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

Matej Pavlovic*[1], Alex Kogan†[2], Virendra J. Marathe‡[2], and Tim Harris§[2]

[1]EPFL
[2]Oracle Labs

Abstract

Emerging persistent memory technologies such as Intel and Micron's 3D XPoint are on the horizon. The combination of byte addressability, performance, and persistence these technologies promise to deliver has the potential to significantly affect the way we manage persistent data in future applications. Foundational concurrent data structures and primitives may need to be revisited with persistence in mind. While traditional concurrent algorithms largely need to address the problem of correct synchronization between concurrent threads, persistent memory brings the additional challenge of making sure that updates to persistent memory are persisted in the order that enables correct recovery in the face of failures (we consider fail–stop failure semantics).

This brief announcement presents a fundamental concurrent primitive – a persistent 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 in the correct order. Our algorithm leverages hardware transactional memory (HTM) for concurrency control, and 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.

*matej.pavlovic@epfl.ch: Author was interning at Oracle Labs during this work.
†alex.kogan@oracle.com
‡virendra.marathe@oracle.com
§tim.harris@gmail.com: Author was employed at Oracle Labs during this work.

1 Introduction

Emerging persistent memory (PM) technologies such as *spin-transfer torque MRAM (STT-MRAM)* [11, 12], *memristors* [24], and Intel and Micron's 3D XPoint [1] will guarantee the persistence of traditional storage technologies (NAND flash and disks) and are expected to come close to the performance of DRAM (100-1000x faster than state-of-the-art NAND flash). More noteably, these technologies will be packaged in conventional DRAM form factor and deliver the byte-addressability of DRAM which would enable a simple load/store to storage. The implications of these features of PM could profoundly affect the way we manage persistent data in modern applications.

The load/store interface to persistent memory is however not sufficient since the processor state and various layers in the memory hierarchy (viz. store buffers, caches) are expected to remain *nonpersistent* in the foreseeable future. Applications need better primitives to control when data moves through the memory hierarchy layers to PM. To that end, prior research [6, 16, 21] and processor vendors such as Intel have proposed new hardware instructions [14] to flush or write back cache lines to lower layers in the memory hierarchy (i.e., memory controller buffers), and new forms of persistence ordering barrier instructions that can be used by programmers to ensure that prior stores to persistent memory are persisted before subsequent stores. With these new instructions, programmers are expected to rebuild applications to leverage PM. This is, however, a non-trivial challenge.

Recognizing these programming challenges, several researchers and practitioners have proposed use of various forms of transactions to access and manipulate data in persistent memory [3, 4, 5, 8, 17, 22, 25]. 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 [18, 19]. In addition, more efficient concurrent data structure implementations can be built without transactions [7] or with a simple multi-word compare-and-swap (MCAS) primitive [9, 10, 20]. The principal challenge in building such algorithms centers around the fact that the caching layers in processor memory hierarchy are not persistent, and programmers cannot control the order in which stores to cache lines are evicted, thereby becoming persistent. Algorithms must carefully order persistence of stores to enable correct recovery in the face of failures. This brief announcement presents new *durably linearizable* [15] 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.

2 Persistent MCAS Algorithm

In this section we describe in detail our algorithm for the PMCAS primitive. For simplicity and due to space restrictions, we focus on PMCAS-htm, a PMCAS variant that leverages the hardware transactional memory (HTM) feature available in some of the contemporary processors [13, 23]. We briefly mention nonblocking reads, and compare-and-swap (CAS) and CASHTM hybrid based variants at the end.

For each PMCAS operation, the application creates a persistent *update structure* that holds the old and new values of the target addresses. We assume that the application persistently tracks all these structures that are in use by potentially multiple threads (e.g. by having a persistent pool of update structures reachable from a "root" object located in a persistent region). These structures must be reachable from the recovery code to correctly recover all in-flight PMCAS-htms after a failure.

In a nutshell, PMCAS-htm uses a HTM transaction to acquire "ownership" of each address

```
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 htm_PMCAS(Upd &my_upd,
51                  uint64_t* ad[],
52                  uint64_t old[],
53                  uint64_t new[]) {
54
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 htm_PMCAS_run(my_upd);
64  }
65
66  State htm_PMCAS_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();
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

being updated. The operation then applies (and persists) the updates on the owned addresses, thereby releasing the ownership. This straightforward approach turns out to be surprisingly tricky when we consider (fail-stop) failures at arbitray 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?

The C language style pseudocode of the full PMCAS-htm algorithm is shown 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; as an effect, 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, the PMCAS-htm operation takes an *update structure* as its argument. This update structure is used to correctly apply PMCAS-htm. Internally, the update 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 $1-15$). The update structure is the first to be updated with addresses, old and new values. These updates are persisted before proceeding with the PMCAS (lines $55-63$).

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. The state transitions must also be correctly persisted for recovery to determine the action needed to recover a PMCAS-htm – roll forward or roll back.

After initializing and persisting the update structure, PMCAS-htm uses a hardware transaction (represented by the atomic block at lines $70-80$) to first check if all the target addresses 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 update structure of the PMCAS-htm in the target addresses. The ownership acquisition success is indicated by a local committed flag, which is subsequently used to determine what state the PMCAS-htm transitions to. On success, the ownership acquisitions are all persisted (lines $83-86$), the PMCAS-htm' status is updated to SUCCESS and persisted. Finally, the new values are written back to the target addresses and persisted (line 92 and $28-37$). In case of failure, the PMCAS-htm's 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-htms whose status is SUCCESS, and rolls back the rest. Rolling forward simply requires calling write_new_values() for the PMCAS-htm. Rolling backward applies to only the ACTIVE PMCAS-htmes since their partial updates may have persisted (between lines 80 and 89). Rolling back such a PMCAS-htm may sound like a conservative approach, however it is correct and keeps recovery simple, logically serializing the crash before the rolled back PMCAS-htms.

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 [2]. The HTM transaction can be avoided using a loop that uses CAS instructions to acquire ownership of the 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 in future work. Our future work will also focus on developing *lock-free* variants of this algorithm.

References

[1] 3D XPoint Technology Revolutionizes Storage Memory.

http://www.intel.com/content/www/us/en/architecture-and-technology/3d-xpoint-technology-animation.html, 2015.

[2] K. Bhandari, D. R. Chakrabarti, and H. Boehm. 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*, pages 677–694, 2016.

[3] B. Bridge. Nvm-direct library. *https://github.com/oracle/nvm-direct*, 2015.

[4] D. R. Chakrabarti, H. Boehm, and K. Bhandari. 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*, pages 433–452, 2014.

[5] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. 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*, pages 105–118, 2011.

[6] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 133–146, 2009.

[7] M. Friedman, M. Herlihy, V. Marathe, and E. Petrank. A persistent lock-free queue for non-volatile memory. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 28–40, 2018.

[8] E. Giles, K. Doshi, and P. J. Varman. Softwrap: A lightweight framework for transactional support of storage class memory. In *IEEE 31st Symposium on Mass Storage Systems and Technologies, MSST 2015, Santa Clara, CA, USA, May 30 - June 5, 2015*, pages 1–14, 2015.

[9] M. Greenwald. 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*, pages 260–269, 2002.

[10] T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *Distributed Computing, 16th International Conference*, pages 265–279, 2002.

[11] M. Hosomi, H. Yamagishi, T. Yamamoto, K. Bessho, Y. Higo, K. Yamane, H. Yamada, M. Shoji, H. Hachino, C. Fukumoto, H. Nagao, and H. Kano. A novel nonvolatile memory with spin torque transfer magnetization switching: Spin-RAM. *International Electron Devices Meeting*, pages 459–462, 2005.

[12] Y. Huai. Spin-Transfer Torque MRAM (STT-MRAM): Challenges and Prospects. *AAPPS Bulletin*, 18(6):33–40, 2008.

[13] IBM. AIX transactional memory programming. *https://www.ibm.com/support/knowledgecenter/en/ssw_aix aix.genprogc/transactional_memory.htm.*

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

[15] J. Izraelevitz, H. Mendes, and M. L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *Distributed Computing - 30th International Symposium*, pages 313–327, 2016.

[16] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas. Efficient persist barriers for multicores. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 660–671, 2015.

[17] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch. High-Performance Transactions for Persistent Memories. In *21th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[18] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren. 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*, pages 329–343, 2017.

[19] A. Memaripour, A. Badam, A. Phanishayee, Y. Zhou, R. Alagappan, K. Strauss, and S. Swanson. Atomic in-place updates for non-volatile main memories with kamino-tx. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 499–512, 2017.

[20] F. Nawab, D. R. Chakrabarti, T. Kelly, and C. B. M. III. Procrastination beats prevention: Timely sufficient persistence for efficient crash resilience. In *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015*, pages 689–694, 2015.

[21] S. Pelley, P. M. Chen, and T. F. Wenisch. Memory persistency. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, pages 265–276, 2014.

[22] pmem.io: Persistent Memory Programming. *http://pmem.io/*, 2015.

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

[24] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing Memristor found. *Nature*, 453:80–83, 2008.

[25] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 91–104, 2011.