Efficient, Distributed, and Non-Speculative Multi-Address Atomic Operations

Eduardo José Gómez-Hernández
eduardojose.gomez@um.es
Computer Engineering Department
University of Murcia
Murcia, Spain

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

Rubén Titos-Gil
rtitos@um.es
Computer Engineering Department
University of Murcia
Murcia, Spain

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

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

ABSTRACT

Critical sections that read, modify, and write (RMW) a small set of addresses are common in parallel applications and concurrent data structures. However, to escape from the intricacies of fine-grained locks, which require reasoning about all possible thread interleavings, programmers often resort to coarse-grained locks to ensure atomicity. This results in atomic protection of a much larger set of potentially conflicting addresses, and, consequently, increased lock contention and unneeded serialization. As many before us have observed, these problems would be solved if only general RMW multi-address atomic operations were available, but current proposals are impractical because of deadlock scenarios that appear due to resource limitations. Alternatively, transactional memory can detect conflicts at run-time aiming to maximize concurrency, but it has significant overheads in highly-contended critical sections.

In this work, we propose multi-address atomic operations (MAD atomics). MAD atomics achieve complexity-effective, non-speculative, non-deadlocking, fine-grained locking for multiple addresses, relying solely on the coherence protocol and a predetermined locking order. Unlike prior works, MAD atomics address the challenge of enabling atomic modification over a set of cachelines with arbitrary addresses, simultaneously locking all of them while side-stepping deadlock. MAD atomics only require a small storage per core (around 68 bytes), while significantly outperforming typical lock implementations. Indeed, our evaluation using gem5-20 shows that MAD atomics can improve performance by up to 18× (3.4×, on average, for the applications and concurrent data structures evaluated in this work) over a baseline implemented with locks running on 16 cores. More importantly, the improvement still reaches 2.7×, on average, compared to an Intel hardware transactional memory implementation running on 16 cores.

CCS CONCEPTS

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

KEYWORDS

Multi-core architectures, synchronization, critical sections, atomicity, multi-address atomics

ACM Reference Format:
Eduardo José Gómez-Hernández, Juan M. Cebrian, Rubén Titos-Gil, Stefanos Kaxiras, and Alberto Ros. 2021. Efficient, Distributed, and Non-Speculative Multi-Address Atomic Operations. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3466752.3480073

1 INTRODUCTION

The current trend of increasing the number of cores in the same package potentially increases the performance of parallel applications and concurrent data structures. Unfortunately, executing a larger number of threads causes more synchronization overhead, which results in less efficient algorithms when relying on locks [9, 14].

Mutexes,[1] while fast when there is little contention, exhibit a large overhead for contended locks. In some implementations they even require the aid of the operating system. A common use of mutexes is to conservatively protect a critical section with a single lock as *coarse-grained mutexes*. Many times, this strategy scales poorly. The underlying reason is that execution is serialized, *even when the sets of addresses accessed by two critical sections guarded by the same lock are disjoint.* On the other hand, *fine-grained mutexes* aim to protect only the specific set of critical data of a critical

[1] MUTually EXclusive: An object in a program that serves as a lock, used to negotiate mutual exclusion among threads.

MICRO '21, October 18–22, 2021, Virtual Event, Greece

ACM ISBN 978-1-4503-8557-2/21/10…$15.00
https://doi.org/10.1145/3466752.3480073

section. Unfortunately, fine-grained locking is not an easy task for anything other than small critical sections. Moreover, a fine-grained locking scheme may result in the need to acquire more than one mutex to perform some task. This introduces the overhead of multiple mutexes in contrast to coarse-grained locking, and more importantly, there is a risk of deadlock if locks are not acquired in a global order.

An alternative that has long been considered as more scalable than mutex locks is the concept of *non-blocking algorithms.* Non-blocking algorithms rely on directly using atomic read-modify-write (RMW) primitives natively provided by the hardware. Commonly, a compare-and-swap (CAS) instruction checks whether the condition for the atomic execution of an operation holds and performs some action, storing the result. Although non-blocking algorithms have been proposed for several data structures [47], writing correct non-blocking algorithms that guarantee system-wide progress (lock-free algorithms) is a notoriously difficult task [46].

A third synchronization alternative appeared more recently in commodity processors, as they adopted hardware transactional memory (HTM) [19, 25]. With HTM, critical regions (*transactions*) are executed concurrently in a speculative manner, while the hardware monitors their memory accesses and rollbacks execution if conflicts arise. In existing best-effort HTM implementations, a transaction is typically re-executed up to a number of times before taking the non-speculative path, in which mutual exclusion is enforced through a global lock [51]. The downside of HTM is that it scales poorly for contended critical sections due to high abort ratios and frequent serialization of threads on the fallback lock.

The three previous alternatives rely on atomic RMW operations implemented in hardware. High-performance implementations of atomic RMW operations *lock* the target cacheline from the time the data is read from the memory subsystem until it is written to the memory subsystem [27, 42]. Atomic RMW instructions are decomposed in several micro-ops, including a load, an arithmetic instruction, and a store [3]. The load asks for the target cacheline with read-write permissions. On arrival to the private cache, the cacheline is locked and the required value is read. A new value can then be computed. The ensuing store writes the value into the cacheline and unlocks it. While the cachelines are locked, invalidation messages from remote cores and write requests from the local core cannot proceed.

Atomic RMW instructions are the most efficient way to perform an atomic update of a variable, since they are genuinely hardware operations. Compared to mutex schemes, they minimize serialization and require no OS intervention. Compared to HTM, they have less overheads and are faster when facing contention, as stalling is more efficient than squashing. However, the implementation of *multi-address* atomic operations is a challenge we have yet to overcome.

Our goal is to exploit the same *cacheline locking* mechanism used for atomic RMW operations *to lock every address accessed in a critical section.* This way we can replace a number of critical sections protected by mutex locks or transactions with a much faster ***multi-address (MAD) atomic operation***. These new MAD atomic operations run at cacheline fine-grain level, removing false contention and allowing concurrent access to disjoint data. Similar to other complex instructions, MAD instructions are translated into multiple micro-ops at the decode stage, including locking and unlocking the addresses accessed within the critical section.

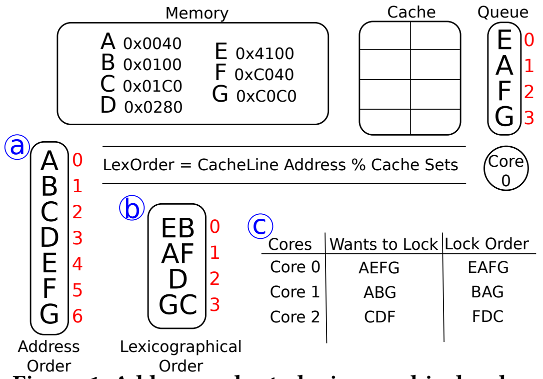
While prior work proposed multi-address atomicity in several forms [35, 38, 39, 44], they did not consider several *deadlock* scenarios when *distributively* locking several cachelines, nor provide a safe, *non-speculative*, solution. Deadlocks do not appear in current implementations of atomic primitives because they only lock one cacheline at a time. In addition, we do not rely on any kind of centralized hub for managing atomicity, neither on a distributed but transactional method, i.e., try, but roll back on a conflict. There are two reasons why we want to follow a non-centralized, non-transactional approach. First, centralizing the management of atomicity creates a bottleneck that restricts scalability with respect to how many cores can execute *independent, non-conflicting* operations concurrently. Second, trying and rolling-back becomes very expensive in scenarios of contention: conflicting operations, haphazardly fighting it out, can expend considerably more effort (time and traffic) than what they would need if they followed an orderly execution.

We evaluate MAD atomics on the cycle-accurate, full-system gem5-20 [34] multicore simulator and show how MAD atomics affect the performance and instruction count of commonly used data structures accessed in parallel. Performance is improved up to 18× (averaging 3.4× running on 16 cores) over a baseline implemented with locks running on 16 cores. Compared to an Intel-like hardware transactional memory implementation (TSX), the performance benefit reaches 2.7× on average. MAD atomics also introduce a reduction in the number of instructions executed. Indeed, MAD atomics only execute 10% of the original instruction count, on average, for 16 cores, compared to our baseline, reaching less than 5% on 64 cores. This means that MAD atomics are not only faster than TSX but also more energy efficient.

2 BACKGROUND

Dijkstra first presented the problem of taking several locks, giving an analogy of five philosophers and a table set with five forks [12]. His solution was based on a predetermined order for acquiring the locks. The predetermined-order approach eliminates deadlocks for acquiring any number of mutex locks since software resources are basically unlimited: if locks are taken in order, no deadlock is possible. However, when locking cachelines in the local caches of the cores, resource limitations in the system, such as, for example, cache associativity, jeopardize deadlock freedom. The problem is exacerbated when considering that multi-address atomics cannot create custom atomic groups at their convenience, since *all* of the requested addresses must be locked as a single atomic group. A detailed analysis of each resource deadlock is provided in Section 4.

Ros and Kaxiras [41], inspired by the theoretical foundation set by Coffman et al. [10], address the resource limitation problem by assigning a global *sub-address* order (called *lex order*) to each cacheline. The lex order is dictated by a set of bits from the cacheline address that are used to map it on a resource, e.g., the bits used to index the cache set where the line is placed. Two lines with the same lex order compete for the same resource in private or shared cache structures and directories. This is termed a *lex conflict.* On a lex conflict, the contended resource will serve as an ordering point



Cores	Wants to Lock	Lock Order
Core 0	AEFG	EAFG
Core 1	ABG	BAG
Core 2	CDF	FDC

Figure 1: Address order to lexicographical order

for the conflicting requests, allowing the core which first grabs the resource to continue while all other cores will have to wait.

Figure 1 depicts how the lex order compares to the address order. Solutions like Dijkstra, follow the address order to establish the locking address (Figure 1-a). Meanwhile lex order (Figure 1-b) takes into account the size of the smallest shared cache and the cacheline address (removing the offset bits inside the cacheline). Using the lex order, Figure 1-c shows an example of three cores locking different sets of addresses and how are they reordered.

Because of lex order, waiting cores cannot form a cycle with the cachelines that they have already locked and the lines that they want to grab next. Lex order guarantees that conflicts will occur in the minimum common lex order between any atomic group. Thus, resource deadlocks are avoided.

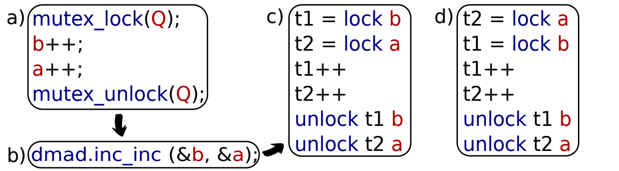
Our solution is also based on lex ordering. However, for private caches, Ros and Kaxiras resolve lex conflicts by not allowing two cachelines with the same lex order in the same atomic group. This is not a possibility when implementing multi-address atomic operations, as if the program dictates that two accesses need to be performed atomically, the hardware cannot impose an atomicity restriction, and *should be able* to hold those locks at the same time. Therefore, what we need is a non-speculative, non-deadlocking, fine-grained locking for multiple addresses.

3 MAD ATOMICS

MAD atomics are a set of individual instructions able to atomically update a small number of memory locations. Ideally, we want to encapsulate a code section, such as the one shown in Figure 2-a, into a single hardware primitive (Figure 2-b). The instruction will be decomposed in several micro-ops (Figure 2-c), which may acquire the locks in a different order with the purpose of avoiding deadlocks (Figure 2-d).

Encapsulation and lock acquisition reorder also apply to transactions, or any other *ad hoc* synchronization construction. MAD atomics can coexist in a program with other larger critical sections using locks/transactions, thus offering flexibility to the programmer.

Similarly, Figure 3 is an example of how a two-address fetch_and _add MAD atomic translates into micro-ops (syntax adapted from the gem5-20 simulator micro-code). The micro-ops are divided in three blocks: locking (lines 3-4), computing (lines 6-7), and storing (lines 9-10). The entire atomic instruction is guarded by two



```
mutex_lock(Q);
b++;
a++;
mutex_unlock(Q);
```

```
dmad.inc_inc (&b, &a);
```

```
t1 = lock b
t2 = lock a
t1++
t2++
unlock t1 b
unlock t2 a
```

```
t2 = lock a
t1 = lock b
t1++
t2++
unlock t1 b
unlock t2 a
```

Figure 2: a) Critical section with mutex locks; b) MAD atomic instruction; c) micro-ops generated by the MAD atomic; d) run-time order of instructions.

mfence micro-ops as it is the case for single-address atomics.[2] In the locking block, we find the *load_lock* micro-op, and a variation: *load_lock.exec*. The *load_lock* micro-op informs the core about an address to lock, however, the address is not locked immediately. Data will be loaded when it is eventually locked. The *load_lock.exec* variation provides an address to lock and load the data, but it also notifies the core that it is the *last* address of the multi-address atomic operation to lock. This means all locks can now become effective, and data loaded to each of the corresponding registers. *Add* will add the value stored in the registers. Subsequently, the *store_unlock* writes the updated values to the locked cacheline, unlocking it after the write.

MAD atomics use a single macro-instruction, decoded at runtime into several micro-ops. The new *lock* micro-ops proposed in this work are implemented similarly to a load that requests exclusive permissions (i.e., the load micro-op in a x86 atomic operation). If the target memory page is not writable, a page fault is triggered in order to obtain write permissions. Then the macro-ops simply re-starts when the fault is resolved. As MAD atomics are implemented with a single instruction, they cannot be interrupted during execution, and context switches need to wait until the MAD atomic operation completes and retires. Interrupts may therefore be delayed. However, since there is a progress guarantee and the critical sections that MAD atomics target are short, waiting times are not extremely long.

3.1 The Lexical reOrder Unit

Cachelines involved in a multi-address atomic operation need to take the lock in a predetermined order to avoid deadlocks. The unit responsible for tracking the cachelines and acquiring the locks is the *Lexical reOrder Unit* (LexOU).

When a *lock* micro-op is executed, the target cacheline is not directly issued to the memory subsystem. Instead, cacheline addresses are stored in a small buffer implemented as a content-addressable memory (CAM) named the *Lock Queue*. Cachelines are not locked until all locks are inserted in the *Lock Queue*. Different locks can target the same cacheline. In this case, a single cacheline address is stored in the *Lock Queue* and a counter stored along with the cacheline address indicates the number of lock operations performed on that cacheline. The counter size is $log_2(maxLocksPerCPU)$ bits.

The counter is used to unlock the cacheline after all the stores writing to the cacheline have completed their writes. We unlock cachelines using the same micro-op used in x86 atomics (`stul` in gem5-20 terminology). In case of single-address atomics the `stul` micro-op will find the *Lock Queue* with a single address (the

[2] As specified in by Intel [29] "The processor waits until all previous instructions have been completed and all buffered writes have been drained to memory before executing the serializing instruction".

```
mfence
// Load-Locking block
load_lock t1, rax
load_lock.exec t2, rcx
// Computing block
add t1, t1, reg
add t2, t2, regm
// Storing-Unlocking block
store_unlock t1, rax
store_unlock t2, rcx
mfence
```

Figure 3: A two-address fetch_and_add atomic operation in gem5-like micro-code (irrelevant parts have been omitted)

value of the lock counter is always one), it will decrement the lock counter, and unlock the cacheline after performing the write. In MAD atomics, the only difference is that the lock counter can have a value greater than one. In that case, the cache line is unlocked after writing only if the value of the counter was one before the write.

In addition to the cacheline address and the lock counter, each entry in the *Lock Queue* includes three more bits: a collision bit to indicate if the entry has the same lex order as another entry, a hit bit indicating the presence (with write permission) of the cacheline in the private cache, and a lock bit that indicates if the cacheline is already locked.

3.2 Use Case: MCAS Instructions

Once implemented MAD atomics, we can easily derive multi-address compare_and_swap (MCAS) instructions.

Our implementation of the MCAS instruction receives the following registers as input: *rax*, *rdx*, *rcx*, *rsi*, *rdi*, *r8*, *r9*, *r10*, *r11*, *r12*, *r13*, and *r14*, in that specific order. Each set of three registers represent a CAS operation. The final number of registers used depends on how many CAS operations need to be done atomically (six for MCAS of arity two, nine for MCAS of arity three, and so on). For example, MCAS of arity two follows this format: *dmad_cas(*addr0, old0, new0, *addr1, old1, new1)*, where add0 is stored in rax, old0 in rdx, etc. Once decoded, it will be translated into a sequence of micro-ops similar to Figure 2-c.

Similarly to the more general multi-address atomic instructions, memory fences are implicit at the beginning and end of the MCAS instruction to avoid load→store reordering. Subsequently, it executes lock micro-ops for the set of addresses and waits until all corresponding cachelines are locked. Then, values are safely read, the conditions are evaluated, and the new values are written if all conditions are fulfilled. Finally, the instruction sets the *rax* register to 0 if any of the conditions fail, or 1 if all of them were successful.

3.3 Other Considerations

Arity of MAD atomics. The maximum number of addresses to be conditionally updated in the same instruction is limited by the number of logical registers. MAD operations can require up to three registers (target address, old value, and new value) per address. Since X86_64 has up to 16 registers available for the programmer [29], the maximum arity to implement MADs without requiring additional instructions and hardware to bypass the register restriction is five.

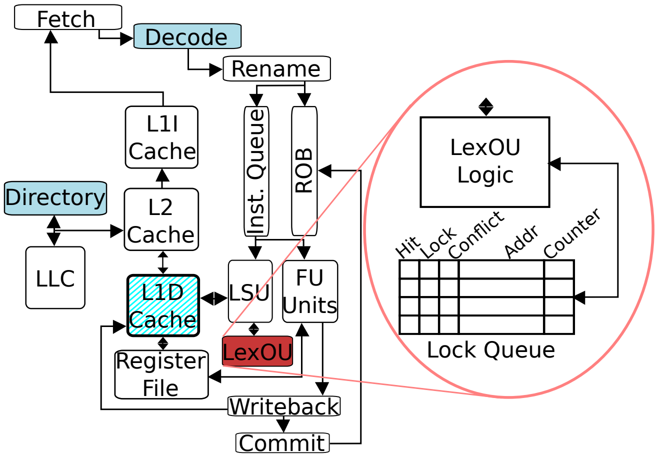


Figure 4: Microarchitectural overview. Modified units appear in blue. New structures appear in red.

Microarchitectural changes. Figure 4 highlights the changes performed to implement MAD atomics. The decode stage include the addition of MAD instructions and their corresponding micro instructions. The L1D cache needs to keep track of all locked cachelines. Depending on the actual single-address lock hardware implementation, the L1D may require modifications or not. If a bit per cache entry is used no modifications are required. Otherwise, if a single register is used to track the locked cache line, this register is extended to an array of registers. The replacement policy is also modified to never select for eviction locked cachelines. Finally, as discussed in the next section, each set of the directory will require an extra bit in order to prevent deadlocks.

Memory requirements. Each *Lock Queue* entry stores three bits for its current state (conflict, private cache hit, and lock), the lock counter ($log_2(maxLocksPerCPU)$ bits), and 16 bytes for the memory packet of the address to load and lock. This gives a total of $3 + (16 * 8) + log_2(maxLocksPerCPU)$ bits per *Lock Queue* entry. Since our current implementation supports up to four addresses (see Section 7 regarding this limitation), the memory requirements of the *LexOU* is 532 bits, i.e., less than 68 bytes.

Simultaneous Multithreading (SMT). In SMT implementations, the LexOU is not replicated nor partitioned. SMT threads require mutual exclusion to prevent deadlocks: only one thread can use the LexOU at a time. A new register storing the SMT core using the LexOU is added. A lock instruction from a different thread that the one using the LexOU stalls until the Lock Queue is empty.

Note that an SMT core must not have access to a value locked by another SMT core. When reading the value, if the cacheline is locked, it will check which SMT core is allowed to lock in the LexOU, and only the one that matches, can read or write into the cacheline. This is not a restriction of MAD atomics but of atomic operations in general.

4 RESOURCE LIMITATIONS

This section discusses all possible resource-related deadlocks and how MAD atomics handles them.

4.1 Private caches

The key idea of our approach is that we can lock *exclusive* cachelines *locally in the private caches, usually without communicating with the directories.*[3] If the addresses we wish to atomically modify are all present in the private cache (*local*), there is no communication required. *Only* when we are missing one or more of the addresses we need to consider the interaction with shared caches and the directories (which is discussed in the following sections). The reason is that we avoid the centralization of the decision making for the locking, which is one of our main goals.

Therefore, a necessary property for the private cache is that it must be able to simultaneously hold all the cachelines that will be locked during a multi-address atomic operation. Otherwise, a deadlock is possible. As we mentioned, locking is strictly local: no other core, nor the directory, is usually informed when a cacheline is locked. Because of this, other cores may see their invalidation or forwarding message delayed, as they cannot receive a reply until the lock is released.

However, (unless the cache is fully-associative) cachelines are not freely distributed in the private cache, but mapped to a particular set in the cache. For example, if three cachelines are mapped on the same set, but the cache only has two ways, after locking two out of three, there is no more space in the set to hold the third. This leads to a resource-limitation deadlock because we cannot release any of the previously held locks, and the multi-address atomic instruction cannot progress.

The resource limitations of the private cache is a straightforward problem to solve since we target multi-address atomic operation with low *arity*. **The rule for private caches is that the arity of the atomic operation must be lower or equal than the private cache associativity.** Since our current implementation of MAD atomics supports up to four addresses (see Section 7), this is compatible with most commodity processors. For example, the ARM Cortex-A78 [6] has four ways, Intel's Skylake [18] has eight ways and Intel's Icelake [1] has twelve ways for the L1 data cache.

In contrast to the previous state-of-the-art solution for the private caches [41], we do not limit the sub-address lex order to the number of entries in the private cache (associativity×sets). Furthermore, the practice of defining a sub-address lex order as associativity×sets of the private cache has the undesired effect of restricting what can go into any one set without causing a *lex conflict.* In particular, to avoid a lex conflict, the block addresses that are mapped on the same set *must differ in the bits that correspond to the associativity in the lex order*, typically the $\log_2$(associativity) bits that follow the cache index bits of the address. Our approach does not impose this limitation and, thus, is more flexible.

Lastly, in energy efficient systems that use direct-mapped private caches, no multi-address atomic operations could be implemented following the associativity rule. Fortunately, this limitation can be resolved with the use of fully associative victim caches [31]. In this case, the maximum arity of our MAD atomics is the associativity of the victim cache, usually larger than our maximum arity. Note that locked cachelines cannot be evicted (unless they are released), and they are excluded from the replacement policy (e.g., LRU). Similarly, when relying on victim caches, locked cachelines cannot be evicted from the victim cache until their lock is released.

4.2 Directories and shared data caches

Without limiting the sub-address order to the number of entries of the private cache (and at the same time without imposing any restriction on what block addresses can go into the same set in the private cache) we are free to select a lex order that best suits the shared cache and the directory. We will first discuss the directory which is critical in allowing the locks to be held in the private caches (even though the directory is unaware of this locking) and then expand on the shared caches. If the shared cache (commonly the last level cache on chip) enforces inclusion with the private caches, the shared cache becomes another limited resource. Since shared caches usually have a larger number of entries than a directory (or at least an equal number when the cache includes directory information), in the following paragraphs we focus our reasoning on the directory. However, our solution also applies to inclusive shared caches.

Cachelines in the private caches are tracked by the directory. This brings an inclusive property between private caches and directories: if a cacheline is stored in a private cache, there must exist an entry in the directory for such a cacheline. In other words, to be able to hold and lock a number of cachelines, the directory should dedicate the corresponding entries. Therefore, limited resources at the directory can cause deadlocks too.

Assume now, for the sake of example, that all block addresses that we wish to lock in a private cache are mapped on the same directory set. Furthermore, as we are not bound by the private caches on which lex order to use, we pick the lex order to be equal to the number of sets in the directory (i.e., the lex order is defined by the directory index bits of the address). If the private cache already has all the cachelines, this means that the corresponding directory entries are already present in the directory and all is well. Any attempt to evict one of the corresponding directory entries will get stalled trying to invalidate the locked cacheline in the private cache until the lock is released.

However, if the private cache does not already have all the cachelines that we wish to lock, we must ask for the missing ones from the directory. Assuming that the directory has at least the same associativity as the private cache (a valid assumption for the majority of systems), the directory can easily accommodate any combination of addresses that we wish to lock. However, this ignores the case where another private cache has silently locked a number of cachelines in a directory set. Recall that one of the basic premises of our work is that we do not ask for permission to lock when we have a private cacheline as exclusive.

The problem now is that the apparent associativity of the directory with respect to a private cache wanting to lock a number of entries in a set, has been reduced through the actions of another private cache. Furthermore, since we chose the lex order to be equal to the number of sets in the directory, the addresses that map in the same directory set have the same lex order rank, and conflict with each other (in lex order).

An example is shown in Figure 5. Cores 0 and 1 are trying to lock a set of cachelines, represented with a letter and a number of

[3] We assume distributed directories in the general case, so different addresses may be handled by different directories.

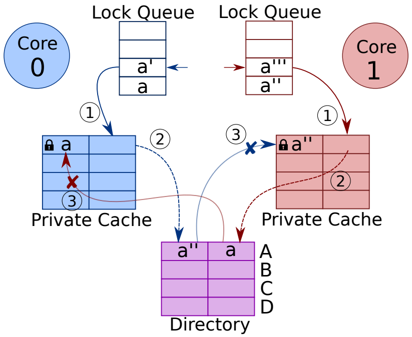


Figure 5: Deadlock due to limited directory resources

apostrophes and placed in lex order in a *Lock Queue*. The letter indicates the set in the directory where the cacheline is indexed. The apostrophes differentiate cachelines placed in the same set. Both cores start locking the first cacheline in lex order (directory set order): a for core 0 and a'' for core 1. Because both cachelines map in the same directory set, the A enties are exhausted, as the example concerns a 2-way directory. The second cacheline to be locked by each core will need to evict the cachelines already tracked by the directory, which will trigger an invalidation. Recall that directories are not aware of the locked status of the cachelines in the private caches. Invalidation messages, however, will be retained at the private cache controller, as the cacheline is locked. Therefore, neither of the cores can lock all the cachelines needed by its corresponding multi-address atomic operation, and the first locked cacheline of each core will never be freed. This is a deadlock.

The Ros and Kaxiras solution to avoid deadlocks due to limited directory resources is similar to their solution for a private cache [41]. The sub-address order (associativity × sets) has to be chosen considering the smallest structure in the system, which, in the depicted figure, is the directory. This means that addresses in a set are ranked in lex order (so two entries of the same rank will conflict). Effectively, ordering the directory entries within the sets increases the number of conflicts, and will cause two cores to serialize in the same directory entry (even though they could both fit in the same set). However, as in the private cache case, this restricts the addresses that can be used in the same multi-address atomic operation, a limitation that we cannot accept.

To illustrate the risk of deadlock stemming from directory resources, consider the following two scenarios:

First, let us assume for now that every multi-address atomic operation is composed of non-conflicting addresses in the chosen lex order (number of sets in the directory). This means that the operation will need at most one entry in each involved directory set (note that such entries may already exist in the directory, or may be requested). No matter how many cores perform multi-address atomics concurrently, deadlocks are not possible because the lex order guarantees that any two conflicting multi-address atomic operations will conflict in their minimum common lex order. This means that a cyclic dependency among concurrent multi-address atomic operations can never be formed as long as each individual operation is conflict-free in the lex order.

Second, let us consider multi-address atomic operations whose addresses conflict with the chosen lex order—in other words, each atomic operation needs more than one entry from a directory set. When several such multi-address atomic operations run concurrently, if at least one wishes to lock multiple entries in the same directory set, deadlock is possible as there might not be enough directory associativity to accommodate all lex conflicts. Note that there is no risk of deadlock if a single atomic operation of such kind were to be in progress across the entire system, since the associativity of the directory (being equal to or larger than the associativity of the private cache) would guarantee by itself that all the lex conflicts in a directory set are accommodated (the number of lex conflicts is limited by the associativity of the private cache).

Our approach to prevent the aforementioned kind of deadlocks consists in conservatively locking directory resources in those particular circumstances: **when a multi-address atomic operation wishes to lock more than one entry of a directory set, the whole set must be locked**. We assume that each directory set has a lock bit that can be set upon a request reaching the directory set.

If a multi-address atomic communicates with the directory to ask for a missing cacheline (or request exclusivity for a cacheline) it declares whether it *wants to lock* (or *has already locked*) one or more cachelines of the same set. If the multi-address atomic operates on just a single entry out of a directory set, it does not have to lock the set but has to wait in case someone else has locked the set. If, on the other hand, the multi-address atomic operates on multiple entries out of a directory set, it has to lock the whole set and, of course, has to wait in case the set is already locked.

A multi-address atomic releases the lock it has on a set only after it manages to install all of its entries in the set (hence, multi-address atomics that want to install only one entry in a set do not lock it, as we assume that requests to that set are handled by the directory controller in a FIFO manner.) Note that some of these entries may correspond to cachelines already in its private cache and some entries may need to be installed in the directory set. While a multi-address atomic holds the lock of the set, it will experience no interference from other cores for this set. Note also that, despite having a lock on a set, a multi-address atomic may have to wait for other multi-address atomics that have silently locked cachelines in the set. However, based on the principle of acquiring all the required entries before releasing the lock, it is impossible to have a deadlock with another atomic.

4.3 Eviction buffers

An eviction buffer is a temporary storage for a cacheline being evicted from cache. This allows the immediate use of the cache entry without having to wait for eviction-induced acknowledgments. Hence, cache evictions are performed out of the critical path of a cache miss. When a cacheline is not silently evicted (i.e., requires performing coherence actions), the cacheline remains in the eviction buffer until the coherence actions finish. When the eviction buffers are full, cachelines cannot be evicted until a new entry is freed. That is, the eviction latency is not hidden.

Eviction buffers of private caches do not impose risk of deadlock due to resource limitations as locked cachelines cannot be evicted. If the eviction buffer is filled with non-locked cachelines, subsequent cache misses will wait until an entry in the eviction buffer is free.

However, eviction buffers for shared caches can hold cachelines locked in private caches. This gives rise to the problem depicted in

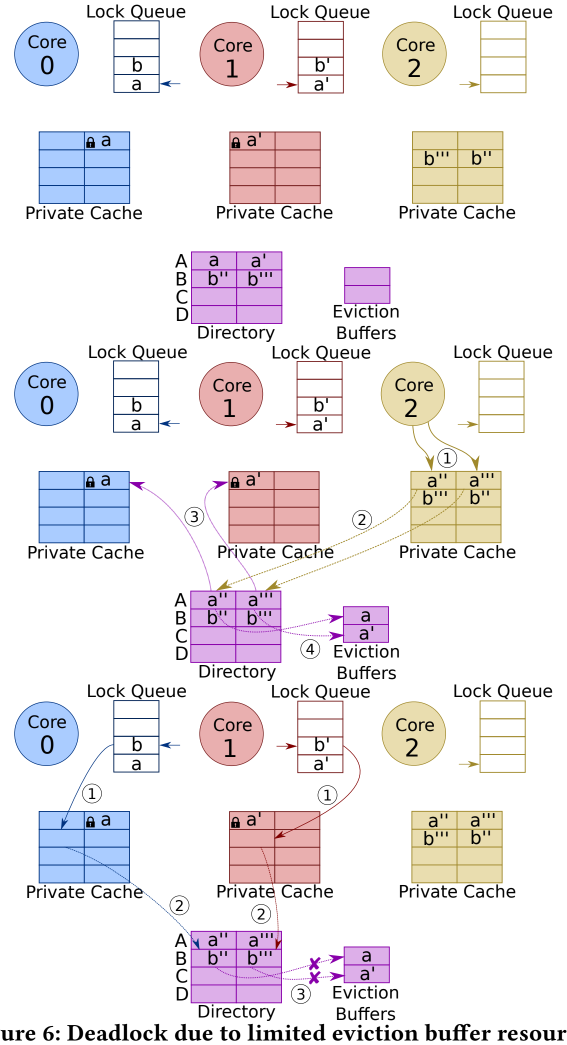


Figure 6: Deadlock due to limited eviction buffer resources

Figure 6. The figure shows a deadlock caused by limited resources in the directory eviction buffer. For the sake of clarity shared cache and directory are combined in the example. The eviction buffer consists of just two entries. Core 2 causes two evictions in the directory, filling the eviction buffer entries. For the evicted entries to leave the eviction buffer, the tracked cachelines need to be invalidated at the private caches. However, the cachelines are locked by Core 0 and Core 1, which are in the middle of a multi-address atomic operation. These cores cannot finish locking the required cachelines because the directory is not able to evict any cacheline, and therefore cannot leave space for new cachelines to be tracked (no matter the set where they are indexed). A cycle is formed in this case, causing a deadlock. It is important to note that although with a small number of cores the total number of locked cache lines in the system may be smaller than the capacity of an eviction buffer, as the number of cores in the system increases, this deadlock becomes increasingly probable.

This problem was previously described in the WritersBlock protocol [40]. In such a protocol the deadlock-free condition is to ensure that loads can always progress. The solution proposed to this problem is to make the read requests un-cacheable, and therefore not requiring directory tracking. However, when applying this concept to multi-address atomic operations, the locked cachelines need to reside in the private cache, requiring directory tracking. As far as we know, there is no published solution to this problem suitable for multi-address atomics.

4.3.1 Addressing eviction buffer exhaustion. Eviction buffers holding locked cachelines could prevent other cachelines from being evicted from a cache or directory. Locked cachelines delay any invalidation message until they are unlocked in private caches. They can be only unlocked when the atomic operation from the lock owner completes.

Increasing the number of eviction buffer entries up to $numCores * (maxLocksPerCore - 1) + 1$ would solve the deadlock, as at least one core would finish. Yet, increasing eviction buffer size or denying a core to perform locks through an arbiter are non-scalable and very restrictive solutions.

To prevent this, we perform *in-situ* directory replacements for evictions of private entries (i.e., without using the eviction buffer). We start from the observation that the directory contains information about the current private/shared status of the cached entries. Shared cache lines cannot be locked at a L1 cache, since locking requires exclusive permission. Therefore, evictions of shared entries can be performed as usual: they are immediately moved to the eviction buffer and issue invalidation messages to L1; when the eviction buffer collects all the acknowledgments the eviction completes and the entry can be released.

On the other hand, private entries may be silently locked at L1 caches, potentially causing a deadlock. ***The in-situ* replacement strategy allows directory evictions of private entries to proceed without using the eviction buffer**. Therefore, private entries that need to be evicted remain in the directory while sending the single invalidation message to the private cache. On the arrival of the acknowledgment, the eviction is completed and the entry can be freed, leaving space for the incoming cacheline. Note that although *in-situ* replacements increase the latency of the requests that do not find an entry in the directory, this is an infrequent scenario.

5 METHODOLOGY

5.1 Simulation Environment

We simulate a multicore processor using the gem5-20 [34] full-system simulator. The simulated system runs Ubuntu 16.04 with Linux kernel 4.9.4. The processor parameters, mimicking a Intel Skylake processor, are shown in Table 1. We use Ruby to model the memory hierarchy and the coherence protocol. Execution and issue latencies are modeled as measured on real hardware by Fog [17].

Apart from our baseline system, we also model an Intel TSX-like best-effort HTM system, based on the transactional memory support made available in gem5-20.1 for ARM TME. Because best-effort HTM systems typically employ the L1 data cache for speculative buffering, capacity-induced aborts are not an issue given the small size of the critical sections considered in this work; thus, virtualized HTM systems are expected to provide little or no additional performance gains. The conflict resolution policy of our TSX model

Table 1: System Configuration

Processor	
Core count	1,2,4,8,16,32,64
Issue width	4 instructions
Commit width	8 instructions
Instruction queue	128 entries
Reorder buffer	224 entries
Load queue	72 entries
Store queue + store buffer	56 entries
Memory	
Private L1 I&D caches	32K/core, 8 ways
Access latency	1 cycle
Private L2 cache	256K/core, 8 ways
Access latency	4 cycles tags, 10 cycles data
Shared L3 cache	32M, 16 ways
Access latency	5 cycles tags, 45 cycles data
Directory	32768 sets, 16 ways
Memory access time	80ns
Network topology	Crossbar

follows the requester-wins policy as seen in current commercial implementations by Intel, which can result in temporary livelocks that are resolved by falling back to mutual exclusion.

5.2 Benchmarks

We evaluate our proposal using a set of benchmarks whose critical sections can be translated to two categories of MAD atomics: multi-address atomic operations and multi-address compare-and-swap (MCAS) operations. We gather statistics for the region of interest of the benchmarks, namely *after* the initialization phase of the application and before any final output. The simulated benchmarks are run to completion.

5.2.1 ***Multi-address atomic operations***. Some applications include critical sections that modify a small number of memory locations atomically. Replacing lock/HTM protection of such critical sections with a single multi-address atomic avoids lock acquisition times, removes retry loops or aborts, and allows all cores to proceed in parallel (if no conflicts are found).

Bitcoin is an application that performs bitcoin payment transactions [32]. It emulates operations often made in the bitcoin network over a set of bitcoin wallets. The application has a critical section where a given amount is withdrawn from one wallet and deposited into another. We replaced it with two multi-address atomics, similar to fetch_and_add, which add opposite amounts to each wallet. The baseline version has a single lock for the entire transaction process. The lock-free version of Bitcoin uses a different locking scheme that locks only the used entries of each transaction. **Water-NS**[4]and **Water-SP**[4] are two applications from the Splash-2 benchmark suite [50] that model the forces among water molecules. One of the replaced critical sections is a loop of additions over different shared memory addresses with no aliasing. As commutative property of the addition guarantees they can be executed in any order without affecting the global result, we substitute it using a MAD atomic within the loop.

5.2.2 ***MCAS operations***. We also evaluate our MAD atomics for several data structures widely used in real applications. For each application, we implemented a transactional memory version.

BSTree uses a binary search tree suited for storing hierarchical data. The lock-free and MCAS versions are based on the work of Patel *et al.* [38], which, in turn, is based on Herlihy *et al.* [26]. **Deque** implements a double-ended queue (elements can be added and removed from either the front or the back). The MCAS implementation is based on the work by Doherty et al. [13]. The lock-free implementation is based on the work by Lagwankar [2], which, in turn, is based on Chase *et al.*. [8] and Herlihy *et al.* [26]. **HashMap** implements an associative array that maps keys to values using a hash function to index the elements. Because of its efficiency when looking up for data, it is commonly used in database indexing. Our version is implemented using separate chaining method with a lock-free doubly linked list [22] for collision resolution [11]. **MWObject** is a simple application that increments a set of four variables [15, 16]. In the original version, increments are made in steps of 16, forcing the least significant two bits of the variables to be 0. This limitation is imposed by the mechanism used in the original research but in our case we do not face this limitation and increments are made in steps of 1. **Queue** is an application that uses a queue to store collections of objects in FIFO order. The lock-free and MCAS versions are based on the work by Patel *et al.* [38], which, in turn, is based on Herlihy *et al.* [26]. **Stack** implements a LIFO-order queue (the last element inserted is the first to be removed). The lock-free implementation is based on the work of Williams [49] and Herlihy *et al.* [26]. **Intruder** is an application from the STAMP benchmark suite [36] that models a network intrusion detection system. We transformed the transaction used in its capture phase to extracts packets from a simple FIFO queue, into a MAD atomic operation.

6 RESULTS

Execution time results are normalized to a baseline implementation that uses mutex locks with the same system configuration, except for the scalability results, that are normalized to the lock version with one thread.

The performance improvements of MAD atomics heavily depend on the applications characteristics. Figure 7 shows the normalized execution time for the selected applications and concurrent data structures. MAD atomics outperform the mutex lock implementation in all cases but Water-SP. Water-SP is the only application affected by this extra synchronization point, mainly because it has almost no collisions or retries. This hypothesis is reinforced by the fact that TSX outperforms MAD even for one thread. Bitcoin benefits the most from MAD atomics, with up to 98% execution reduction time over mutex locks (when running on 32 threads). The baseline version of Bitcoin uses a naive implementation that has a single lock to block the transaction table. This shows the huge potential benefits of MAD atomics on codes written by inexperienced programmers that use coarse-grained locks. On average, MAD atomics perform 40% to 50% better than mutex locks for the same thread count. As thread count increases, the overhead of

[4] Water-NS and Water-SP are the only applications from the Splash-2 benchmark suite that meet the requirements to be transformed using MAD atomics.

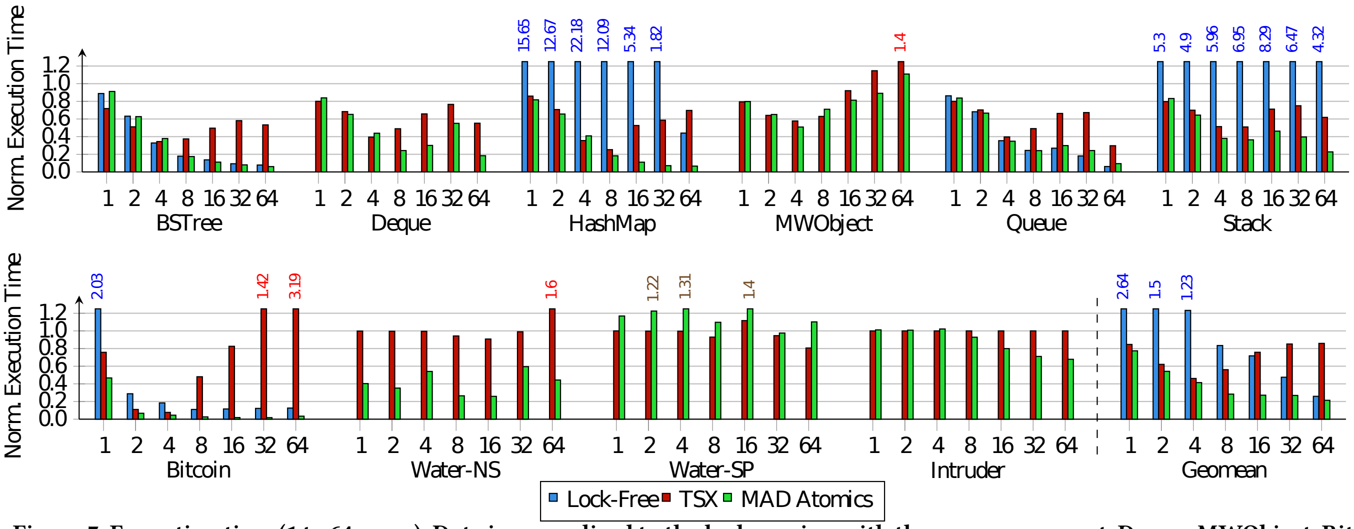


Figure 7: Execution time (1 to 64 cores). Data is normalized to the lock version with the same core count. Deque, MWObject, Bitcoin, Water-NS and Water-SP do not have lock-free version. Intruder does not have a lock or lock-free version, it is normalized against TSX.

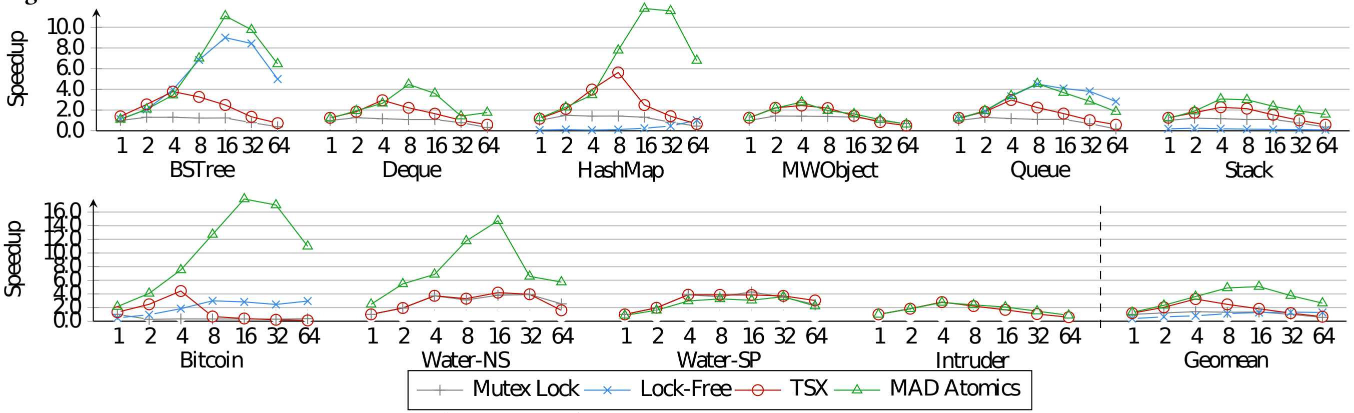


Figure 8: Scalability of the benchmarks (1 to 64 cores) . Each version is normalized to the lock version running a single thread. Deque, MWObject, Bitcoin, Water-NS and Water-SP do not have lock-free version. Intruder does not have a lock or lock-free version, it is normalized against TSX.

rollbacks due to additional collisions in TSX out-weight the performance improvements, making it less scalable than MAD atomics in our evaluation.

Scalability improves for all applications with respect to mutex locks, especially for BSTree, HashMap, Bitcoin and Water-NS, that can now can easily scale up to 16 threads (Figure 8). TSX scalability is limited to 4 threads (8 for HashMap), and only outperforms MAD atomics marginally for Water-SP due to the additional *mfence* as explained previously. For most benchmarks, the critical sections are too small and highly contended to get any advantage from current implementations of transactional memory: because of its requester-wins policy, recurrent conflict-induced aborts in TSX eventually make threads resort to the fallback path and execute the critical sections non-speculatively, in mutual exclusion, while the rest is forced to wait on the fallback lock. In summary, MAD atomics is a much simpler and elegant approach to ensure deadlock-free consistency.

Limited scalability of MWObject is due to constant updates to several memory addresses. MWObject is implemented using an MCAS operation inside a loop that will perform a retry when the data it compares has changed.

Finally, Figure 9 shows the normalized executed instructions for the different benchmarks. A reduction in executed instructions (not necessarily committed) translates into a reduction on the energy consumed by the processor. All benchmarks show a significant reduction in executed instructions. For 16 threads, MAD atomics execute, on average, only 21% of the original instruction count. This trend continues all the way up to 64 threads. TSX follows closely, with 41% instructions executed compared to the mutex lock implementation. This means that MAD atomics not only is faster than TSX, but also more energy efficient. And this is not even including the extra energy required by the TSX hardware.

7 DISCUSSION

When an application tries to acquire a contended mutex lock, it commonly ends up using a system call to interrupt the thread until the lock is freed. Alternatively, if the application notifies the

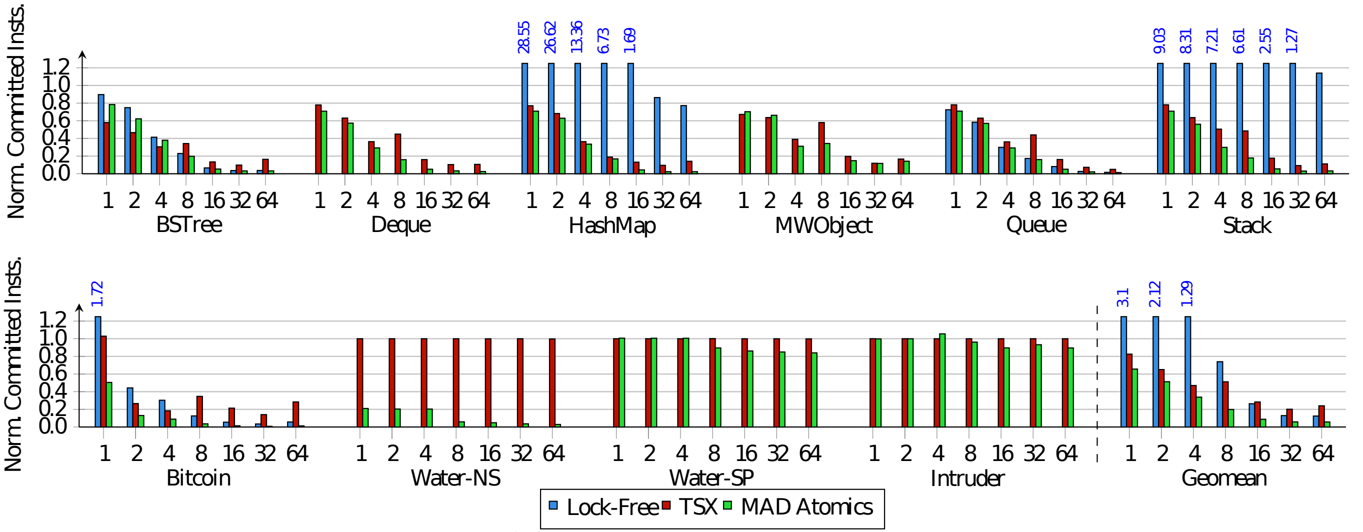


Figure 9: Normalized committed instructions (1 to 64 cores). Data is normalized against the lock version with the same core count. Deque, MWObject, Bitcoin, Water-NS and Water-SP do not have lock-free version. Intruder does not have a lock or lock-free version, it is normalized against TSX.

operating system during the mutex lock initialization, it can stay spinning waiting for the lock.

Either way, if a lock cannot be acquired, the execution has to continue with a system call. This means squashing and removing potentially useful instructions fetched, decoded, or even executed from the application. This may also introduce some cache pollution, as useful data can be evicted by the system calls and would need to be retrieved again from other levels of the memory hierarchy.

Furthermore, system calls can also produce other side effects besides squashing instructions. Indeed, system calls also pollute the translation lookaside buffer (TLB), with new accesses to the page walker for OS addresses.

On lock-free implementations, the application is heavily modified so that data can be updated with a single atomic operation. This approach increases instruction count due to spin-waiting and, in many cases, poisons the branch predictor with iterations of a loop that depends on other cores.

At this point we have already established why atomic read-modify-write (RMW) primitives outperform other approaches. Compared to mutex schemes, RMW primitives minimize serialization and require no OS intervention. Compared to HTM, RMW primitives have less overheads and are faster when facing contention, as stalling is more efficient than squashing. We have shown how MAD atomics can significantly reduce the overheads produced by a mutex-lock implementation. As our locks are non-speculative, the core can continue fetching new instructions and executing the non-memory related ones speculatively while waiting for the lock. Lock order is no longer relevant for the programmer, as the hardware will reorder them in a deadlock-free manner.

Support for multi-address atomics not only benefits scalar architectures, but also enables RMW instructions in vector systems (e.g., SIMD). A vector instruction computes several data elements exploiting data-level parallelism. Therefore, without multi-address atomics, it is not possible to provide vector atomic operations in a SIMD processor. SIMD architectures are ubiquitous in all market segments, and atomic vector instructions are critical in their development [33].

In this work we focus on a simple implementation of MAD atomics that can lock up to four addresses. We could not find applications with more than four *immutable* in-conflict addresses. This *immutable* restriction means that iterating over trees or linked lists within a critical section, as it is done in several STAMP benchmarks, is in most cases not transformable to MADs. Technically speaking, our simulated architecture can handle up to five addresses, due to general purpose x86 register limitations. This limitation can be relaxed by using newer ISAs that increase the logical register count from 16 to 32 (AVX-512 [28] and SVE [43]). Indeed, considering these ISAs would instantly increase MADs limit to eight addresses. Given the lack of applications with high arity requirements, we leave as future work increasing the arity of MAD atomics beyond eigth memory addresses.

In this paper we model an x86-like architecture as a commodity processor. However, MAD atomics can be used in any ISA that supports synchronization primitives like *mfence*. Implementation in CISC architectures is easier, since MAD atomics can be implement with a single instruction. RISC architectures, however, would require to suspend interrupts to ensure that no other instruction is inserted within the MAD atomic code.

8 RELATED WORK

Making non-blocking algorithm design easier has been one of the main goals of researchers for decades. Much of this work has focused on the double compare-and-swap (DCAS, or CAS2) primitive, a natural generalization of the CAS primitive. DCAS allows a thread to atomically perform a compare-and-swap with two memory locations and store two respective *new* values if *both* comparisons succeed. DCAS was implemented in the Motorola 68040 processor. However, Doherty *et al.* demonstrate that DCAS is not good enough to be used in non-blocking algorithm design, and higher arity operators are required [13].

Herlihy proposed a methodology for implementing highly concurrent data objects, that will become the basis for all n-ari primitives, but it was not disjoint [23, 24]. Disjoint-access parallel implementations allow two or more transactions to proceed in parallel, without interference, if they access disjoint sets of addresses.

Israeli and Rappaport introduced a non-blocking implementation of CASn (n-way atomic Compare-and-Swap) and n-way LL/SC for $P_{processors}$ out of single LL/SC [30]. Their two-phase locking approach is the basis for almost most CASn implementations, and was the first disjoint-access parallel implementation. Their CASn was not wait-free, however, and had worst-case time complexity $O(nP^3)$.

Anderson and Moir [5] improved upon the helping mechanism of [30] to implement a wait-free CASn, rather than merely non-blocking. They require only realistic sized words, and impose no worst-case performance penalty for the wait-free property, but the $O(nP^3)$ worst-case time is still the same. Unfortunately, they also still require a prohibitively large amount of space. Finally Moir proposed a more efficient non-blocking version of CASn (MW-CAS) [37]. The most significant gains were made by relaxing the wait-free property. However, it still requires extra space per word in the shared memory.

All of the previous implementations of CASn are complex and have high space overheads. Harris *et al.* introduced a practical multi-word compare-and-swap operation. It only requires either 0 or 2 bits per word and it is disjoint [21]. Wang *et al.* extend Harris work with persistence guarantees and support for recovery [48]. Feldman *et al.* propose a practical MCAS design that is wait-free in all scenarios [16], even with interrupts consistently causing a thread to retry. It is built from only portable atomic operations (e.g. atomic reads, atomic writes, compare-and-swap). It only requires a single bit to be reserved from each word, not requiring use of explicit memory barriers, and requiring only four words per address in the operation.

Regarding hardware implementations, Stone *et al.* introduced the Oklahoma update [44]. They offer a distributed transactional proposal to atomically write a group cachelines without resorting to broadcast or to a centralized arbiter. Because it is an all-or-nothing solution, it suffers from livelocks. Rajwar and Goodman address this issue using timestamps in [39]. Still, both proposals suffer from deadlocks that may arise from resource conflicts. Carter *et al.* compare several hardware lock implementations with their software counterparts [7]. They showed that hardware implementations can significantly reduce the lock acquisition and release times (25-94%). However, in highly optimized applications such as SPLASH-2, hardware locks only provided 3-6% better performance.

To the best of our knowledge, the only hardware implementation of MCAS (CASn) was performed by Patel *et al.* [38]. Their implementation is also based on a structure that re-orders locks (the MCAS Unit), which contains a table (the MCAS table) in which each entry keeps a memory location that will be locked (in ascending order). Multiple MTS instructions are used to setup the MCAS table and later an MCAS instruction starts locking the addresses from the MCAS table. Using separate instructions for each step incurs the risk of having a context switch or other interrupt in the middle of the procedure, leaving it incomplete. On the other hand, two versions of the MCAS implementation are proposed: a *basic* proposal (MCAS-BASE) and an optimized version (MCAS-OPT). MCAS-BASE is similar to our approach, but it does not consider the risk of deadlock due to resource limitations, a key contribution of this work, and therefore it is not deadlock-free. MCAS-OPT employs a back-off mechanism that causes to abort upon receiving an invalidation, which makes it to behave as an HTM implementation using lazy conflict detection.

Afshar *et al.* [4] and Strøm *et al.* [45] introduced dedicated hardware support for locking on-chip. Their approaches are similar to a centralized scoreboard for all cores. A field for each core is used to register synchronization participation. These implementations can be used for CAS implementations, but would require further modifications to support MCAS. Besides using a CAS operation on a shared, external main memory, an on-chip shared scratchpad memory can be used to support synchronization [20].

Finally, although Motorola also patented in 1986 a DCAS hardware implementation for linked lists, this line was abandoned and the patent is now expired [35].

9 CONCLUSIONS

Efficient, non-speculative, deadlock-free multi-address atomic operations are a desirable feature missing in current processors. In this work, we proposed a truly deadlock-free solution for multi-address atomics that takes into account real hardware limitations, such as private/shared structures associativity and eviction buffers. MAD atomics allow for an efficient implementation of the MCAS primitive and opens the door to many other multi-address atomic primitives. MAD atomics are also proposed as a alternative for short transactions. As MAD atomics are non-speculative, they offer better performance than HTM and do not suffer aborts/re-executions. Ideally, both techniques can be used together as needed.

Our results show that for the evaluated applications and concurrent data structures, MAD atomics outperform software locks by up to 18× (3.4× on average) and 2.7× on average compared to TSX, improving scalability from one core (software locks) up to 16 cores. Performance and executed instruction count improvements directly translate into energy savings, that reach 23× in average for 32 and 64 cores.

Indeed, lock-free data structures introduce an interesting performance and energy improvement over the mutex lock-based approach. But this is further improved with MAD atomics, which also makes its development easier and more flexible than lock-free versions, achieving complexity-effective, non-speculative, non-deadlocking, fine-grained locking for multiple addresses, relying only on the coherence protocol and a predetermined locking order, and requiring less that 68 bytes of extra storage per core.

ACKNOWLEDGMENTS

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 819134), Vetenskapsradet project 2018-05254 and by the European joint Effort toward a Highly Productive Programming Environment for Heterogeneous Exascale Computing (EPEEC) (grant No 801051).

REFERENCES

[1] [n.d.]. Ice Lake (client) - Microarchitectures - Intel. https://en.wikichip.org/wiki/intel/microarchitectures/ice_lake_(client)

[2] [n.d.]. A lock-free work-stealing deque. https://github.com/ssbl/concurrent-deque

[3] Andreas Abel and Jan Reineke. 2019. uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In *ASPLOS (ASPLOS '19)*. New York, NY, USA, 673–686.

[4] S. Afshar, M. Behnam, R.J. Bril, and T. Nolte. 2017. Per processor spin-based protocols for multiprocessor real-time systems. *Leibniz Transactions on Embedded Systems* (2017).

[5] James H. Anderson and Mark Moir. 1995. Universal Constructions for Multi-Object Operations. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*. 184–193.

[6] ARM. 2020. *ARM® Cortex®-X1 Core Technical Reference Manual*. Number r1p1.

[7] John Carter, Chen chi Kuo, and Ravindra Kuramkote. 2003. A Comparison of Software and Hardware Synchronization Mechanisms for Distributed Shared Memory Multiprocessors.

[8] David Chase and Yossi Lev. 2005. Dynamic Circular Work-Stealing Deque. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures* (Las Vegas, Nevada, USA) *(SPAA '05)*. Association for Computing Machinery, New York, NY, USA, 21–28.

[9] Guancheng Chen and Per Stenstrom. 2012. Critical lock analysis: Diagnosing critical section bottlenecks in multithreaded applications. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–11.

[10] Edward G. Coffman, Melanie Elphick, and Arie Shoshani. 1971. System deadlocks. *ACM Computing Surveys (CSUR)* 3, 2 (June 1971), 67–78.

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

[12] Edsger W. Dijkstra. [n.d.]. Hierarchical ordering of sequential processes. *EDW-310, E.W. Dijkstra Archive, Center for American History, University of Texas at Austin* ([n. d.]).

[13] Simon Doherty, David L. Detlefs, Lindsay Groves, Christine H. Flood, Victor Luchangco, Paul A. Martin, Mark Moir, Nir Shavit, and Guy L. Steele. 2004. DCAS is Not a Silver Bullet for Nonblocking Algorithm Design. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*. 216–224.

[14] Stijn Eyerman and Lieven Eeckhout. 2010. Modeling Critical Sections in Amdahl's Law and Its Implications for Multicore Design. *SIGARCH Comput. Archit. News* 38, 3 (June 2010), 362–370.

[15] Steven Feldman, Pierre Laborde, and Damian Dechev. 2013. A Practical Wait-Free Multi-Word Compare-and-Swap Operation.

[16] Steven Feldman, Pierre Laborde, and Damian Dechev. 2014. A Wait-Free Multi-Word Compare-and-Swap Operation. *International Journal of Parallel Programming* (Aug. 2014).

[17] Agner Fog. 2018. Instruction Tables. Instruction latencies, throughputs and micro-operation breakdowns. Available at http://www.agner.org/optimize/instruction_tables.pdf.

[18] Agner Fog. 2020. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. Available at https://www.agner.org/optimize/microarchitecture.pdf.

[19] Lance Hammond, Brian D. Carlstrom, Vicky Wong, Ben Hertzberg, Mike Chen, Christos Kozyrakis, and Kunle Olukotun. 2004. Programming with Transactional Coherence and Consistency (TCC). In *11th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*. 1–13.

[20] Henrik Hansen, Emad Maroun, Andreas Kristensen, Jimmi Marquart, and Martin Schoeberl. 2017. A shared scratchpad memory with synchronization support. 1–6.

[21] Timothy Harris, Keir Fraser, and Ian Pratt. 2003. A Practical Multi-Word Compare-And-Swap Operation. *Proceedings of the 16th International Symposium on Distributed Computing*.

[22] Timothy L. Harris. 2001. A Pragmatic Implementation of Non-Blocking Linked-Lists. In *Proceedings of the 15th International Conference on Distributed Computing (DISC '01)*. Springer-Verlag, Berlin, Heidelberg, 300–314.

[23] Maurice Herlihy. 1988. Impossibility and universality results for wait-free synchronization. In *PODC '88*.

[24] Maurice Herlihy. 1993. A Methodology for Implementing Highly Concurrent Data Objects. (Nov. 1993), 745–770.

[25] Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *20st Int'l Symp. on Computer Architecture (ISCA)*. 289–300.

[26] Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[27] Intel. 2016. Intel® 64 and IA-32 Architectures Optimization Reference Manual. www.intel.com.

[28] Intel Corporation. 2017. *Intel® Architecture Instruction Set Extensions Programming Reference*. Number 319433-029US.

[29] Intel Corporation. 2020. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Number 325462-072US.

[30] Amos Israeli and Lihu Rappoport. 1994. Disjoint-Access-Parallel Implementations of Strong Shared Memory Primitives. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*. 151–160.

[31] Norman P. Jouppi. 1990. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *17th Int'l Symp. on Computer Architecture (ISCA)*. 364–373.

[32] Daniel Kondor. [n.d.]. https://senseable2015-6.mit.edu/bitcoin/

[33] S. Kumar, D. Kim, M. Smelyanskiy, Y. Chen, J. Chhugani, C. J. Hughes, C. Kim, V. W. Lee, and A. D. Nguyen. 2008. Atomic Vector Operations on Chip Multiprocessors. In *35th Int'l Symp. on Computer Architecture (ISCA)*. 441–452.

[34] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Carlos Escuin, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Anthony Gutierrez, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Miquel Moreto, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, William Wang, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. 2020. The gem5 Simulator: Version 20.0+. arXiv:2007.03152 [cs.AR]

[35] Douglas Macgregor, David S. Mothersole, and John Zolnowsky. 1986. Method and apparatus for a compare and swap instruction. U.S. Patent 4584640.

[36] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. 2008. STAMP: Stanford Transactional Applications for Multi-Processing. In *Int'l Symp. on Workload Characterization (IISWC)*. 35–46.

[37] Mark Moir. 1997. Transparent support for wait-free transactions. In *Distributed Algorithms*. 305–319.

[38] Srishty Patel, Rajshekar Kalayappan, Ishani Mahajan, and Smruti R. Sarangi. 2017. A hardware implementation of the MCAS synchronization primitive. In *2017 Design, Automation, and Test in Europe (DATE)*. 918–921.

[39] Ravi Rajwar and James R Goodman. 2002. Transactional Lock-Free Execution of Lock-Based Programs. In *10th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*. 5–17.

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

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

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

[43] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, Alastair Reid, Alejandro Rico, and Paul Walker. 2017. The ARM Scalable Vector Extension. *IEEE Micro* (2017).

[44] Janice M. Stone, Harold S. Stone, Philip Heidelberger, and John Turek. 1993. Multiple Reservations and the Oklahoma Update. *IEEE Parallel & Distributed Technology: Systems & Applications* 1, 4 (Nov. 1993), 58–71.

[45] T. B. Strøm and M. Schoeberl. 2018. Hardlock: A Concurrent Real-Time Multicore Locking Unit. In *2018 IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC)*. 9–16.

[46] Herb Sutter. 2005. The Trouble With Locks. *C/C++ Users Journal* (March 2005).

[47] Herb Sutter. 2008. Writing Lock-Free Code: A Corrected Queue. *Dr. Dobb's Journal* (October 2008).

[48] Tianzheng Wang, Justin J. Levandoski, and Per-Åke Larson. 2018. Easy Lock-Free Indexing in Non-Volatile Memory. In *34th IEEE International Conference on Data Engineering (ICDE)*. 461–472.

[49] A. Williams. 2019. *C++ Concurrency in Action*. Manning Publications. https://books.google.es/books?id=PsPItwEACAAJ

[50] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *22nd Int'l Symp. on Computer Architecture (ISCA)*. 24–36.

[51] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. 2013. Performance Evaluation of Intel Transactional Synchronization Extensions for High-Performance. In *2013 Conf. on Supercomputing (SC)*. 19:1–19:11.