favors local updates over external requests. However, a cap on chaining the lock is needed to avoid starvation in other cores asking for the same cacheline (see section VII-B).

Fig. 3 shows an example of lock chaining with two non-atomic RMWs that use cache locking. First, at time T1 - 1, the $\texttt{ldstl}_1$ of the first cache-locking RMW executes and locks cacheline (A). As mentioned before, cache locking only prevents external cores from accessing the data, while local load instructions can read from an already locked cacheline. Hence, $\texttt{ldstl}_2$, from the second cache-locking RMW, reads and locks an already locked cacheline (e.g., incrementing a lock counter, 2 at T1). Then, at time T2, once both cache-locking RMWs are already committed and have left the ROB, the $\texttt{stul}_1$ of the first cache-locking RMW leaves the SB and writes to L1D. However, since lock chaining is enabled, the cacheline remains locked until the completion of the second cache-locking RMW (e.g., decrementing a lock counter). Finally, at time T3, the $\texttt{stul}_2$ leaves the SB, updating the cacheline in L1D and unlocking it, making it available to other cores. If lock chaining was disabled or the cap of lock chaining was reached before chaining with the second one, the first cache-locking RMW would unlock the cacheline upon writing, making the second cache-locking RMW susceptible to invalidation requests or losing the write permission.

V. An Efficient Hardware Implementation

This section provides hardware implementation details for CLAU, which support all the requirements and features discussed in Section IV. We assume a MESI protocol with a three-level cache hierarchy.

A. Cache-Locking RMWs

CLAU transforms (non-atomic) RMWs into cache-locking RMWs. The cache locking functionality is already available in processors using cache locking for atomic RMWs [14]. The transformation can be easily done at decode time when generating micro-operations, that is, using ldstl and stul micro-operations instead of ldst and st. When a cache-locking RMW is reverted to the baseline behavior, e.g., due to the timeout, store-to-load forwarding, or lock chaining limit, the instruction codes can be reverted (or if a cache locking flag is already available in the LQ and SQ for atomic operations, just set and reset that flag is enough). Cache-locking RMWs are differentiated from atomic RMWs via the instruction op-codes; cache-locking RMWs are non-atomic RMWs with the capability of hardware cache locking.

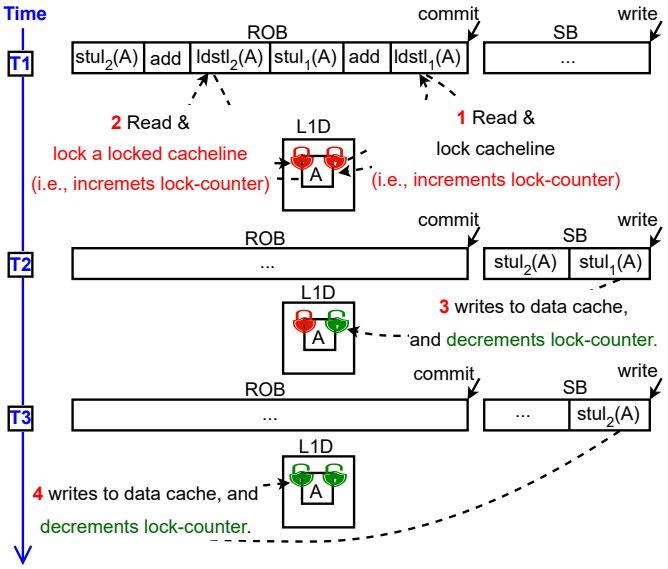


Fig. 3: Lock-chaining mechamism

B. Tracking and Handling Locked Cachelines

We assume that baseline MESI protocol has a locked state per cacheline (needed for atomic operations to support hardware cache locking). Whenever the load micro-op of a cache-locking RMW or atomic RMW (ldstl) executes and reads the cacheline with exclusive permission, the state of the cacheline will be transformed to locked (the locked state is only reached from an exclusive state). Consequently, when the store micro-op of these instructions leaves the SB and writes to the cacheline, it reverts the state to modified. While a cacheline is locked, invalidation or downgrade messages should be stalled. Also, the L1 data cache replacement policy is responsible for skipping locked cachelines as victims of eviction and guaranteeing that at least one cacheline per set is kept unlocked for progressing the non-cache-locking instructions.

C. Watchdog Mechanism

CLAU uses a timer per core that, when triggered, forces all cache-locking RMWs to unlock their cachelines. This timer is enabled when the first cache-locking RMW locks its target cacheline and reset whenever the store micro-op writes to the L1 data cacheline (forward progress of cache-locking RMWs).

D. Handling local squashes and re-execution

Any cache-locking RMW that needs to be squashed (triggered by local events such as branch misprediction) or reexecuted (due to memory dependency violation), must unlock the cacheline. In other words, having the cache locking memory flag set in a re-executed or squashed load operation (i.e., ldstl is being squashed) means it must release the cacheline lock, by changing the cacheline coherence state back to exclusive.

E. Lock-Chaining RMWs

To enable lock chaining we need two capabilities: (i) revoke unlocking responsibility from the store micro-operation of some cache-locking RMWs to hold the cacheline locked for more than one RMW (only the last cache-locking RMW in the chain should unlock the cacheline) and (ii) count how many cache-locking RMWs have executed concurrently to break the chain when the threshold is reached.

Revoking unlocking responsibility relies on the SQ snoop that is performed when executing load instructions to find a possible match for store-to-load forwarding, but narrowing it to report a cacheline address match. That is, a snoop match should be reported if the cacheline address bits match, not if also the offset bits match. The cacheline address match does not need to prioritize the stores depending on their program order regarding the load instruction; we consider all stores (older and younger) as cache-locking RMWs execute speculatively.

The lock-chaining limit mechanism requires augmenting each SQ entry with a saturated counter (chain-length, or CL) that tracks how many cache-locking RMWs have already locked the same cacheline.

Given these two mechanisms, when a load micro-op of a cache-locking RMW executes and snoops the SQ while accessing the L1 data cache, depending on any store micro-operation from an older or younger cache-locking RMW matches the same cacheline address and the value of their CL, the following decisions will be made:

a) the load finds a store in the SQ with no responsibility for unlocking (or no matching store is found). Then the new CL for the store micro-op pairs with the executing load micro-op obtains 0 value, and the load micro-op locks the cacheline.
b) a matching store with unlocking responsibility is found; its CL counter is copied to the counter of store micro-op pairs with executing load micro-op and the new CL is incremented. If the new CL value does not overflow: the matched store no longer has to unlock the cacheline (loses that responsibility), and the store micro-op pairs with executing load micro-op now has that responsibility (must unlock the cacheline). When the counter overflows, nothing is done, effectively not chaining the RMW to the previous already running chain.

It is possible that the store micro-op of a cache-locking RMW faces an already unlocked cacheline as the cache-locking RMW did not join a lock chain (a threashold was reached) and the cacheline was already unlocked. This does not jeopardize the correctness of application as cache locking for non-atomic RMWs is considered a performance optimization and can be revoked at anytime.

F. Memory overhead

Cache locking requires a 13-bit timer, and lock chaining needs augmenting each SQ entry with a 3-bit counter (considering the best-performance option with lock-chaining of 8 RMWs – see Section VII-B). Overall, for an Alderlake-like core implementing a 114-entry SQ, the memory overhead is only 45 bytes.

TABLE I: Gem5 simulated parameters

Component	Parameter
Core	8 cores out-of-order Alderlake-like; Fetch, Decode, Rename width: 8, 6, 6 instructions per cycle; Issue, Commit width: 12, 8 instructions per cycle; ROB: 512 micro-operations; LQ, SQ: 192, 114 entries; RAS: 64 entries; Branch predictor: L-TAGE [29]; Memory-dependency predictor: StoreSet [7]; Processor-prefetch: At-commit store prefetch [30]
Private L1 Cache	Instructions: 32KB, 8-way, 1-cycle hit latency; Data: 48KB, 12-way, 4-cycle hit latency, stride prefetcher [4].
Private L2 Cache	1MB, 16-way, 4-cycle tag and 10 cycle data latency.
L3 Cache	4MB, 16-way, 5-cycles tag and 45-cycle data latency.
Memory	80ns access time.
Coherence	Three-level MESI protocol interconnected with a crossbar [2].

VI. Methodology

We simulate a multi-core processor using the gem5-20 full-system simulator. The processor and memory parameters, trying to resemble an Intel Alderlake core, are shown in Table I. ALU execution latencies are modeled as measured on real hardware by Fog [10]. We integrated a McPAT [19] with Xi et al. [31] bug fixes into gem5 to measure energy consumption using a process technology of 22nm (minimum available in McPAT), a voltage of 0.6V, and the default clock gating scheme for the core. We do not measure energy for the memory controller or interconnection network.

We run parallel applications from different benchmark suits: SPLASH-3 [27], PARSEC 3.0 [6], PHOENIX-2.0 [25] (all with *simmedium* inputs), and write-intensive workloads [12], [18]. We omit four applications from PARSEC and one from PHOENIX that did not finish execution on the baseline gem5 simulator when running with 8 cores. We focus our evaluation on those applications that bear noticeable contention between their non-atomic RMWs (i.e., face at least one pipeline flush initiated by invalidating RMWs per million instructions), but also report the performance of our proposal for all applications (RMW-contention-intensive and less contended). We pinned the threads to the cores to prevent the scheduler from generating an unbalanced load and report statistics for the region of interest (ROI), that is, code after initialization and before output. We account for variability by running applications 20 times starting at a different architectural state. We then compute the average of all simulations.

VII. Evaluation

This section examines the performance improvement and energy savings achieved by CLAU by mitigating the drawbacks of contention in a multi-core processor. In all the ex-

TABLE II: Number of RMWs per one thousand instructions

Benchmark	Value	Benchmark	Value
histogram	183.44	pca	1.05
linear_regression	1.26	barnes	9.28
radiosity	1.65	concurrent_queue	7.16
streamcluster	0.28	freqmine	31.29
matrix_multiply	1.21	TATP	6.86

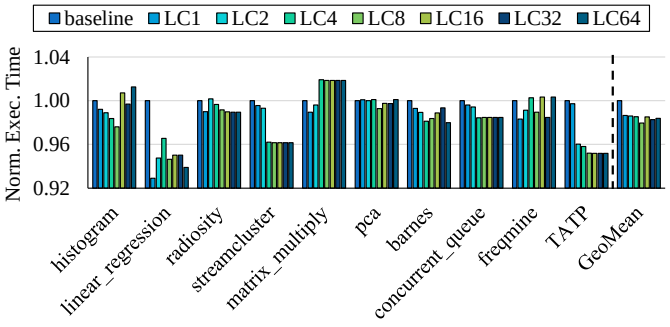


Fig. 4: Sensitivity analysis: Lock chaining

periments, the baseline uses non-atomic RMWs without cache locking, and normalized results are against this configuration.

A. *Frequency of RMWs*

First, we need to measure the number of RMWs that appear in our evaluated applications. When a compiler is configured to generate optimized code (e.g., -O3 in GCC), non-atomic memory updates will consistently use RMW instructions. Table II reports the frequency of the appearance of RMWs (per one thousand instructions) in contented applications. This case study indicates that the number of in-flight RMWs can be substantial in some applications like *histogram*.

B. *Sensitivity analysis*

Cache locking and lock chaining of RMWs rely on some parameters that impact the performance of CLAU. Therefore, a sensitivity analysis, for each of these parameters, is needed to achieve the optimal performance. We tested that each parameter is orthogonal and therefore can be examined independently.

Lock chaining. We begin by studying the optimal length for chaining the lock. The longer the lock chaining goes, the more cache-locking RMWs can execute consecutively without releasing the lock. However, if some of the cache-locking RMWs belong to a mispredicted speculative path that faces a squash, other cores have waited unnecessarily, causing a load imbalance. Therefore, a cap is needed to moderate this trade-off. Based on our analysis (Fig. 4), chaining eight consecutive cache-locking RMWs achieves the best performance improvement for the contented applications. The only exception is *matrix_multiply*; it suffers when increasing the chain length, probably because the memory updates are in the speculative path of the pipeline, facing mispredictions and hence causing a slight load imbalance.

Timeout threshold. The second parameter, timeout threshold, decides when to withdraw the locked cachelines to prevent

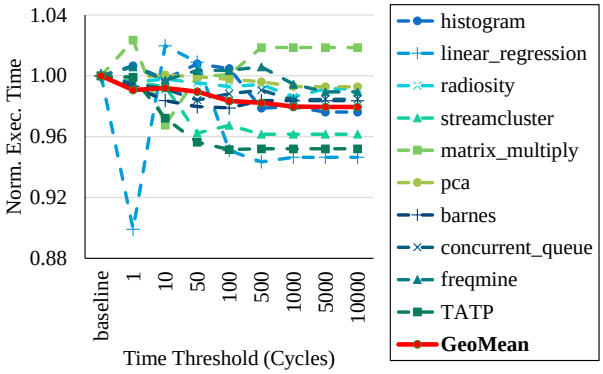


Fig. 5: Sensitivity analysis: Timeout threshold

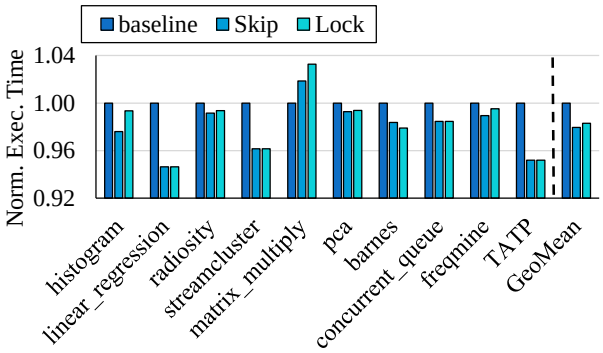
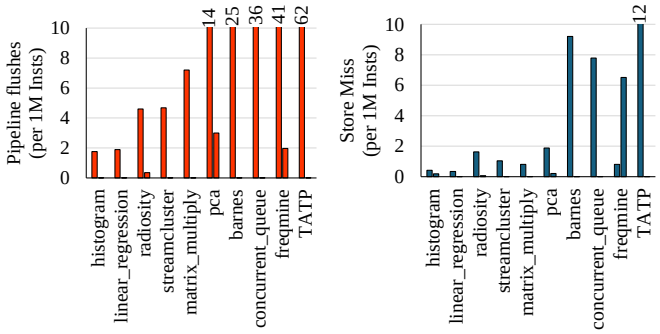


Fig. 6: Sensitivity analysis: Store-to-load forwarding

deadlocks. This parameter depends on the size of the network (including the number of cores) and the contention within applications. Fig. 5 shows that choosing a tiny value (1 cycle) impairs the benefit of cache locking as invalidations will be stalled only for a very short time; therefore having a great chance to invalidate or steal the write permission of cache-locking RMWs. Nevertheless, *linear_regression* is an exception. This comes from having small idle time in cores when setting the threshold to 1 cycle. However, still by increasing the threshold the active cycles of cores, in this application, will reduce compared to 1 cycle. On the other hand, a large number (like 10-kilo cycles) not only increases the memory overhead of hardware implementation but also unnecessarily postpones avoiding a potential deadlock. Hence, a mid-point of 5-kilo cycles is the optimal value for an Alderlake-like core. Similar to the lock chaining analysis, *matrix_multiply* gains better performance when timeout threshold has smaller value. This is because most of its RMWs belong to the mispredicted speculative path, and therefore relinquishing the lock earlier than facing the squash is beneficial for the application as letting other cores progress earlier.

Cache locking on store-to-load forwarding. The last parameter is to decide whether a store-to-load forwarded cache-locking RMW should perform or skip cache locking when it finds the cacheline in L1D with exclusive permission (the prerequisite needed IV-B). In Fig. 6, we observe that a store-to-load forwarded cache-locking RMW benefits in skipping



(a) Pipeline flushes by invalidation (b) Store Miss

Fig. 7: CLAU (2nd bar) reduces the drawbacks of contention in baseline (1st bar)

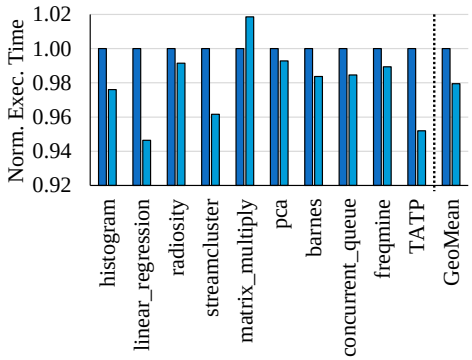


Fig. 8: CLAU (2nd bar) performance improvement with respect to baseline (1st bar)

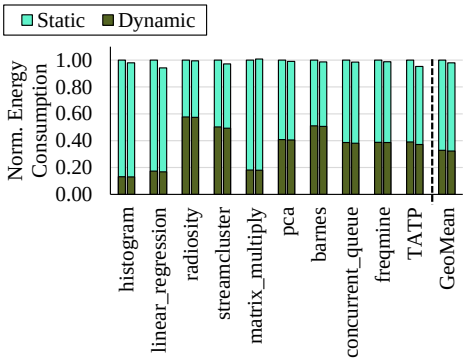


Fig. 9: CLAU (2nd bar) energy improvements with respect to baseline (1st bar)

the cache locking even in finding the cacheline with write permission in L1D. In this figure, the second bar (Skip) shows the case that forwarded cache-locking RMW will skip cache locking. On the contrary, the third bar (Lock) shows the normalized execution time of the case that forwarded cache-locking RMW will perform cache locking which faces a slowdown. In many applications like *histogram*, *matrix_multiply*, *freqmine*, the store-to-load forwarded RMWs belong to the speculative path, which will not be committed, and therefore skipping cache locking will be beneficial in performance as it prevents load imbalance. The only exception is *barnes*, where forwarded RMWs belong to the non-speculative path.

C. Reducing the drawbacks of contention

Fig. 7a shows that cache-locking RMWs can reduce the pipeline flushes induced by invalidations enabling RMWs to complete their execution before relinquishing the cacheline to other cores. This reduction in *TATP* moves from 62 to practically 0 flushes per million instructions, and the maximum flush ratio reaches 3 when using CLAU (in *pca*).

Fig. 7b shows that cache-locking RMWs also reduce store miss to a negligible value by preventing losing the write permission due to contention (invalidation or downgrade message) or replacement policy eviction. The only exception in store misses is *freqmine* which faced an increase compared to baseline. This is because *freqmine* shows lock locality and therefore benefits from large lock chaining like (LC=32 - see VII-B). In this experiment, the lock chaining was limited to 8, and consequently, breaking lock chaining makes some RMWs susceptible to losing write permission again. Nevertheless, considering both aspects (pipeline flushes and store miss), this application still is harmed less by contention when using cache locking.

D. Performance & Energy improvement

The sensitivity analysis identified the optimal parameters for the three discussed variables as follows: limiting lock chaining to eight, setting the timeout threshold to 5K cycles, and skipping cache locking if a cache-locking RMW obtains data through store-to-load forwarding. Using this configuration, we measured the effect of CLAU on the selected benchmarks. Fig. 8 shows an average improvement of 2.05%, for contended applications, over the baseline without any cache locking done by RMW instructions. The maximum performance gains are achieved by *linear regression* (5.36%). The only application faces overhead with lock chaining is *matrix_multiply*, while still benefits from cache-locking RMWs. When considering less contended applications from all evaluated benchmark suites, CLAU improves performance of *string_match* by 1.59%, while *fft* and *lu_ncb* face performance overhead of 2.65%, and 1.79% respectively. This slowdown comes from increased idle time of cores when using CLAU. However, still active cycles of cores are similar between CLAU and the baseline, and CLAU can reduce the squashed rate of RMWs even if it is negligible (less contention). Other less contended applications are unaffected by CLAU. Moreover, in Fig. 9, we observe that the total energy used by the applications is reduced by 2.06% on average (and 5.8% maximum in *linear_regression* similar to performance trend). This reduction comes mainly from static energy, which is directly related to the reduction in execution time. A smaller fraction comes from dynamic energy, attributed to savings from reduced re-execution and squashes. All this considering the small area overhead required by CLAU (45 bytes).

VIII. Related work

Mitigating false sharing. Several proposals try to reduce false sharing as a subtle performance bug of an application [8], [15], [17], [20]. Our paper differs from these proposals in some aspects. First, we try to improve the performance of non-atomic RMWs by reducing their squash rate not only in the case of false sharing but also in the case of true-sharing. Moreover, our paper prevents squashing non-atomic RMWs due to the cache replacement policy in local caches. Second, we propose a transparent hardware solution without compiler/programmer intervention. Third, some false sharing mitigation techniques in related works impose memory overheads relative to the application size. However, our proposal is a pure hardware modification with a cost-efficient implementation independent of the application. Finally, our proposal complies with the x86-TSO memory consistency model and does not alter the cache coherence protocol which might have jeopardized the correctness of the application (a possible scenario in Ghostwriter [15] where for some period a cacheline cannot receive any coherence messages; this is different than stalling an invalidation as when CLAU locks a cacheline guarantees that it will not be updated by any other cores).

Load-load reordering and preventing memory consistency violation. Several proposals allow load-load reordering while preventing memory consistency violation (MCV) in TSO memory model. Writers Block [26], stop invalidation messages while loads are re-ordered, thus avoiding squashes, and allowing loads to commit out-of-order. On the other hand, Zhao et al. present Pinned loads [32], a secure-wise technique that protects loads against MCV by pinning (locking) cachelines that are being loaded until the load is guaranteed to be MCV-free. In contrast, our proposal targets one specific kind of load (part of RMW instructions), and yet mitigating false sharing for them yields reasonable performance improvement. Also, CLAU provides a trade-off between lock locality and thread level parallelism with a simple lock chaining mechanism without jeopardizing correctness (compared to Writers Block [26] that sends uncachable copy of a cacheline to other cores while it is locked in the local core), nor changing the coherence protocol (including directory) opposed to Pinned loads [32]. Lastly, CLAU avoids any possible deadlocks without limiting the execution of RMWs while Pinned loads [32] requires to guarantee enough SB entries for pending stores ahead of a load before executing the load and lock its cacheline.

Efficient implementation of atomic RMWs. Free atomics is a fence-free implementation of *atomic* RMWs that enables out-of-order optimizations (e.g., speculative execution, store-to-load forwarding) for atomic RMWs [3]. However, the following highlights the main differences between this work and the free atomics proposal: 1. store-to-load forwarding is an always built-on feature for *non-atomic* RMWs that is being kept while transforming them to cache-locking RMWs without facing any deadlock or livelock. However, atomic RMWs cannot have store-to-load forwarding in baseline implementation [14] unless implemented as explained in [3]. 2. free-atomics relies on flushing the pipeline to avoid deadlock, but in our proposal, cache-locking RMWs can be freely reverted to baseline implementation (i.e., without cache locking) to prevent any possible deadlock. Therefore, this solution does not impose any performance overhead. 3. even free atomics implementation has some consistency restrictions such as not being allowed to commit until the store buffer is empty. However, cache-locking RMWs can commit freely as before this transformation without having concern regarding memory consistency or any possible deadlock.

IX. Conclusion

Non-atomic Read-Modify-Write instructions are commonly used in DRF regions by CISC architectures to produce more efficient and shorter code. RMW instructions have the advantage of requesting the data with write permissions when reading it, reducing the cost of the store operation as the data should be found with appropriate permissions in L1D. This advantage does not reach full performance if the cacheline is highly contended, as other cores may invalidate the cacheline to perform their own RMW operation.

This paper has proposed CLAU, a micro-architectural mechanism that enables non-atomic RMWs to fetch the cacheline locking it, thus preventing other cores from "stealing" the cacheline while allowing cache-locking RMWs to run concurrently with other instructions in the same core. Our proposal enables concurrent hardware cache locking for multiple non-atomic RMWs while guaranteeing deadlock freedom and no programmer/compiler intervention. We also propose *lock chaining*, a mechanism to allow multiple consecutive memory updates to the same cacheline up to a predefined maximum (to prevent starvation and load imbalance).

CLAU drastically reduces the number of squashes caused by remote cores and significantly increases the hit ratio of the store micro-operation. In our simulation infrastructure, using parallel workloads, this change makes all the pipeline flushes due to the invalidation of RMWs vanish (e.g., from 62 to 0). As a result, we gain performance improvement up to 5.36% (2.05% on average) with a total overhead of just 45 bytes per core.

References

[1] S. V. Adve and H.-J. Boehm, "Memory models: A case for rethinking parallel languages and hardware," *Communications of the ACM*, vol. 53, no. 8, pp. 90–101, Aug. 2010.

[2] 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.

[3] 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.

[4] J.-L. Baer, *Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors*, 1st ed. Cambridge University Press, 2009.

[5] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: A scalable memory allocator for multithreaded applications," *ACM Sigplan Notices*, vol. 35, no. 11, pp. 117–128, 2000.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *17th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2008, pp. 72–81.

[7] 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.

[8] C. DeLozier, A. Eizenberg, S. Hu, G. Pokam, and J. Devietti, "TMI: thread memory isolation for false sharing repair," in *50th Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2017, pp. 639–650.

[9] E. W. Dijkstra, "Hierarchical ordering of sequential processes," *Acta Informatica*, vol. 1, pp. 115–138, 1971.

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

[11] K. Gharachorloo, A. Gupta, and J. Hennessy, "Two techniques to enhance the performance of memory consistency models," in *20th Int'l Conf. on Parallel Processing (ICPP)*, Aug. 1991, pp. 355–364.

[12] 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.

[13] E. J. Gómez-Hernández, J. M. Cebrian, J. R. T. Gil, S. Kaxiras, and A. Ros, "Efficient, distributed, and non-speculative multi-address atomic operations," in *54th Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2021, pp. 337–349.

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

[15] H. Kao, J. S. Miguel, and N. D. E. Jerger, "Ghostwriter: A cache coherence protocol for error-tolerant applications," in *ICPP Workshops 2021: 50th International Conference on Parallel Processing*, Aug. 2021, pp. 12:1–12:10.

[16] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon, "Implementing a cache consistency protocol," in *12th Int'l Symp. on Computer Architecture (ISCA)*, Washington, DC, USA, Jun. 1985, p. 276–283.

[17] T. A. Khan, Y. Zhao, G. Pokam, B. Mozafari, and B. Kasikci, "Huron: hybrid false sharing detection and repair," in *40th Conf. on Programming Language Design and Implementation (PLDI)*, Jun. 2019, pp. 453–468.

[18] 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.

[19] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *42nd Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2009, pp. 469–480.

[20] T. Liu and E. D. Berger, "SHERIFF: precise detection and automatic mitigation of false sharing," in *26th ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, Oct. 2011, pp. 3–18.

[21] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillon, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, C. Escuin, M. Fariborz, A. Farmahini-Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, A. Gutierrez, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, M. Moreto, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, M. D. S. Boris Shingarov, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, W. Wang, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and Éder F. Zulian, "The gem5 simulator: Version 20.0+," *arXiv preprint arXiv:2007.03152*, 2020.

[22] 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.

[23] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design MIPS Edition*, 5th ed., ser. The Morgan Kaufmann Series in Computer Architecture and Design. Oxford, England: Morgan Kaufmann, Sep. 2013.

[24] 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.

[25] C. Ranger, R. Raghuraman, A. Penmetsa, G. R. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," in *13th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2007, pp. 13–24.

[26] A. Ros, T. E. Carlson, M. Alipour, and S. Kaxiras, "Non-speculative load-load reordering in tso," in *44th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2017, pp. 187–200.

[27] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A properly synchronized benchmark suite for contemporary research," in *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2016, pp. 101–111.

[28] 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.

[29] A. Seznec, "The L-TAGE branch predictor," *The Journal of Instruction-Level Parallelism*, vol. 9, pp. 1–13, May 2007.

[30] T.-F. Tsuei and W. Yamamoto, "Queuing simulation model for multiprocessor systems," *IEEE Computer*, vol. 36, no. 2, pp. 58–64, Feb. 2003.

[31] S. Xi, H. Jacobson, P. Bose, G.-Y. Wei, and D. Brooks, "Quantifying sources of error in McPAT and potential impacts on architectural studies," in *21st Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2015, pp. 577–589.

[32] Z. N. Zhao, H. Ji, A. Morrison, D. Marinov, and J. Torrellas, "Pinned loads: Taming speculative loads in secure processors," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 314–328. [Online]. Available: https://doi.org/10.1145/3503222.3507724