Hardware Support for Durable Atomic Instructions for Persistent Parallel Programming [1]

Khan Shaikhul Hadi
University of Central Florida
shaikhulhadi@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

Motivation

please click here for the PDF of original paper.

Atomic instructions like *compare and swap* (CAS), *fetch and op* , *atomic exchange*, etc. are immensely useful for higher-level synchronization and for supporting lock-free data structures as they serve at least three primary functions. First, they provide a high-performance alternative to locks . Second, they are foundational building blocks for constructing higher-level synchronization primitives. Finally, they are essential primitives for developing lock-free data structures [2], [3], in particular CAS. However, absence of durable version of atomic instructions makes it challenging to support the three primary functions above for persistent data.

```
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 [4].

Listing 1 [4] illustrate the code snippet for node insertion on a lock-free linked-list. Each *update step* (line 1) performs a CAS followed by a *persist step* (i.e., CLFLUSH and FENCE) on lines 2-3. Figure 1a illustrates 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. 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. In this paper, we propose a new approach: *durable atomic instructions* (DAIs). DAIs are the durable version of atomic instructions, guaranteeing both atomicity and persistency. 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) . Our proposed DAIs require minor hardware modification, no significant application modifications, no crash recovery code, no need for abort and retry, or alternative code to guarantee forward progress while preserve atomic instruction's semantics and add durability/persistency.

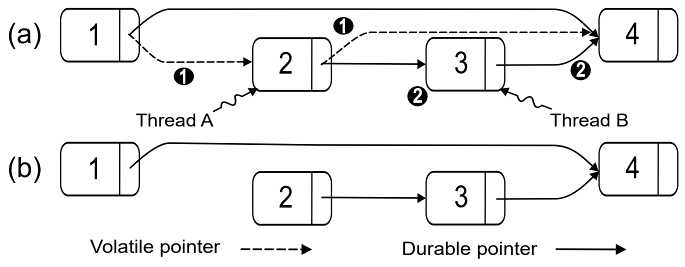

Fig. 1: The problem with the use of current atomic instructions on persistent data. (a) shows concurrent insertions by threads A and B, while (b) shows a possible crash inconsistent state.

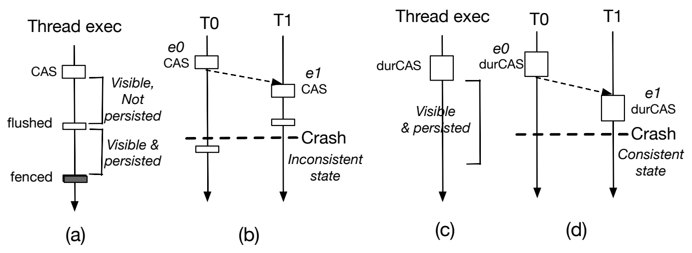

Fig. 2: How traditional compare-and-swap (CAS) violates crash consistency ((a) & (b)), compared to how durable CAS (durCAS) preserves crash consistency ((c) & (d)).

Design of Durable Atomic Instructions (DAIs)

A. Correctness Criteria and Design Principles

Figure 2(a) illustrates the timeline of thread execution using code from listing 1. At the completion of CAS, the updated data is visible but not yet persisted, until CLFLUSH is completed. This could lead crash consistency violation illustrated in Figure 2(b). Thread T1 consumes data in e_1 produced by thread T0 in e_0, resulting dependence relationship as $e_0 \rightarrow e_1$. As stated in [5], persist order must adhere to $e_0 \rightarrow e_1$ as well which cannot be guaranteed with CAS and flush/fence as separate instructions.

DAIs achieve persistence and visibility in a single instruction (Figure 2(c,d)) to ensures durability when data is visible to other core, hence a consistent state is achieved when a crash occurs. To achieve the visible-after-persist property, 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.

After exploring different approaches of atomic instruction implementation, we choose to rely on cache coherence protocol to implement DAIs as it is scalable & require minimum hardware changes. The basic mechanism for atomic instruction is to reserve the involved cache block in private cache of the core that executes the instruction, and refuse intervention/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.

B. Durable MESI (durMESI) Protocol

To illustrate the needed modification, we will discuss a design built on top of the MESI protocol. We refer to our protocol as durMESI. Figure 3 shows process side finite state diagram of 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 notations follow from [6].

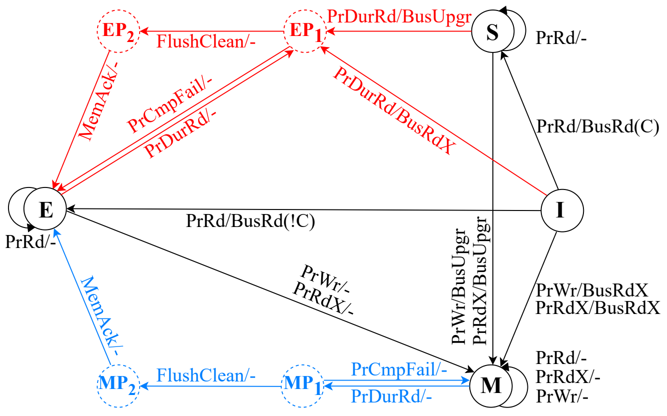

Fig. 3: durable MESI protocol (durMESI). Process Signal.

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 instead of reusing PrRdX, because the final state is different (E vs. M). PrCmpFail is specific to durCAS to indicates that the comparison did not yield a match. The MemAck signal is generated by the persistence domain indicating the receipt of the flushed cache block.

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) which posts BusRdX on the bus, and transitions to EP1. After receiving a response with the data block, read-modify-write is performed in EP1. Then the CC clean-flushes the updated block to the persistence domain and transitions from EP1 to EP2. Upon receiving the flushed block, the persistent domain replies with a MemAck signal so that the CC could conclude that the block has persisted and transitions from EP2 to Exclusive. Note that for durCAS comparison fail, we let the CC know through PrCmpFail, resulting transitions to Exclusive directly, skipping the flushing. If initial state is Modified, PrDurRd trigger transition to MP1 instead of EP1 to prevent transition to Exclusive state upon PrCmpFail as the block may be dirty. While a block is in our proposed transient state, any snoop request will not be served, thus prevents global visibility until durability is guaranteed.

Evaluation

DAI have two advantages over atomic instruction followed by flush and fence (ACF):(1) fewer dynamic instructions and (2) reliance on clean flushing helps keeping data with high locality in the cache. However, DAI locks a cache block for longer time than a regular atomic instruction, which forces other cores wishing to access the same block wait longer time which we refer as cache block contention (CBC).

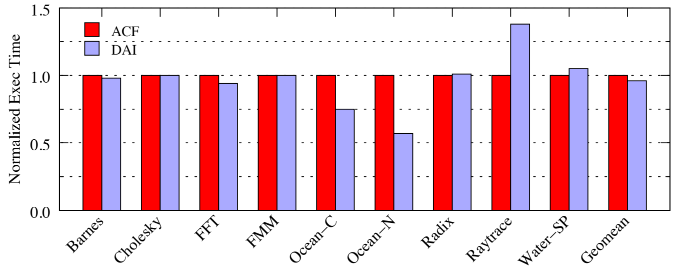

Fig. 4: Execution time of ACF vs. DAI (lower is better), normalized to that of ACF.

Figure 4 shows the execution time of DAI on durMESI, normalized to atomic instruction with ACF for Splash-4 [7] benchmarks on 16 threads. On average, DAIs show 6.4% speedup over ACF, besides achieving crash consistency. For some benchmarks (*Ocean-Cont* and *Ocean-Non*), DAI performs significantly better, but performs significantly worse for *Raytrace* due to significantly higher CBC. Scalability of DAIs and ACF are comparable with DAIs outscaling ACF in *Ocean-Non* and vice versa for *Raytrace*.

References

[1] Khan Shaikhul Hadi, Naveed Ul Mustafa, Mark Heinrich, and Yan Solihin. Hardware support for durable atomic instructions for persistent parallel programming. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2023.

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

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

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

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

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

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