Brief Announcement: Preserving Happens-before in Persistent Memory *

Joseph Izraelevitz Hammurabi Mendes Michael L. Scott

University of Rochester, Rochester, NY, USA
{jhi1,hmendes,scott}@cs.rochester.edu

ABSTRACT

Nonvolatile, byte-addressable memory (NVM) will soon be commercially available, but registers and caches are expected to remain transient on most machines. Without careful management, the data preserved in the wake of a crash are likely to be inconsistent and thus unusable.

Previous work has explored the semantics of instructions used to push the contents of cache to NVM. These semantics comprise a “memory persistency model,” analogous to a traditional “memory consistency model.” In this brief announcement we introduce *explicit epoch persistency*, a memory persistency model that captures the current and expected semantics of Intel x86 and ARM v8 persistent memory instructions. We also present a construction that augments any data-race-free program (for release consistency or any stronger memory model) in such a way that preserved data are guaranteed to represent a consistent cut in the happens-before graph of the program’s execution.

1. INTRODUCTION

Nonvolatile, byte-addressable memory (NVM) is expected to be common in coming years. Caches and registers, however, are likely to remain in SRAM for some time to come. While NVM offers the opportunity to keep persistent data in main memory (not just in the file system), the fact that recent updates to registers and cache may be lost during a power failure means that the data in main memory, if not carefully managed, may not be consistent at recovery time.

Maintaining a consistent state in NVM requires special care to order main memory updates. Several libraries have been designed to support transactional updates of persistent state [3, 7, 12]. Similarly, some high performance data structures [11, 13] carefully control concurrent updates to ensure that the persistent portion of a data structure remains consistent.

*This work was supported in part by NSF grants CNS-1319417, CCF-1337224, and CCF-1422649, and by support from the IBM Canada Centres for Advanced Study.

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’16, July 11–13, 2016, Pacific Grove, CA, USA.

© 2016 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-4210-0/16/07.

DOI: <http://dx.doi.org/10.1145/2935764.2935810>

Concurrent and crash-resilient programs are closely connected: both must ensure that state can safely be read concurrently—either by a “real” thread during normal execution or by a conceptual “recovery thread” [10] that sees the post-crash state. By leveraging this connection, we can ensure that the happens-before ordering of the concurrent program is seen by the recovery thread.

Specifically, we present a construction that augments any properly synchronized concurrent program in a way that forces stores to be written back to NVM in an order consistent with happens-before. Our construction assumes that the input program is data-race free under release consistency or some stronger memory model. It further assumes that the hardware supports what we call *explicit epoch persistency*, a relaxed memory persistency model [10] that captures current and forthcoming processors in the Intel x86 and ARM v8 product lines. As output, our construction produces a program that guarantees that the contents of NVM in the wake of a crash will always represent a state that might have been seen by some concurrent reader thread during pre-crash execution. For nonblocking linearizable concurrent objects, this state will be one from which execution could reasonably continue. For programs using locks or other blocking mechanisms, additional machinery (e.g., redo or undo logs) will be required.

1.1 Consistency

On a machine with *relaxed consistency*, writes to a single location are totally ordered across threads, and each thread sees its own writes (to all locations) in program order [5]. Absent explicit synchronization, however, threads may see each others’ writes in arbitrary order. In this work we consider a *release consistent* model with `str` (*store*), `ld` (*load*), `str_rel` (*store release*), `ld_acq` (*load acquire*), and `CAS` (*compare-and-swap*) instructions. The last three of these are synchronization instructions. They appear, across all threads and locations, to execute in some total order such that, transitively, (1) each `ld_acq` or `CAS` appears to other threads to occur before any subsequent instructions in its own thread; (2) each `str_rel` or `CAS` appears to other threads to occur after any previous instructions in its own thread; and (3) each `str_rel` or `CAS` appears, to all threads, to occur before the next `ld_acq` or `CAS` in synchronization order that touches the same location.

This model corresponds closely to that of the ARM v8 instruction set. For purposes of our construction, it can also be considered to subsume the model of Intel’s x86 instruction set, where `str_rel` is emulated by an ordinary `str`, and where `ld_acq` is emulated with `<mfence;ld>` to force ordering with

respect to any previous `strs` that serve as `str_rel`. Absent a cross-thread `str_rel/CAS-to-ld_acq/CAS` dependence, we make no assumptions about when (or even if) the stores of one thread will become visible to the loads of another.

1.2 Persistency

Given transient registers and caches but persistent main memory, an instruction set must give the programmer control over the order and timing by which information becomes persistent. The instructions used to control this ordering and timing embody a memory *persistence model* [10]. Persistency may be independent of consistency, and controlled by different instructions. In particular, acquires and releases on current and forthcoming machines are expected to have no bearing on the order in which data are persisted; they may, however, interact with the persistency instructions.

We introduce an *explicit epoch persistency* model. Unlike previous, *implicit* versions of epoch persistency, which assume that the hardware will force dirty data back when necessary to preserve ordering [4, 8, 10], our model captures the behavior of both ARM and x86, and requires that programs employ explicit writes-back when ordering is required. As noted above, we also consider a release-consistent model of transient memory, rather the more restrictive TSO [4, 8] or sequential consistency [10] of prior work.

Under explicit epoch persistency, threads control the ordering and timing of persistency using three special instructions. A *persist write-back* (`pwb`) initiates write-back of a specified location to persistent memory, but does not block. A subsequent *persist fence* (`pfence`) enforces an ordering between previous and subsequent writes-back in the current thread. Finally, a *persist sync* (`psync`) blocks until all preceding `pfences` in the current thread have become persistent.

In the absence of fences, `pwb` instructions are allowed to reorder with respect to both ordinary and synchronization instructions. At the same time,

- each `pwb` is ordered with respect to (i) each preceding or subsequent `pfence` in its thread;
- for any given thread and location, each `pwb` is ordered with respect to (ii) each preceding `str/str_rel`, (iii) each preceding `ld/ld_acq`, and (iv) each preceding `pwb` of the same location in the same thread; and
- for any given thread, each `pfence` is ordered with respect to (v) each preceding `ld_acq` and (vi) each subsequent `str_rel` in that thread.

We also assume three other properties: First, writes-back persist atomically at some specified granularity [4]: their values cannot be torn across a fixed size. For generality, we assume here that a full-word write-back appears in its entirety or not at all in the wake of a crash. On real machines, atomicity is likely to be guaranteed at larger granularity—e.g., the width of a cache line. Second, persists to a given location respect coherence: the programmer need never worry that a newly persisted value will subsequently be overwritten by the write-back of some earlier value. Third, stored locations can “leak” back to persistence at any point after a store; in effect, extraneous `pwb`s can be inserted at will by the hardware or runtime system. Like explicit `pwb`s, these extraneous `pwb`s respect coherence and ordering.

Table 1 summarizes the mapping of our persistence instructions to the x86 and ARM ISAs. Neither instruction set currently distinguishes between `pfence` and `psync`, though

<i>Explicit Epoch Persistency</i>	<i>Intel x86 [6]</i>	<i>ARM v8 [1]</i>
<code>pwb addr</code>	<code>CLWB addr</code>	<code>DC CVAC addr</code>
<code>pfence</code>	<code>SFENCE</code> <code>PCOMMIT*</code> <code>SFENCE*</code>	<code>DSB†</code>
<code>psync</code>	<code>‘ ’</code>	<code>‘ ’</code>

Table 1: Equivalent instruction sequences for explicit epoch persistency.

*Required if memory controller buffer is transient.

†Requires persistent memory controller buffer; transient memory buffers are unsupported on ARM v8 at this time.

both may do so at some point in the future. For now, ordering requires that the current thread wait for values to reach persistence.

2. TRANSFORMATION

Our transformation takes a transient concurrent program annotated for release consistency and turns it into an equivalent program for explicit epoch persistency. This transformation preserves the happens-before ordering of the original concurrent program—that is, in the event of a crash, the values present in persistent memory are guaranteed to represent a consistent cut in the happens-before partial order of the original program.

Our transformation is as follows:

1. Immediately after every `str`, write back the written value by issuing a `pwb`.
2. Immediately before a `str_rel`, issue a `pfence`; immediately after a `str_rel`, write back the written value by issuing a `pwb`.
3. Immediately after an `ld_acq`, write back the loaded value by issuing a `pwb`, then issue a `pfence`.
4. Handle acquire-release CAS instructions as both `str_rel` and `ld_acq`: immediately before the CAS, issue a `pfence`; after the CAS, write back the loaded value by issuing a `pwb`, then issue another `pfence`.
5. Take no persistence action on `lds`.
6. Before taking any I/O action, issue a `psync` to ensure all changes have reached persistent storage.

2.1 Argument for Correctness

Let $O = x.ins_t(params)$ denote an instruction ins that is performed by thread t on memory location x , with parameters $params$. Let $\mathcal{M}$ be the set of *memory instructions* $\{\text{ld}, \text{str}, \text{ld_acq}, \text{str_rel}, \text{CAS}\}$, and $\mathcal{P}$ the set of *persistence instructions* $\{\text{pwb}, \text{pfence}, \text{psync}\}$. For any $O = x.ins_t(params)$, with $ins \in \mathcal{M}$, let $P = x.px_t^O$ denote the persistence operation with $px \in \mathcal{P}$ associated with the transformation of operation O . Note that $x.px_t^O$ is well-defined and unique, given our transformation. We treat $x.CAS_t(a, b)$ as an atomic $\langle O1; O2 \rangle = \langle x.str_rel_t(b); x.ld_acq_t(a) \rangle$ pair, transformed to $\text{pfence}_t^{O1}; \langle O1; O2 \rangle; x.pwb_t^{O1, O2}; \text{pfence}_t^{O2}$.

For every memory location x , we have a total order of values written to x , as memory is coherent. Without loss of generality, assume that the i -th value written to x is i , and its initial value is 0. Operations in our execution history are partially ordered by the *happens-before* relation, denoted $\prec$, respecting the constraints of the memory and persistence models. We wish to show that, in the wake of a crash,

the contents of persistent memory will always respect the happens-before order of `str` and `str_rel` instructions in the pre-crash execution.

Assume the contrary: $\exists A = x.st1_t(a), B = y.st2_u(b)$, with $st1, st2 \in \{\text{str}, \text{str_rel}\}$, such that $A \prec B$, but in the wake of a crash, B is seen to have persisted while A has not. Without loss of generality, we may assume that A and B are *consecutive* operations—that is, $\nexists C : A \prec C \prec B$ (otherwise, if C has persisted, proceed with C in place of B or, if C has not persisted, with C in place of A).

Let $W_B = y.pwb_u(b)$ be the write-back that persisted B . Our starting assumption implies that $\nexists W_A = x.pwb_v(a') \prec W_B$, with $a' \geq a$. Note the arbitrary issuing thread in W_A .

In the discussion below, we write $\prec^{(i, \dots, vi)}$ to justify a happens-before statement based on ordering properties (i) through (vi), enumerated in Section 1.2. The following cases are exhaustive:

1. If $t = u$ and $st2 = \text{str}$, then $y = x$ (otherwise $A \not\prec B$, since A and B are consecutive). In this case, we have $y.pwb_t^A(a) \prec^{(iv)} y.pwb_t^B(b)$.
2. If $t = u$ and $st2 = \text{str_rel}$, either $y = x$ or $y \neq x$. In either case, we have $x.pwb_t^A(a) \prec^{(i)} \text{pfence}_t^B \prec^{(vi)} B \prec^{(ii)} y.pwb_t^B(b)$.
3. If $t \neq u$, then $st1 = \text{str_rel}$, or we contradict the fact that A and B are consecutive. Hence, $[A = x.str_rel_t(a)] \prec [L = x.ld_acq_u(a)] \prec [B = y.st2_u(b)]$. However,
 - (a) $A \prec L$, so $A \prec L \prec^{(iii)} x.pwb_u^L(a') \prec^{(i)} \text{pfence}_u^L$, with $a' \geq a$ since writes-back are coherent; and
 - (b) $L \prec B$, so $\text{pfence}_u^L \prec^{(i)} y.pwb_u(b)$.

Therefore, $x.pwb_u^L(a') \prec \text{pfence}_u^L \prec y.pwb_u^B(b)$.

In all three cases, we have contradicted the starting assumption that $\nexists W_A \prec W_B$. That is, for any consecutively ordered stores A and B , if $W_B = y.pwb_u(b)$ persists into memory, some $W_A = x.pwb_v(a')$ also persists into memory, and becomes visible before W_B does. Between any consecutive stores, we have a `pwb` and a `pfence` (cases 2 and 3), or their persistence respects coherence (case 1).

3. CONCLUSIONS AND FUTURE WORK

Explicit epoch persistency, we believe, accurately captures the semantics that can be expected of forthcoming NVM systems. Our construction demonstrates that simple, mechanical transformations can preserve the happens-before order of properly synchronized programs, leading to meaningful post-crash memory contents. More specifically, in the wake of a crash, the contents of memory will reflect some consistent cut of the happens-before graph of pre-crash execution. For nonblocking concurrent objects, this cut represents a valid static state of the object, which can be trivially recovered [9]. For blocking objects, if the cut interrupts a failure-atomic or critical section, additional recovery mechanisms may be needed to roll the cut forward or backward [3, 7, 12] to reach a consistent state [2].

In ongoing work, we are continuing to investigate the notion of correctness for persistent programs and to develop programming methodologies to (1) reduce the cost of persistence for nonblocking concurrent objects, (2) preserve the consistency of blocking objects, and (3) support the composition of atomic operations into larger ACID transactions—atomic, consistency-preserving, isolated, and persistent (durable).

4. REFERENCES

- [1] ARM Limited. ARM Cortex-A series programmer's guide for ARMv8-A. Technical Report DEN0024A:ID050815, ARM Limited, Mar. 2015.
- [2] H.-J. Boehm and D. Chakrabarti. Persistence programming models for non-volatile memory. Technical Report HP-2015-59, HP Labs, Aug. 2015.
- [3] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *Proc. of the 2014 ACM Intl. Conf. on Object Oriented Programming Systems Languages and Applications*, Portland, OR, USA, 2014.
- [4] 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 *Proc. of the ACM 22nd Symp. on Operating Systems Principles*, Big Sky, MT, USA, 2009.
- [5] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. of the 17th Intl. Symp. on Computer Architecture*, Seattle, WA, USA, 1990.
- [6] Intel Corporation. Intel architecture instruction set extensions programming reference. Technical Report 319433-022, Intel Corporation, Oct. 2014.
- [7] J. Izraelevitz, T. Kelly, and A. Kolli. Failure-atomic persistent memory updates via JUSTDO logging. In *Proc. of the 21st Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Atlanta, GA, USA, 2016.
- [8] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas. Efficient persist barriers for multicores. In *Proc. of the 48th Intl. Symp. on Microarchitecture*, 2015.
- [9] F. Nawab, D. R. Chakrabarti, T. Kelly, and C. B. Morrey III. Procrastination beats prevention: Timely sufficient persistence for efficient crash resilience. In *Proc. of the 18th Intl. Conf. on Extending Database Technology*, Brussels, Belgium, Mar. 2015.
- [10] S. Pelley, P. M. Chen, and T. F. Wenisch. Memory persistency. In *Proc. of the 41st Intl. Symposium on Computer Architecture*, Piscataway, NJ, USA, 2014.
- [11] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proc. of the 9th USENIX Conf. on File and Storage Technologies*, Berkeley, CA, USA, 2011.
- [12] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *Proc. of the 16th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, CA, USA, 2011.
- [13] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He. NV-Tree: Reducing consistency cost for NVM-based single level systems. In *Proc. of the 13th USENIX Conf. on File and Storage Technologies*, Santa Clara, CA, USA, Feb. 2015.