Brief Announcement: Persistent Atomics for Implementing Durable Lock-Free Data Structures for Non-Volatile Memory

William Wang, Stephan Diestelhorst
Arm Research
{william.wang,stephan.diestelhorst}@arm.com

ABSTRACT

This brief announcement presents a persist ordering problem uncovered in implementing durable lock-free data structures for non-volatile memory, and proposes a hardware solution with persistent atomics in the Arm instruction set architecture.

KEYWORDS

lock-free data structure; persistent memory; compare-and-swap

1 INTRODUCTION

Lock-free programming has been widely used towards concurrent data structures, such as linked lists and queues, since it was introduced two decades ago [7, 16, 21]. In-memory databases and message queues heavily rely on lock-free data structures for indexing and queuing [15]. More recently, with the emergence of non-volatile memories such as 3DXP and PCM that promise to displace DRAM due to higher density and non-volatility, various efforts have been taken to *design* lock-free data structures for emerging non-volatile memories [2, 6, 8, 17], as lock-free programming does not incur any logging overhead that's required of lock-based or transaction-based programs for failure atomicity [5]. Meanwhile, several efforts also aimed to *port* existing lock-free data structures to non-volatile memories [9, 10], similar to Aquire-Release Persistency [11, 12]. For instance, durable linearizability [9] was introduced for converting lock-free programs to persistent lock-free programs. In addition, software primitives to construct lock-free programs such as persistent multiple-word compare-and-swap (PMWCAS) have also been proposed to help with build more complex lock-free data structures for non-volatile memories [3, 17, 22], such as doubly linked list and trees. Industrial associations, such as SNIA and HMC, have also come forward with standards that include support for atomics with persistent memory [4, 18, 19].

2 THE PROBLEM

To make lock-free data structures work with *persistent memory*, an atomic compare-and-swap (CAS) operation is no longer sufficient, a cacheline flush operation followed by a synchronization barrier is needed to make sure the data is persisted to the point-of-persistence to synchronize the concurrent view with the recovery view for crash consistency. As shown in Listing 1, swinging a pointer atomically by CAS then followed by a cacheline flush and a synchronization fence is needed to publish a new node to a lock-free singly linked list.

Listing 1: A compare-and-swap is followed by a cacheline flush and a fence

```
1  if(CAS(&last->next, next, node)) {
2    FLUSH(&last->next);
3    FENCE;
4  }
```

Arm v8.1 [14] introduces compare-and-swap atomics. In Arm v8.0 [13] and prior versions, atomic compare-and-swap can be implemented with *load-linked* and *store-conditional* pairs, i.e., *LDXR* and *STXR*. As CAS and FLUSH are two separate operations, i.e., non-atomic, interrupts can happen in between and inconsistencies can arise due to write-after-read dependencies, where a thread persists a new value computed as the result of reading a value that might not have been persisted.

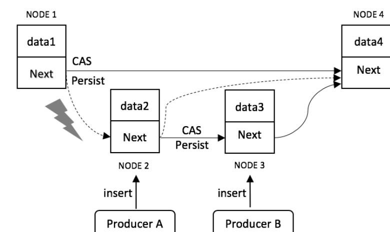

Figure 1: Two threads A and B insert a node each to a lock-free linked list.

For instance, as illustrated in Figure 1, Node 2 is inserted by Producer A thread after Node 1, the *Next* pointer of Node 1 gets atomically switched (CAS) to point to Node 2 from Node 4, however, this CAS is not persisted, i.e., code execution gets interrupted after Line 1 but before Line 2 as shown in Listing 1. Then, Node 3 gets inserted by Producer B thread after Node 2 as CAS for Node 2 is already visible to all concurrent threads. CAS atomically switches the *Next* pointer of Node 2 to point to Node 3 from Node 4, and this CAS gets persisted. If a power failure happens at this point before Line 2 get executed for Producer A, the application that uses the linked list would be left in an inconsistent state, with both Node 2 and Node 3 lost, as the *Next* pointer from Node 1 to Node 2 has not been persisted. As Node 3 has been published but can't be accessed after a reboot, and other data may have been persisted that are accessed through or dependent on Node 3, all subsequent accesses to such data will not be possible, therefore causing data loss that should have never happened.

The problem described above is particular to lock-free linked lists with CAS for *persistent memory*, in addition to two other well-known problems for lock-free linked lists with CAS. Valois [20] described both problems with solutions for lock-free linked lists using compare-and-swap: a) The concurrent insertion and deletion problem, where both nodes can be lost, similar to the problem

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
SPAA '19, June 22–24, 2019, Phoenix, AZ, USA
© 2019 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-6184-2/19/06.
https://doi.org/10.1145/3323165.3323166

above; and b) The concurrent deletions of adjacent nodes problem, where the second deletion can be undone. Harris [7] also proposed an improved pragmatic solution to the first problem.

The *persist ordering* problem is not limited to lock-free data structures alone, the problem can manifest with large scale databases, such as SAP HANA[1], while interacting with operating systems. Linux filesystems often creates files with holes by default, storage is physically allocated only at the time of actual writes to the file. For *mmap*'ed files, not all memories are allocated at *mmap* call time, i.e., demand paging. When a userspace application does a store to a location in an mmaped DAX file with a 'hole', the store causes a page fault and the file system fills the hole with a new memory allocation. The filesystem metadata that gets updated by that fault may not be flushed to persistence even though userspace stores are flushed and persisted, if a power failure strikes before the metadata changes are persisted, the application will be left in an inconsistent state. As a result, the recovered application, i.e., SAP HANA, can crash upon accesses to the data stored onto the lost pages, as the stores to such pages would have been lost even though the flushes had been performed.

The problem can be formulated as follows:

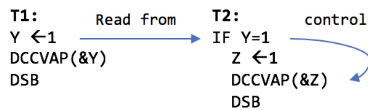

Figure 2: Inconsistencies can arise due to write-after-read dependencies, where a thread T2 persists a new value computed as the result of reading a value from T1 that might not have been persisted.

where the inconsistent state with `Z = 1` and `Y = 0` is possible following a power failure. `Y` and `Z` are both initialized to zero at start. T1 can't guarantee to execute `DCCVAP(&Y)` and `DSB` atomically with `Y=1`, i.e., `DCCVAP` and `DSB` can get executed after T2 has completed.

3 SOLUTIONS

3.1 Software Solutions

There are software solutions that can overcome the problem, such as using metadata, i.e., extra communication to indicate whether a line has been persisted or not between different threads [17, 22]. However, they suffer from overloading reads and come at a high cost, i.e., ~10% overhead as in [22]. There have been proposals to make compare-and-swaps recoverable as well with logging [3].

A software solution to prevent this from happening can be using metadata to indicate whether `Y` has been persisted, and a) making the normal read '`IF Y=1`' into a persist read, i.e., if `Y` has not been persisted, persist `Y` first and then return the value of `Y` [22], or b) looping around read '`IF (Y=1)&&Persisted`' to wait until `Y` is persisted. This would overload all reads, all the reads would pay the cost of checking whether the line has been persisted. Instead of overloading reads with the burden to persist dirty data, and as reads are much more frequent than writes, we propose to shift the burden to writes, to make compare-and-swaps atomic with persists.

3.2 Persistent Stores

Here we describe the range of instructions proposed to Arm instruction set architecture [13, 14] to make such stores persistent, including store exclusives, store releases, and atomics.

1. Persistent store exclusives, which can be used to build persistent atomic operations together with load exclusives.
   - PST[L]XR[B|H], PST[L]XP
2. Persistent store releases, i.e., stores with ordering semantics, combine store and persist for acquire-release persistency.
   - PSTLR[B|H]
3. Persistent atomics, which can be used to build lock-free data structures.
   - PCAS[P][A|L|AL][B|H]
   - PLD<OP>[A|L|AL][B|H] where OP = ADD, CLR, EOR, SET, SMAX, SMIN, UMAX, UMIN
   - PSWP[A|L|AL][B|H]

where the leading P stands for Persistent. A stands for Acquire (half barrier, order after), L stands for Release (half barrier, order before), AL stands for Acquire-Release (full barrier, order both before and after). B stands for Byte, H stands for Half-word, P stands for Pair of words, i.e., granularity of operation.

The proposed instructions would fix the problem introduced in the transformations as described in [9, 10] for durable linearizability and in [11, 12] for language-level acquire-release persistency.

3.3 Persistent Lock-free Queue

The proposed persistent compare-and-swap instruction can simplify porting lock-free data structures to persistent memory by simply replacing CAS with PCAS, eliminating the need to FLUSH and FENCE as shown in Listing 1. For instance, the *enqueue* operation of the durable queue as in [6] can be simplified as follows.

Listing 2: Implementation of a durable lock-free queue with PCAS

```
1  void enqueue(T value){
2    Node* node = new Node(value); FLUSH(node);
3    while(true){
4      Node* last = tail;
5      Node* next = last->next;
6      if(last == tail){
7        if(next == NULL){
8          if(PCAS(&last->next, next, node)){
9            CAS(&tail, last, node); return; }
10       } else {
11         CAS(&tail, last, next) } } } }
```

4 CONCLUSIONS AND FUTURE WORK

The persist ordering problem as observed with durable lock-free data structures for non-volatile memory is described, and a hardware solution that involves new persistent store instructions is proposed. The problem exists in userspace such as durable lock-free data structures, the interactions between user space and kernel space such as SAP HANA interacting with OS demand paging, as well as transformations as proposed for durable linearizability and acquire-release persistency. The proposed instructions would eliminate the problem for all such cases on Arm, and the persist ordering problem is not limited to the Arm instruction set architecture.

We're conscious that our hardware-based solution shifts the burden from software developers to CPU designers, which carries a significant implementation cost. As future work, we'll quantitatively evaluate the cost of software-based solutions as compared to the cost of the hardware-based solutions as proposed in this brief announcement.

REFERENCES

[1] Mihnea Andrei, Christian Lemke, Günter Radestock, Robert Schulze, Carsten Thiel, Rolando Blanco, Akanksha Meghlan, Muhammad Sharique, Sebastian Seifert, Surendra Vishnoi, et al. 2017. SAP HANA adoption of non-volatile memory. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1754–1765.

[2] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. 2018. Bztree: A high-performance latch-free range index for non-volatile memory. *Proceedings of the VLDB Endowment* 11, 5 (2018), 553–565.

[3] Naama Ben-David, Guy E. Blelloch, and Yuanhao Wei. 2018. Making Concurrent Algorithms Detectable. *CoRR* abs/1806.04780 (2018). arXiv:1806.04780 http://arxiv.org/abs/1806.04780

[4] HMC Consortium. 2015. *Hybrid Memory Cube Specification 2.1.* Technical Report.

[5] Tudor David, Aleksandar DragojevicÌĄ, Rachid Guerraoui, and Mihail Igor Zablotchi. 2017. *Log-Free Concurrent Data Structures.* Technical Report.

[6] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. 2018. A persistent lock-free queue for non-volatile memory. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM, 28–40.

[7] Timothy L Harris. 2001. A pragmatic implementation of non-blocking linked-lists. In *International Symposium on Distributed Computing.* Springer, 300–314.

[8] Yihe Huang, Matej Pavlovic, Virendra Marathe, Margo Seltzer, Tim Harris, and Steve Byan. 2018. Closing the performance gap between volatile and persistent key-value stores using cross-referencing logs. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18).* USENIX Association, 967–979.

[9] Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. 2016. Brief announcement: Preserving happens-before in persistent memory. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures.* ACM, 157–159.

[10] Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. 2016. Linearizability of persistent memory objects under a full-system-crash failure model. In *International Symposium on Distributed Computing.* Springer, 313–327.

[11] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M Chen, Satish Narayanasamy, and Thomas F Wenisch. 2017. Language-level persistency. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on.* IEEE, 481–493.

[12] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M Chen, Satish Narayanasamy, and Thomas F Wenisch. 2017. Tarp: Translating acquire-release persistency.

[13] Arm Ltd. 2016. *ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile.* Technical Report. https://developer.arm.com/docs/ddi0487/a/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile

[14] Arm Ltd. 2016. *ARM Architecture Reference Manual Supplement ARMv8.1, for ARMv8-A architecture profile Documentation.* Technical Report. https://developer.arm.com/docs/ddi0557/latest

[15] Darko Makreshanski, Justin Levandoski, and Ryan Stutsman. 2015. To lock, swap, or elide: On the interplay of hardware transactional memory and lock-free indexing. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1298–1309.

[16] Maged M Michael and Michael L Scott. 1995. *Simple, fast, and practical non-blocking and blocking concurrent queue algorithms.* Technical Report. ROCHESTER UNIV NY DEPT OF COMPUTER SCIENCE.

[17] Matej Pavlovic, Alex Kogan, Virendra J Marathe, and Tim Harris. 2018. Brief Announcement: Persistent Multi-Word Compare-and-Swap. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing.* ACM, 37–39.

[18] SNIA. 2017. *Persistent Memory Atomics and Transactions v1.2.* Technical Report.

[19] SNIA. 2019. *NVM PM Remote Access for High Availability v1.08.* Technical Report.

[20] John D Valois. 1995. Lock-free linked lists using compare-and-swap. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing.* ACM, 214–222.

[21] John David Valois. 1996. Lock-free data structures. (1996).

[22] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. 2018. Easy lock-free indexing in non-volatile memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE).* IEEE, 461–472.