Hardware Support for Durable Atomic Instructions for Persistent Parallel Programming

Khan Shaikhul Hadi
University of Central Florida
shaikhulhadi@knights.ucf.edu

Naveed Ul Mustafa
University of Central Florida
unknown.naveedulmustafa@ucf.edu

Mark Heinrich
University of Central Florida
heinrich@ucf.edu

Yan Solihin
University of Central Florida
yan.solihin@ucf.edu

2023 60th ACM/IEEE Design Automation Conference (DAC) | 979-8-3503-2348-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/DAC56929.2023.10247729

***Abstract*—Persistent memory is emerging as an attractive main memory fabric capable of hosting persistent data. However, its programmability is hampered by the lack of persistent synchronization primitives. Atomic instructions are immensely useful for higher-level synchronization (locks and barriers) and for supporting lock-free data structures, but they have no durable/persistent version. In this paper, we propose a new approach to solve the problem: *durable atomic instructions* (DAIs). We show that DAIs can be supported with minor hardware support (low-cost modifications to the cache coherence protocol), and simultaneously achieve high performance, scalability, and crash consistency.**

I. Introduction

The introduction of Non-Volatile Memory (NVM) devices, such as Intel Optane PMem [1], provides system designers with the option of replacing or augmenting volatile memory with NVM. While Intel has announced discontinuation of NVM products, alternatives are emerging, such as SLC NAND-based memory (e.g. Kioxia FL6 [2]), as well as ReRAM and MRAM-based memories.

Besides its non-volatility, NVM's byte addressability and low access latency make it attractive to host persistent data in memory instead of storage. However, despite NVM's potential, taking advantage of NVM for persistent data requires high programming effort. The programmer is expected to orchestrate data movement from the volatile cache hierarchy to NVM using low-level instructions such as cache line flushes and store fences. He/she is also expected to define durable atomic regions that mingle both concurrency and persistency. Higher level abstraction and useful primitives that simplify persistency programming are sorely needed.

One type of primitives that are widely relied upon by programmers are atomic instructions. An atomic instruction provides an atomic sequence of read, modify and write (RMW) on a memory word. Examples include *compare and swap* (CAS), *fetch and op* (such as fetch and add, fetch and increment, etc.), *atomic exchange*, etc. The processor ensures that an atomic instruction executed by one thread appears to execute sequentially with respect to atomic instructions executed by all other threads. As a result, atomic instructions serve at least three primary functions. First, they provide a high-performance alternative to locks for a critical section that only modifies a single word. Second, they are foundational building blocks for constructing higher-level synchronization primitives, such as locks, barriers, etc. Finally, they are essential primitives for developing lock-free data structures [3], [4], [5], in particular CAS.

```
1       if(CAS(&last->next,next,new_node)){
2           CLFLUSH(&last->next);
3           FENCE;}
```

Listing 1: A compare-and-swap is followed by cache line flush and fence to persist lock free data structure update [6].

However, currently there is no durable version of atomic instructions. This makes it challenging to support the three primary functions above for persistent data. To illustrate this, consider the code snippet for node insertion on a lock-free linked-list in Listing 1 [6]. Each *update step* (line 1) performs a CAS, and is followed by a *persist step* (i.e., CLFLUSH and FENCE) on lines 2-3. The CLFLUSH evicts a memory block from volatile caches to the NVM, while the FENCE provides a persist barrier by ensuring that the flush reaches the persistence domain before new reads/writes are allowed to persist.

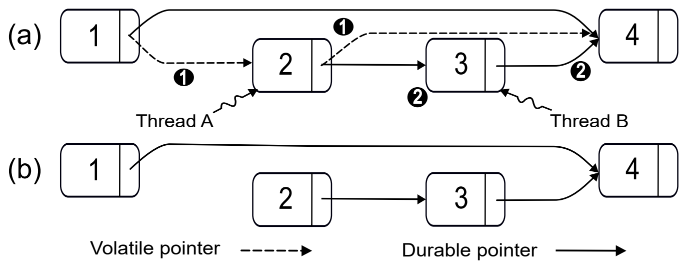


Fig. 1: Illustrating the problem with the use of current atomic instructions on persistent data. (a) shows concurrent insertions by threads A and B, while (b) shows a possible crash inconsistent state.

Figure 1a illustrates a lock-free linked list, where two threads (A and B) are attempting to simultaneously insert nodes 2 and 3, respectively, using the code in Listing 1. Suppose that the CASes have been executed atomically, resulting in successful insertion of node 2 ❶ followed by insertion of node 3, reflected in the volatile caches. Next, suppose that thread B's flush and fence are completed ❷ which makes node 3's insertion durable, but a power failure occurs before thread A's flush and fence are completed. Hence, thread A's change to node 1's `next` pointer is lost in the volatile caches. Figure 1b shows the state of the linked list upon crash recovery, where nodes 2 and 3 have been lost.

Fundamentally, the problem arises because *update* and *persist* are not atomic as they are performed by different instructions. There are currently no satisfactory solutions for the problem. One could wrap the entire node insertion code with a durable transaction memory (DTM) [7]. However, DTM is intended to replace the use of locks, hence it requires heavy duty hardware support with substantial changes to the processor and cache designs. In contrast, an atomic instruction is a building block for synchronization primitives including locks. Furthermore, DTM semantics require abort and retry to be supported, whereas an atomic instruction does not. Finally, DTM is built on top of TMs, and most commercial TMs are best effort as they provide no guarantee that a transaction will always commit. In contrast, atomic instructions will always complete. Software approaches like PMCAS [8] and PMwCAS [9] keep a log of memory information to ensure that data can be recovered on a crash and prevent other threads from reading non-persistent data by enforcing a flush-on-read mechanism. While they could work, they are overkill, as they require crash recovery code to be developed, and incur a significant performance loss due to the overhead involved in reading and the additional cache line flushes.

In this paper, we propose a new approach to the problem: *durable atomic instructions* (DAIs). DAIs are the durable version of atomic instructions, guaranteeing both atomicity and persistency. DAIs ensure that a word that becomes globally visible (to other threads) has persisted. To support DAIs, we extend cache coherence protocol mechanisms that already exist for atomic instructions, resulting in a modified MESI protocol (which we refer as durMESI) that combine atomicity and persistence required for DAIs.

Our proposed DAIs provide multiple advantages. First, DAIs require only minor hardware modification to the cache coherence protocol. Second, DAIs do not require significant application modifications as applications can simply replace atomic instructions with DAIs in their code, and the replacement can be automated by the compiler. This is in contrast to PMCAS [8] and PMwCAS [9] which add many instructions and require additional application modifications and correctness reasoning effort. Finally, DAIs preserve atomic instructions' semantics but add durability/persistency, which is a key to DAIs' simplicity. DAIs do not require crash recovery code, do not need abort and retry, or alternative code to guarantee forward progress.

To summarize, our work makes the following contributions:

1) We propose DAIs, a novel persistent version of atomic instructions, as primitives that provide building blocks for other synchronization primitives and lock-free data structures.
2) We present modifications to cache coherence protocols to enable DAIs in MESI, which we refer to as the durMESI protocol.
3) We evaluate DAIs using Splash-4 parallel benchmarks that use lock-free data structures. Our results show that on average, DAI 6.4% faster than regular atomic instructions with flush and fence and comparable scalability, while at the same time provide crash-consistency that the latter does not.

II. Background and Related Work

A. Persistency Management Instructions

At the instruction set level, Intel x86 provides cache line flush instructions (CLFLUSH, CLFLUSHOPT and CLWB) to flush/write back data to main memory, and memory or store fence instructions (MFENCE, SFENCE) to enforce ordering [10]. Together, cache line flush and fence form a persist barrier. In the ARM instruction set [11] similar instructions are provided (DC CVAP, DC CVADP for durability and DSB as a persist barrier). These instructions could be used directly by the programmer or by a persistent programming library such as Intel PMDK [12]. These instructions manage persistency but not atomicity.

B. Software-Based Atomicity of Update and Persistency

Recent works such as PMCAS [8] and PMwCAS [9] proposed the design of *multiword* crash consistent CAS. For each word in a multiword *frame*, they designate an unused address bit of the word as a *persistency marker*. The bit is set to 1 on updating the word, indicating that the update is not yet persisted and reads are not allowed. The bit is cleared after persisting the update. Furthermore, frame-level metadata is maintained to monitor whether all words have been successfully persisted or not to support recovery in the event of a crash. When a thread requests an updated but not-yet persisted word, it must wait for the word to become persistent. In doing so, as a by-product, they provide an illusion of atomicity for the *update* and *persist* steps. Both approaches use expensive cache line flush and fence instructions to persist words in NVM. Additionally, they maintain metadata for each word to support crash recovery and require application modifications to include library calls. While the approach can be used for achieving a durable atomic single-word CAS, it is overkill due to requiring crash recovery code, extra metadata management, and reliance on expensive flush and fence instructions. In contrast, our DAIs do not require any metadata, crash recovery code, and provide just-enough hardware modifications for a single word CAS.

III. Design of Durable Atomic Instructions (DAIs)

A. Correctness Criteria and Design Principles

To understand the correctness criteria DAIs must meet, let us first consider how regular atomic instructions violate crash consistency. Figure 2(a) illustrates the timeline of thread execution, with a sequence of compare and swap instruction (CAS), cache line flush (CLFLUSH), and fence (Listing 1). At the completion of CAS, the updated data is visible (to other threads) but not yet persisted, until CLFLUSH is completed, at which time the update is both visible and persisted. Figure 2(b) illustrates a case where crash consistency is violated. It shows the execution of two threads (T0 and T1) with T0 performing CAS (epoch e_0), with the updated value consumed by T1 which performs its own CAS (epoch e_1). Similar to the correctness requirement in durable transactions [13], we can think of each CAS execution as an epoch. Since T1 consumes data in e_1 produced by T0 in e_0, we denote the dependence

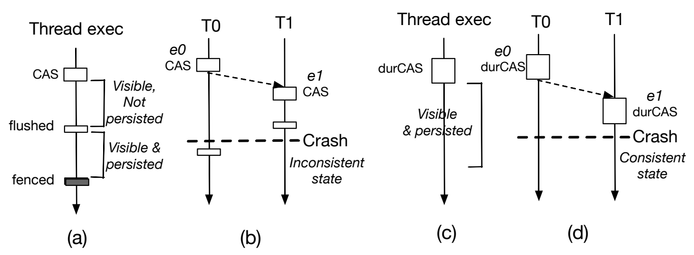


Fig. 2: Illustrating how traditional compare-and-swap (CAS) instruction violates crash consistency ((a) and (b)), compared to how durable CAS preserves crash consistency ((c) and (d)).

relationship as $e_0 \rightarrow e_1$. As stated in [13], persist order must adhere to $e_0 \rightarrow e_1$ as well, meaning that T1's CAS cannot persist before T0's CAS. However, with CAS and flush/fence as separate instructions, such a property cannot be guaranteed. CLFLUSH of T1 may be executed earlier than the CLFLUSH of T0, which means that T1 persisted value that was produced but not persisted yet by T0. If there is a crash at this point, an inconsistent state results.

DAIs achieve persistence and visibility in a single instruction. Figure 2(c) shows a durCAS, the durable version of atomic instruction CAS. durCAS persists an update first before allowing it to be visible to other threads. By the time the value is visible to (and consumed by) T1's durCAS (Figure 2(d)), it was already persisted, hence a consistent state is achieved when a crash occurs.

To achieve the visible-after-persist property, our DAIs should perform the following atomically: (1) send data update to the persistence domain, (2) receive acknowledgment from the persistence domain of its receipt, and (3) allow visibility of the new data. These steps ideally should be achieved with *minimal hardware changes* and be simple to adapt across different platforms. For programmability, DAIs should *preserve the interface* of atomic instructions, adding only durability to their semantics.

B. Semantics of DAIs

Our proposed DAIs preserve the interface and functionality of as atomic instructions but add durability. Table I lists atomic instructions and their DAI counterparts. For example, CAS(A,\$1,\$2) reads a value from address A, if it is equal to the value in register \$1, writes the value in register \$2 to A. In contrast, durCAS(A,\$1,\$2) additionally persists the value in register \$2 to A.

Note that different platforms may define different persistence domains. For example, some platforms may include only the main memory and memory controller's write pending queue (WPQ) in the persistence domain (e.g. Intel ADR), but other platforms may include the entire cache hierarchy in the persistence domain (e.g. Intel eADR). DAI semantics are independent of the persistence domain scope, and the mechanism of persistence is decoupled from the semantics. Thus, DAIs semantics make them easy to port across various platforms.

TABLE I: Semantics of DAIs. \$ denotes a register. Parantheses denote additional semantics added by DAIs for compare-and-swap (CAS), fetch-and-add (FAA), fetch-and-sub (FAS) and test-and-set (TAS) instructions. For brevity, *fetch-and-or/and/xor/nand* are omitted.

Atomic Inst	DAI	Semantics
CAS(A,\$1,\$2)	durCAS(A,\$1,\$2)	read value from address A, if equals to \$1, write (and persist) \$2 to A
FAA(A,\$1)	durFAA(A,\$1)	read value from address A, add it by \$1, write (and persist) the result to A
FAS(A,\$1)	durFAS(A,\$1)	read value from address A, subtract it by \$1, write (and persist) the result to A
TAS(A,\$1)	durTAS(A, \$1)	read value from address A, write (and persist) \$1 to A, put old value of A in \$1

C. Approaches to Architecture Support for DAIs

Recall that one of our design principles is to keep hardware changes to a minimum, and make it adaptable across different platforms. With this principles in mind, we consider several ways atomic instructions are currently implemented.

One method is to lock the bus, e.g. x86 LOCK prefix, so that no two atomic instructions can access the memory at the same time. However, this method has several substantial drawbacks. First, it slows down execution in all cores, even those that do not use atomic instructions. Second, it is not scalable, and harder to design as the number of cores increases as buses are replaced with point-to-point interconnects.

Another method for implementing atomic instructions is to rely on a lower-level primitive, a pair of load-linked (`LL`) and store conditional (`SC`) instructions. The pair provides an illusion of atomicity. LL/SC pair is available in MIPS, ARM (`ldrex`/`strex`), and PowerPC (`lwarx`/`stwcx`) instruction sets. LL reads a word from memory, and records the address in a special register. If an invalidation or intervention message is received to the address (indicating another core reads or writes to it), then the illusion is broken and the register is cleared, causing the subsequent SC to fail. SC succeeds only when no other cores interfere between the LL and SC. While LL/SC is efficient, it cannot be used for DAIs, because DAIs require flushing data to memory, but flushing goes beyond the core control and hence cannot be canceled by the core.

The final method for implementing atomic instructions is to rely on cache coherence protocol support. The basic mechanism is to reserve a block involved in an atomic instruction in the private cache of the core that executes the instruction, and refuse intervention or invalidation requests made by other cores until the instruction is completed, after which the reservation is released. We observe that this mechanism can be extended to support DAIs, simply by extending the reservation until a block has been persisted. Considering that the additional hardware complexity for supporting DAIs is small, we choose this method.

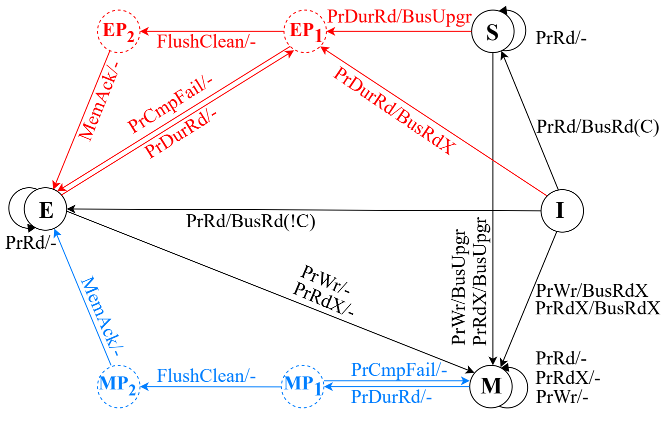
D. *Durable MESI (durMESI) Protocol*

To illustrate the modification that need to be made to the cache coherence protocol, we will discuss a design built on top of the MESI protocol, which is the foundation of cache coherence protocols in many platforms today. We refer to our protocol as durMESI.

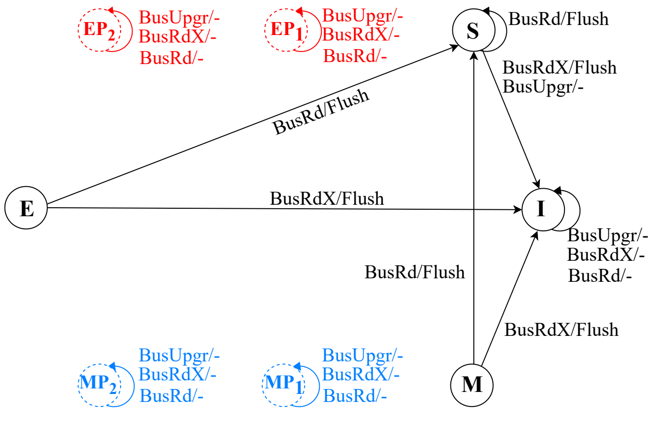
Figure 3 shows our durMESI cache coherence protocol, with additions over MESI shown in blue and red. For simplicity, the diagram assumes multiple cores sharing a bus that connects private caches. The protocol is split into one where state transitions occur because of requests made from the local core (top) versus requests snooped on the bus originating from a remote core (bottom). The notations follow from [14]. Notably, each transition is shown in the format of X/Y where X is the triggering event that causes the state transition, and Y is the signal that results from the state transition. PrRd, PrRdX and PrWr are signals issued by the local core due to executing a load or a store instruction. BusRd, BusUpgr/BusRdX are bus requests issued by the local core (top) or issued by a remote core that were snooped on the bus (bottom), with BusRdX additionally denoting a cache miss. Flush puts data on the bus for other cores and writes it back to the main memory. The local copy is either invalidated (if the final state is Invalid) or retained as a clean copy (if the final state is Shared).

To support DAIs, we add four transient states, *Exclusive-to-be-Persisted* (EP1 and EP2) and *Modified to-be-Persisted* (MP1 and MP2), and new core-generated signals corresponding to the execution of DAIs, namely PrDurRd. We need PrDurRd for DAI execution, instead of reusing PrRdX, because the final state is different (E vs. M). DAI *clean-flushes* (i.e. writes back but retain a copy) a block to the persistence domain, in the process turning the block clean. Thus, the more appropriate final state for PrDurRd is Exclusive (E). An advantage of choosing E over M as the final stage is that if the block is evicted, it can just be discarded, instead of written back, which reduces bandwidth pressure to both the last level cache (LLC) if evicted from the private cache and to main memory if evicted from the LLC. PrCmpFail is specific to durCAS (not needed by other DAIs), that indicates that the comparison did not yield a match. The MemAck signal is generated by the persistence domain, e.g. the memory controller (MC), indicating the receipt of the flushed cache block. Receiving MemAck confirms that durability has been achieved.

We will now cover each initial state of a block to which durCAS is applied to, in Figure 3 (top). Transitions in blue or red are ones added to the MESI protocol. Suppose a block is initially uncached or cached with invalid (I) state. The execution of durCAS results in PrDurRd being sent to the private cache's *coherence controller* (CC). The resulting cache miss posts BusRdX on the bus, and the state transitions to EP1. After receiving a response with the data block (supplied by a peer cache or the LLC/memory), DAI is performed in EP1, e.g. for durCAS the comparison and write are performed. Then the CC clean-flushes the updated block to the persistence domain and transitions from EP1 to EP2. Upon receiving the flushed block, the MC replies with a MemAck signal.



(a) Process Signal



(b) Bus Signal

Fig. 3: durable MESI protocol (durMESI)

Upon receiving MemAck, the CC concludes that the block has persisted, hence it transitions from EP2 to Exclusive. Note that for durCAS, a non-matching comparison skips the swap/write. To optimize for that, we let the CC know through PrCmpFail, which transitions from EP1 directly to Exclusive, skipping the flushing. Let us now consider other initial states. If the initial state is Shared, the execution of DAI requires invalidation of peer caches through BusUpgr to transition to EP1, while if the initial state is Exclusive, no bus transaction is needed as peer caches do not have a copy. Once in EP1, the remaining steps are the same as in the Invalid case.

If the initial state is M, no other cached copies exist, hence no bus transaction is triggered when transitioning from M to MP1. The transitions of MP1 $\rightarrow$ MP2 $\rightarrow$ E are similar to EP1 $\rightarrow$ EP2 $\rightarrow$ E. One difference is when the comparison fails (PrCmpFail). Since the block is dirty, the state must transition back to Modified rather than Exclusive.

Figure 3 (bottom) shows the durMESI protocol reacting to snooped requests from other cores appearing on the bus. The modifications here are minimal, mainly that when the state is in one of the transient states (EP1/2 or MP1/2), any external requests are ignored, i.e. either queued to be responded after the state transitions to a stable state, or responded with a negative acknowledgement (requestors will then retry). While

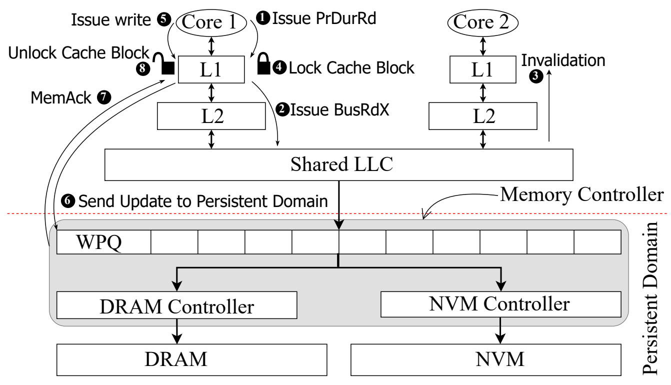


Fig. 4: Different step of DAI at hardware level

in a transient state, the block is *locked* (i.e., cannot be evicted) in the private cache.

Figure 4 illustrates a system in which durMESI is applied. The system has multiple cores with private L1 and L2 caches, a shared LLC, and the persistence domain consisting of NVM and write pending queue (WPQ) in the MC. Suppose Core 1 executes a DAI and issues PrDurRd ❶ to the L1 cache for a block that is not cached in the core's L1/L2 caches, but cached in Modified state in the second core's L1. The first core CC issues BusRdX to the LLC ❷, which sends an invalidation to the second core ❸. The second peer core flushes its block to the LLC, which forwards the block to the first core. Upon receiving the block, the first core CC locks the block in the L1 cache in EP1/2 states ❹, and performs the atomic operation ❺. Once completed, the block is clean-flushed to update memory ❻, and the MC replies with MemAck ❼ indicating persistence is achieved. Finally, the first core CC transitions the block to the Exclusive state which unlocks the block in the L1 ❽.

IV. Experimental Setup

Simulation environment: We built a multicore system with durMESI coherence protocol on top of Snipersim [15], a multicore simulator with an Intel Pin front-end. The simulated system is detailed in Table II. It is a single-socket Xeon Nehalem-like microarchitecture with 32 cores. Cache access latencies were obtained using CACTI [16], and NVM-DRAM latency ratio from [17].

TABLE II: Simulation Environment

Processor	32-core (2.66 GHz each), 64-bit X86 ISA, Gainstown architecture (Nehalem micro-arch), 45nm technology, Pentinum M branch predictor

TLB	L1D TLB: 64 entries, 4-way;
	L2 TLB: 512 entries, 4-way;
	miss penalty: 30 cycle

Cache	64-bytes block, LRU replacement policy Private L1D cache:32KB, 8-way, 4 cycle access time Private L2 cache: 256KB, 8-way, 8 cycle access time Shared L3 : 64MB, 16-way, 24 cycles access time
Memory Controller	Queue : 100 entry Bandwidth: 7.6 GB/s
Main Memory	DRAM latency: 54ns; NVM latency: 162ns

Evaluated Workload: We used the Splash-4 [18] benchmark suite, which consists of high performance computing benchmarks that have been recently adapted to use lock-free constructs. We ported them to utilize persistent memory by allocating the heap and all shared variables in the persistent memory, and applying cache line flush and fence instructions or DAIs to persistent data. For each benchmark, we simulated the entire run but collect statistics for the parallel region of interest (ROI). Some benchmarks that did not run correctly were omitted.

TABLE III: Splash-4 Benchmark Suite

Benchmark	Input	Benchmark	Input
Barnes	n16384	Radix	-n1048576
Cholesky	tk29.O	Raytrace	balls4.env
FFT	-m20 -l6	Water-SP	-n512

FMM	parsec_simsmall	Ocean-N	-n514
			IMAX = 4049
			JMAX = 4049

Ocean-C	-n514		

Schemes Comparison: We compare DAIs against the non-crash consistent solution with cache line flush and fence instructions, along the line shown in Listing 1. We refer to this solution as ACF (Atomic + CLFLUSH + FENCE). ACF does not provide crash consistency, hence it is not a real solution compared to DAI, however it serves as a good reference to assess how efficiently our DAIs perform.

V. Evaluation

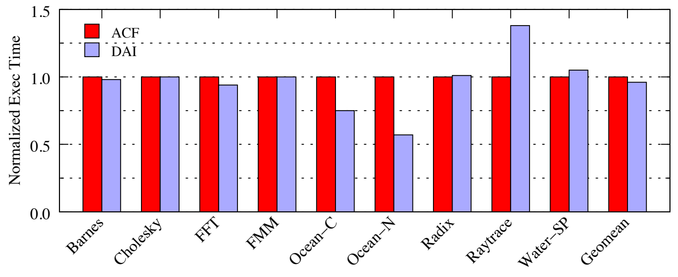


Fig. 5: Execution time of ACF vs. DAI (lower is better), normalized to that of ACF.

Figure 5 shows the execution time of DAI on durMESI, normalized to atomic instruction with cache line flush and fence (ACF) for Splash-4 benchmarks on 16 threads. We observe that on average, DAIs show 6.4% speedup (i.e., lower execution time) over ACF, which is remarkable considering they achieve crash consistency as well. For some benchmarks (*Ocean-Cont* and *Ocean-Non*), DAI performs significantly better compared to ACF, but performs significantly worse for *Raytrace*.

To understand the performance better, Figure 6 (Left) shows the dynamic percentage of atomic/DAI updates as a fraction of total updates (atomics/DAIs plus regular stores). For Splash-4 benchmarks, atomic instructions are mostly used in barriers for synchronization among threads. As a barrier is a global synchronization, its execution has a significant impact on the overall execution time of the workload. Three benchmarks stand out over the others as having much higher fraction of atomics ($> 0.006\%$), which are the same benchmarks where

DAIs performance diverge from ACF. DAIs have two major advantages over ACF: (1) fewer dynamic instructions (one DAI vs. three instructions: atomic, cache line flush, and fence), and (2) the reliance on clean flushing (vs. CLFLUSH in ACF) helps keeping data with high temporal locality resident in the cache.

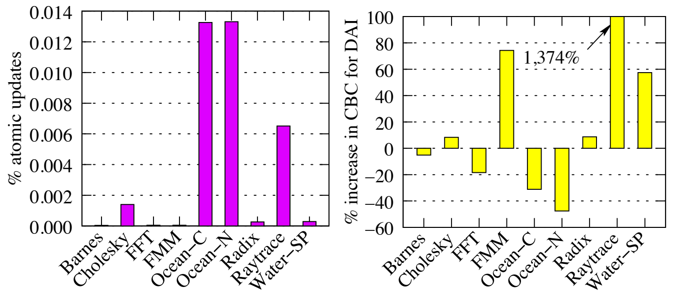


Fig. 6: Ratio of dynamic number of atomics over all writes (left), and % increase in Cache Block Contention (CBC) for DAI compared to ACF.

However, DAI locks a block in the cache for a longer time than a regular atomic instruction, because it needs to flush and wait for a MemAck before releasing the lock. Hence, while a core is executing DAI, other cores wishing to access the same block must wait longer. To asses this effect, we measure the time a request (due to load/store/DAI/atomic) must wait because the block in coherence controller is unable to respond, e.g. due to a transient state or flushing a block. We refer to this wait time (or delay) as cache block contention (CBC), and measure how much CBC increases with DAIs over ACF. Figure 6 (right) shows this percentage increase. There are several interesting observations. First, the figure explains why DAIs on Raytrace are slower than ACF: its CBC increases substantially (more than 1000%). In some benchmarks (*Ocean-Cont* and *Ocean-Non*), CBC decreases with DAIs, contributing to DAIs outperforming ACF.

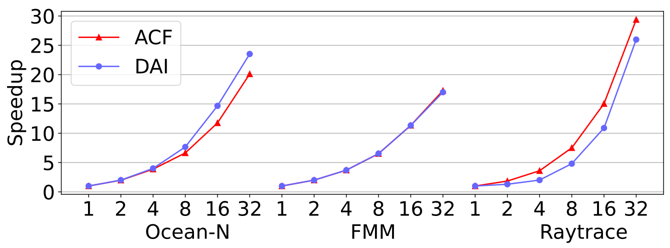


Fig. 7: Performance scalability of ACF and DAI with increasing thread count.

Next, we evaluate the scalability of DAIs vs. ACF, as the number of threads is varied from 1 to 32. The speedup over the respective single-threaded case is plotted in Figure 7, only for benchmarks for which DAIs and ACF performance diverge. For all other benchmarks, the speedup graphs are nearly identical. The figure shows that the scalability of DAIs and ACF are comparable, with DAIs outscaling ACF in `Ocean-Non` and vice versa for *Raytrace*.

Overall, the performance and scalability of our scheme DAIs are comparable to regular atomic instructions with cache line flush and fence (ACF), even though ACF does not provide crash consistency. This striking result demonstrates that DAIs provide crash consistent atomics without sacrificing performance or scalability.

VI. Conclusion

In this paper, we argue the need of durable atomic instructions (DAIs) as low-level primitives that can be used for persistent data synchronization as well as persistent lock-free data structures. We show how DAIs can preserve the semantics of traditional atomic instructions, with the addition of durability. We also discuss approaches to hardware support for DAIs, with our preferred approach requiring only low-cost modifications to the cache coherence protocol. Finally, we show that the performance and scalability of DAIs are comparable to regular atomic instructions with cache line flush and fence (ACF), across benchmarks from the Splash-4 suite.

References

[1] Intel® Optane™ Persistent Memory 200 Series Brief. [Online]. Available:https://investors.micron.com/news-releases/news-release-details/micron-and-intel-announce-update-3d-xpointtm-joint-development [Accessed 08-Sep-2022].

[2] KIOXIA Introduces PCIe 4.0 Storage Class Memory SSDs. [Online]. Available: https://americas.kioxia.com/en-us/business/news/2021/memory-20210913-1.html [Accessed 19-Oct-2022].

[3] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC*, 2001.

[4] M. Friedman, M. Herlihy, V. Marathe, and E. Petrank. A persistent lock-free queue for non-volatile memory. In *ACM SIGPLAN Not.*, 2018.

[5] N. Nguyen and P. Tsigas. Lock-free cuckoo hashing. In *IEEE ICDCS*, 2014.

[6] W. Wang and S. Diestelhorst. Persistent atomics for implementing durable lock-free data structures for non-volatile memory (brief announcement). In *ACM SPAA*, 2019.

[7] M. Liu, M. Zhang, et al. Dudetm: Building durable transactions with decoupling for persistent memory. In *ACM SIGPLAN Not.*, 2017.

[8] V. J. Marathe, M. Pavlovic, et al. Persistent multi-word compare-and-swap, 2020. US Patent 10,678,587.

[9] T. Wang, J. Levandoski, and P. Larson. Easy lock-free indexing in non-volatile memory. In *IEEE ICDE*, 2018.

[10] Intel® 64 and IA-32 Architectures Software Developer's Manual. volume 2a: Instruction set reference. *Order Number: 325383-077US*.

[11] Arm® Architecture Reference Manual for A-profile architecture. [Online]. Available: https://developer.arm.com/documentation/ddi0487/ha/ [Accessed 19-Nov-2022].

[12] Persistent memory development kit (PMDK). [Online]. Available: https://pmem.io/pmdk/ [Accessed 19-Nov-2022].

[13] A. Joshi, V. Nagarajan, et al. Efficient persist barriers for multicores. In *ACM MICRO*, 2015.

[14] Yan Solihin. *Fundamentals of Parallel Multicore Architecture*. CRC Press, 2015.

[15] T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *ACM SC*, 2011.

[16] S. JE Wilton and N. P. Jouppi. CACTI: An enhanced cache access and cycle time model. In *IEEE JSSC*, 1996.

[17] J. Izraelevitz, J. Yang, et al. Basic performance measurements of the intel optane dc persistent memory module. *arXiv:1903.05714 [cs.DC]*, 2019.

[18] E. J. Gómez-Hernández, R. Shao, et al. Splash-4: Improving scalability with lock-free constructs. In *IEEE ISPASS*, 2021.