Brief Announcement: Persistent Multi-Word Compare-and-Swap

Matej Pavlovic*
EPFL
matej.pavlovic@epfl.ch

Alex Kogan
Oracle Labs
alex.kogan@oracle.com

Virendra J. Marathe
Oracle Labs
virendra.marathe@oracle.com

Tim Harris*
tim.harris@gmail.com

ABSTRACT

This brief announcement presents a fundamental concurrent primitive for persistent memory – a persistent atomic multi-word compare-and-swap (PMCAS). We present a novel algorithm carefully crafted to ensure that atomic updates to a multitude of words modified by the PMCAS are persisted correctly. Our algorithm leverages hardware transactional memory (HTM) for concurrency control, and has a total of 3 persist barriers in its critical path. We also overview variants based on just the compare-and-swap (CAS) instruction and a hybrid of CAS and HTM.

ACM Reference Format:
Matej Pavlovic, Alex Kogan, Virendra J. Marathe, and Tim Harris*. 2018. Brief Announcement: Persistent Multi-Word Compare-and-Swap. In *PODC '18: ACM Symposium on Principles of Distributed Computing, July 23–27, 2018, Egham, United Kingdom.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3212734.3212783

1 INTRODUCTION

Emerging persistent memory (PM) technologies such as Intel and Micron's 3D XPoint [1] will offer the persistence of traditional storage technologies (NAND flash and disks), DRAM-like byte-addressability, and performance approaching that of DRAM (100-1000x faster than state-of-the-art NAND flash). The combination of these features of PM could profoundly affect the way we manage persistent data in modern applications. Traditionally, DRAM-resident data structures can be hosted in persistent memory. However, this task is complicated by the fact that the processor cache hierarchy will remain *nonpersistent* in the foreseeable future. Processor vendors, such as Intel [10], have proposed new hardware instructions to order persistence of stores to PM.

With the new instructions (flush/writeback cache lines, persist barriers), programmers are expected to rebuild applications to leverage PM. This is, however, a non-trivial endeavor. The principal challenge in building such algorithms centers around the fact that the caching layers in processor memory hierarchy are not persistent, and cache lines can be evicted (thus, persisted) in an arbitrary order. Algorithms must carefully order persistence of stores, using the aforementioned instructions, for correct recovery after failures.

Recognizing these programming challenges, several researchers and practitioners have proposed the use of various forms of transactions to access and manipulate data in persistent memory [4, 5, 16]. While transactions are a powerful abstraction to program PM, they incur significant performance overheads, which is why the field continues to be a subject of active research [12, 13]. In addition, more efficient concurrent data structure implementations can be built without transactions [6] or with a simple multi-word compare-and-swap (MCAS) primitive [7, 8, 14].

Wang et al. [17] recently proposed a lock-free persistent multi-word compare-and-swap (PMCAS) based on Harris et al's. MCAS algorithm [8]. Although an important advance, the algorithm is complex and suffers from significant performance overheads – for every PMCAS, it uses 5 CASes and 2 persist barriers per modified word in the critical path. In this brief announcement, we present new *durably linearizable* [2, 11] variants of a persistent multi-word compare-and-swap (PMCAS) primitive, where multiple words in PM can be updated atomically and in a crash-tolerant manner. Our algorithm, though blocking, requires just 1 CAS per modified word and 3 persist barriers in total in the critical path.

2 PERSISTENT MCAS ALGORITHM

We assume a conventional shared memory multicore system with deterministic threads that read and write shared memory using load, store, and CAS instructions as well as HTM transactions. We assume a fail-stop failure model, where a failure crashes all threads accessing the shared memory. The system can contain both persistent and nonpersistent (DRAM) memory. Processor buffers and caches are nonpersistent.

For simplicity, we focus on PMCAS-htm, a PMCAS variant that leverages the hardware transactional memory (HTM) feature available in some of the contemporary processors [9, 15]. In this variant, reads from memory locations being updated by a concurrent PMCAS block and wait for the PMCAS to complete. Toward the end, we briefly mention nonblocking reads, and two variants of PMCAS, one that uses the compare-and-swap (CAS) instruction instead of HTM, and another that uses both HTM and CAS.

For each PMCAS, the application creates a persistent *update structure* that holds old and new values of the target addresses. We assume that the application persistently tracks all these structures in use by potentially multiple threads (e.g. by having a persistent pool of update structures reachable from a designated "root" of a persistent region). These structures must be reachable from the recovery subsystem.

*Author was employeed at Oracle Labs during this work.

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).
PODC '18, July 23–27, 2018, Egham, United Kingdom
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-5795-1/18/07.
https://doi.org/10.1145/3212734.3212783

In a nutshell, PMCAS-htm uses a HTM transaction to acquire "ownership" of each target address. It then applies (and persists) the updates on the owned addresses, thereby releasing the ownership. This trivial approach turns out to be surprisingly tricky when we consider (fail-stop) failures at arbitrary points in the algorithm: How can recovery determine if an operation was already applied or failed? How can recovery determine the state of a partially persisted PMCAS-htm and then complete or rollback that state?

PMCAS-htm's pseudocode appears in Figure 1. Our main algorithm begins at line 50. It uses the least significant bit of each target word to represent a "transitory" state of the address being modified; we assume that this bit is not used by the application. In addition to the target addresses, their expected old values, and the new values, PMCAS-htm takes an *update structure* as its argument. Internally, the update structure itself contains a status field and a set of M *update records* (M can vary between different PMCAS-htms), each of which contains a target address, expected old value, and the new value to be applied in the PMCAS (lines 1 – 15). First, the update structure is updated with addresses, old and new values. These updates are persisted before proceeding with the PMCAS (lines 55 – 61). We note that depending on the actual programming interface, this initialization of the update structure (including the persist barrier in line 61) might be done by the application code invoking PMCAS.

PMCAS-htm goes through two states during its execution. It always begins with the ACTIVE state, and eventually switches over to either the SUCCESS or the FAILURE state, denoting success or failure, respectively, of the PMCAS-htm operation. The state transitions must also be correctly persisted for recovery to determine the action to perform on in-flight PMCAS-htm operations – roll forward or roll back.

After initializing and persisting the update structure, PMCAS-htm uses a HTM transaction[1] (the atomic block at lines 70 – 80) to first check if all targets have the expected values. In the same transaction, if the check succeeds, we acquire exclusive ownership of the target addresses by installing a "marked" pointer pointing to the PMCAS-htm's update structure in the target addresses. Ownership acquisition success is indicated by a local committed flag. On success, the ownership acquisitions are all persisted (lines 83 – 86), the status of PMCAS-htm is updated to SUCCESS and persisted. Finally, new values are written back to the targets and persisted (line 92 and 28 – 37). If the PMCAS-htm operation fails, the status is updated to FAILURE and persisted (lines 94 – 96). No roll back is needed since ownership acquisition of the target addresses did not happen on the failure path (line 73).

Recovery proceeds as follows: For all the updates supplied by the application to the PMCAS-htm recovery subsystem, it rolls forward the PMCAS-htm operations whose status is SUCCESS, and rolls back the rest. Rolling forward simply requires calling write_new_values() for the PMCAS-htm operation. Rolling backward applies to only the ACTIVE PMCAS-htm operations since their partial updates may have persisted (between lines 80 and 89). Rolling back such a PMCAS-htm operation may sound like a conservative approach, however it is correct and keeps recovery simple, logically serializing the crash before the rolled back PMCAS-htm operations.

PMCAS variants: PMCAS-htm blocks any readers from reading an address that is owned by a concurrent updater (lines 18 – 26). However, we can avoid blocking by accessing the old value through the marked pointer written by the updater. This optimization however requires a lazy reclamation or garbage collection based scheme for persistent memory management [3]. The HTM transaction can be avoided using a loop that uses CAS instructions to acquire ownership of target addresses. We must ensure that the target addresses are ordered in a global total order to avoid conflicts between concurrent updaters resulting in *spurious* failures of PMCAS-htms. A HTM-CAS hybrid algorithm would attempt to acquire ownerships with a transaction and fall back to the CAS version if the hardware transaction fails to commit. In all these variants, the recovery algorithm remains unchanged.

This work is ongoing, and we plan to evaluate these algorithms going forward. Our future work will also focus on developing *lock-free* variants of this algorithm.

3 CONCLUSION

We presented a simple PMCAS algorithm that guarantees failure-atomic updates to multiple words resident in persistent memory. Our algorithm is simpler than the state-of-the-art lock-free algorithm by Wang et al. [17], and far more efficient in terms of the number of CASes and persist barriers needed per PMCAS. Our algorithm, however, is blocking, and in future work we will explore its lock-free variants.

REFERENCES

[1] 3dxpoint 2015. 3D XPoint Technology Revolutionizes Storage Memory. http://www.intel.com/content/www/us/en/architecture-and-technology/3d-xpoint-technology-animation.html.

[2] Ryan Berryhill, Wojciech M. Golab, and Mahesh Tripunitara. 2015. Robust Shared Objects for Non-Volatile Main Memory. In *19th International Conference on Principles of Distributed Systems*. 20:1–20:17.

[3] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-Juergen Boehm. 2016. Makalu: fast recoverable allocation of non-volatile memory. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 677–694.

[4] Dhruva R. Chakrabarti, Hans-Juergen Boehm, and Kumud Bhandari. 2014. Atlas: leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*. 433–452. https://doi.org/10.1145/2660193.2660224

[5] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. 105–118.

[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*. 28–40.

[7] Michael Greenwald. 2002. Two-handed emulation: how to build non-blocking implementation of complex data-structures using DCAS. In *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Distributed Computing*. 260–269.

[8] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. 2002. A Practical Multi-word Compare-and-Swap Operation. In *Distributed Computing, 16th International Conference*. 265–279.

[9] IBM. [n. d.]. AIX transactional memory programming. *https://www.ibm.com/support/knowledgecenter/en/ssw_aix_71/com.ibm.aix.genprogc/transactional_memory.htm*.

[10] intel-isa 2015. Intel Architecture Instruction Set Extensions Programming Reference. *https://software.intel.com/sites/default/files/managed/0d/53/319433-022.pdf*.

[1] We assume that all hardware transactions will eventually succeed. This can be enforced with a hybrid TM runtime system with progress guarantees; e.g., one that uses a global lock after several failed transactions.

```
1   struct UpdateRecord {
2     uint64_t *ad;
3     uint64_t old_val;
4     uint64_t new_val;
5   };
6
7   enum {
8     ACTIVE = 0,
9     SUCCESS,
10    FAILURE
11  } State;
12
13  __thread struct Upd {
14    State status;
15    UpdateRecord ur[M];
16  } my_upd;
17
18  uint64_t read(
19    uint64_t *ad){
20
21    uint64_t val = *ad;
22    while(is_marked(val)) {
23        val = *ad;
24    }
25    return val;
26  }
27
28  void write_new_values(Upd *u) {
29    for(int i = 0; i < M; i++) {
30      if(unmark(*u->ur[i].ad) == u){
31        *u->ur[i].ad
32          = u->ur[i].new_val;
33        flush(u->ur[i].ad);
34      }
35    }
36    persist_barrier();
37  }
38
39  void restore_old_values(Upd *u) {
40    for(int i = 0; i < M; i++) {
41      if(unmark(*u->ur[i].ad) == u){
42        *u->ur[i].ad
43          = u->ur[i].old_val;
44        flush(u->ur[i].ad);
45      }
46    }
47    persist_barrier();
48  }
49
50  State PMCAS-htm(Upd &my_upd,
51                  uint64_t* ad[],
52                  uint64_t old[],
53                  uint64_t new[]) {
54    // Assume my_upd.status == ACTIVE
55    for(int i = 0; i < M; i++) {
56      my_upd.ur[i].ad = ad[i];
57      my_upd.ur[i].old_val = old[i];
58      my_upd.ur[i].new_val = new[i];
59    }
60    flush(&my_upd);
61    persist_barrier();
62
63    return PMCAS-htm-run(my_upd);
64  }
65
66  State PMCAS-htm-run(
67    Upd &my_upd) {
68
69    bool committed = false;
70    atomic {
71      for(int i = 0; i < M; i++) {
72        if(read(ad[i]) != old[i]) {
73          commit_tx();
74        }
75      }
76      for(int i = 0; i < M; i++) {
77        *ad[i] = mark(&my_upd);
78      }
79      committed = true;
80    }
81
82    if(committed) {
83      for(int i = 0; i < M; i++) {
84        flush(ad[i]);
85      }
86      persist_barrier();
87
88      my_upd.status = SUCCESS;
89      flush(&my_upd.status);
90      persist_barrier();
91
92      write_new_values(&my_upd);
93    } else {
94      my_upd.status = FAILURE;
95      flush(&my_upd.status);
96      persist_barrier();
97    }
98
99    return my_upd.status;
100 }
101
102 void recover() {
103   for(Upd *upd : all_updates) {
104     if (upd->status == SUCCESS) {
105       write_new_values(upd);
106     } else
107     if (upd->status == ACTIVE) {
108       restore_old_values(upd);
109     }
110   }
111 }
```

Figure 1: PMCAS-htm: Persistent MCAS using HTM

[11] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Distributed Computing - 30th International Symposium*. 313–327.

[12] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. 329–343.

[13] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. 2017. Atomic In-place Updates for Non-volatile Main Memories with Kamino-Tx. In *Proceedings of the Twelfth European Conference on Computer Systems*. 499–512.

[14] Faisal Nawab, Dhruva R. Chakrabarti, Terence Kelly, and Charles B. Morrey III. 2015. Procrastination Beats Prevention: Timely Sufficient Persistence for Efficient Crash Resilience. In *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015*. 689–694.

[15] James Reinders. 2012. Transactional Synchronization in Haswell. *https://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell*.

[16] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*. 91–104.

[17] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. 2018. Easy Lock-Free Indexing in Non-Volatile Memory. In *34th IEEE International Conference on Data Engineering*.