Easy Lock-Free Programming in Non-Volatile Memory *

Tianzheng Wang †
Simon Fraser University
tzwang@sfu.ca

Justin Levandoski †
Amazon Web Services
jjl@amazon.com

Per-Åke Larson
University of Waterloo
plarson@uwaterloo.ca

Overview

Many systems use lock-free data structures (e.g., queues, B+-trees) to achieve high performance. Byte-addressable, non-volatile memory (NVRAM) such as Intel 3D XPoint further adds persistence to these data structures on the memory bus, potentially enabling desired features like instant recovery and lower cost while maintaining high performance. For example, a database index stored in NVRAM can save much index rebuild time and thus reduce service downtime.

Lock-Free Programming in NVRAM is Harder. While highly concurrent, lock-free data structures are notoriously hard to build as they usually need to atomically modify multiple 8-byte words (e.g., B+-tree splits), but the hardware only provides atomic instructions such as `compare-and-swap` (`CAS`) that work only on *single* 8-byte words. Lock-free programming in NVRAM is even harder: the same instructions can be used, but since the CPU cache is volatile, there has to be a persistence protocol in place so that the data structure recovers correctly after a crash. Such persistence protocols tend to be data structure specific, complex and error-prone to implement.

Solution: Persistent Multi-Word Compare-and-Swap (`PMwCAS`). `PMwCAS` is an efficient software primitive that allows applications to atomically change multiple arbitrary 8-byte NVRAM words in a lock-free manner. It exhibits the following attractive features:

- **Persistence Guarantees.** `PMwCAS` guards against tricky bugs in NVRAM programming by ensuring that readers only see persisted values without any expensive logging operations.
- **Transparent and Instant Recovery.** A big advantage of `PMwCAS` is that users can completely avoid application-specific recovery code: `PMwCAS` transparently recovers the data structure to a consistent state by completing or rolling back in-flight operations.
- **Simpler Code.** The implementation of a lock-free data structure using `PMwCAS` is almost as mechanical as a lock-based one.
- **Simpler Memory Management.** Extra care must be taken since any leak will be permanent. `PMwCAS` allows data structures to easily piggyback on its memory recycling protocol to ensure safe memory reclamation after a `PMwCAS` operation or a crash.
- **Robust Performance.** `PMwCAS` maintains robust performance even under high contention. Compared to `PMwCAS`, hardware transactional memory is very vulnerable to high contention.

We have used `PMwCAS` to adapt the Bw-tree [3] (a lock-free B+-tree in SQL Server) and a doubly-linked skip list for NVRAM. `PMwCAS` is also used by the BzTree [1], a new NVRAM B+-tree. `PMwCAS` exhibits 4–6% overhead under realistic workloads. `PMwCAS` is open source at https://github.com/Microsoft/pmwcas.

1. Persistent Multi-Word Compare-and-Swap

We base `PMwCAS` on the volatile `MwCAS` by Harris et al. [2] and enhance it to work correctly in NVRAM. Like `MwCAS`, `PMwCAS` is built around the concept of descriptors and employs a two-phase execution approach. Compared to `MwCAS`, `PMwCAS` further supports persistence, memory management and recovery in NVRAM.

1.1 Application Interfaces

Each `PMwCAS` operation uses a descriptor that specifies the memory words to change and tracks the operation's status. Applications use the following APIs to conduct `PMwCAS` operations.

- `AllocateDescriptor(callback)`: Allocate a `PMwCAS` descriptor. The user can provide a callback function for recycling memory pointed to by the words in the `PMwCAS` operation.
- `Descriptor::AddWord(address, expected, desired)`: Specify a word to be modified. The caller provides the address of the word, the expected value and the desired value.
- `Descriptor::ReserveEntry(addr, expected, policy)`: Same as `AddWord`, except the new value is left unspecified; returns a pointer to the `new_value` field so it can be filled in later. Memory referenced by `old_value`/`new_value` will be recycled according to the specified policy (described later).
- `Descriptor::RemoveWord(address)`: Remove the word previously specified as part of the `PMwCAS`.
- `PMwCAS(descriptor)`: Execute the `PMwCAS` and return true if succeeded, false otherwise.
- `Discard(descriptor)`: Cancel (abort) the `PMwCAS` (only valid before calling `PMwCAS`).

To perform a `PMwCAS`, the application first allocates a descriptor and invokes `AddWord` or `ReserveEntry` once for each word to be modified. `RemoveWord` can be used to remove a previously added word. `AddWord` and `ReserveEntry` ensure that target addresses are unique and return an error if they are not. After the descriptor is persisted, calling `PMwCAS` executes the operation, while `Discard` aborts it. A failed `PMwCAS` will leave all target words unchanged.

1.2 PMwCAS Design

Descriptor. Each `PMwCAS` operation uses a descriptor that summarizes the details of the operation. As Figure 1(right) shows, a descriptor includes a `status` variable that tracks the operation's progress, a reference to an optional callback function, and an array of word descriptors. The callback is invoked when the descriptor is no longer needed and typically frees memory objects that are no longer needed after the operation has completed. A word descriptor contains the target word's address, expected and new values, and a pointer to the containing `PMwCAS` descriptor (`0x100` in our example). Each word is also associated with a memory policy that indicates whether the new and old values are pointers to memory objects and, if so, which objects are to be freed on completion (or failure) of the operation. In the example, two words use the `FreeOne` policy that will free the memory pointed to by the old (new) value field if the `PMwCAS` succeeds (fails). Our ICDE paper [4] has a full description of different memory policies.

Word Type Identification. We leverage the fact that modern CPUs use 48 (out of 64) bits for addressing and dedicate three vacant bits to indicate whether a word contains a pointer to a word descriptor, a

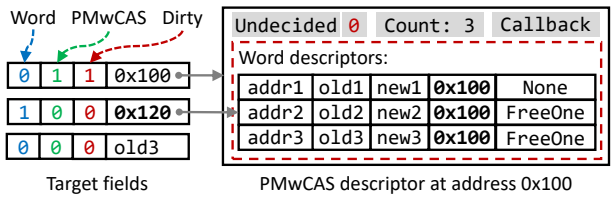

Figure 1: Flag bits employed in target words (left) and an example `PMwCAS` descriptor (right) for changing three NVRAM words.

*Published in ICDE 2018 [4]: www.cs.sfu.ca/~tzwang/pmwcas.pdf.
†Work performed while at Microsoft Research.

pointer to a PMwCAS descriptor, and whether the value might not be persisted (the dirty bit), as shown in Figure 1(left).

Two-Phase Execution. With a descriptor that is filled out and persisted, the PMwCAS operation executes in two phases:

- Phase 1: Install a descriptor pointer in all target words using CAS.
- Phase 2: If Phase 1 succeeded, install the new values in all target words. If Phase 1 failed, then reset any target word that points to the descriptor back to its old value.

If any phase fails, the PMwCAS operation will roll back and the target words will be recovered to carry their original values. In effect, Phase 1 attempts to "lock" each target word. We keep all word entries in the descriptor sorted on the address field, so deadlocks cannot occur as all threads will install descriptors in the same order.

A thread may read a word that stores a descriptor pointer instead of a "regular" value. If so, the thread helps complete the referenced PMwCAS before continuing. Following a *flush-on-read* principle, if a thread sees a word with a set dirty bit, it must first flush the word using CLWB or CLFLUSH before accessing the word; the thread then resets the dirty bit using a CAS in case another thread is trying to modify the word and set the dirty bit. Flush-on-read ensures that a write is persisted in NVRAM before any dependent reads; this way, we guarantee correct recovery of the data structure without logging.

Transparent Recovery. With PMwCAS, the user can completely avoid application-specific recovery code as PMwCAS can transparently recover the data structure to a consistent state. We maintain a pool of descriptors in an application-specified NVRAM location. Upon restart, we scan the descriptors in the pool and process each in-flight operation by either completing or rolling back it depending on its status. Correct recovery requires that the descriptor be persisted before entering Phase 1. Descriptors are reused and we only need to maintain a small descriptor pool (a small multiple of the number of worker threads). Thus, scanning the pool during recovery is not time consuming.

Memory Management. PMwCAS recycles descriptors using epoch-based reclamation. The application can piggyback on this protocol using callbacks that will be invoked once it is determined (by the recycling policy specified through ReserveEntry) that memory behind each pointer is safe to be freed. For example, one can specify recycling memory pointed to by old values if the PMwCAS succeeds. To avoid permanent memory leaks, ReserveEntry returns a pointer to the newly added entry's new value field, which can be given to an allocator as the target location for storing the address of the allocated NVRAM block (similar to posix_memalign).

2. Case Study and Evaluation

We implemented two non-trivial lock-free data structures using PMwCAS: a doubly-linked skip list and the Bw-tree [3]. Doubly-linked skip lists are useful in database systems to enable reverse scan but hard to implement using single-word atomic instructions. Due to space limitation, we focus on the Bw-tree here.

Structure modification operations (SMOs) such as page splits and merges cause most of the complexity in the Bw-tree, since they need to change multiple pages which cannot be done with a CAS. The Bw-tree breaks an SMO into a sequence of atomic steps; each step uses a CAS. While highly concurrent, the Bw-tree contains some subtle race conditions as a result of the SMO protocol. For example, threads can observe in-progress SMOs. A large amount of code and thought is dedicated to detecting and handling such subtle cases.

We use PMwCAS to simplify Bw-tree's SMO protocol by "collapsing" the multi-step SMO into a single PMwCAS. For example, a page split first allocates a new sibling page, along with memory for insertions into the parent page. It then uses PMwCAS to atomically modify all pages involved. If the split triggers further splits at upper levels, we repeat this process for the parent. PMwCAS allows us to cut all the

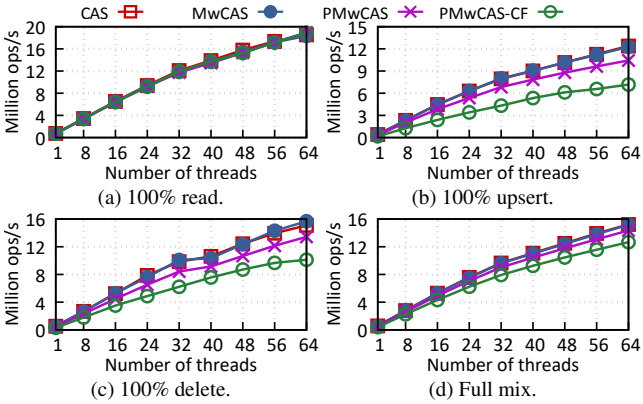

Figure 2: Bw-tree performance with 8-byte keys and 8-byte values.

race handling code and the implementation is almost as mechanical as a lock-based one, with 24% lower cyclomatic complexity.

We conduct experiments on a 4-socket, 64-thread machine with 4 Intel Xeon E5-4620 processors (2.2GHz) and 512GB main memory. Since real NVRAM is yet to come, we target NVDIMM which behaves exactly the same as DRAM at runtime and run all experiments in DRAM. We test both individual (read, upsert, delete) and mixed operations. The mixed workload consists of 20% write, 64% point get and 16% scan. We use 8-byte keys, 8-byte values, and initialize both indexes with 10 million records.

Figure 2 shows Bw-tree's performance under different synchronization primitives. Compared to PMwCAS, PMwCAS-CF issues CLFLUSH that will evict cache line contents; MwCAS simply turns off the persistence guarantees to show an upper bound on performance. All the variants show similar performance for the read-only workload. For upsert and delete, PMwCAS adds ∼15% overheads for persistence guarantees. However, this is a fixed amount of overhead and does not affect scalability. Finally, we observed that CLFLUSH could degrade throughput by more than 30%. This is the worst case scenario and underlines the need for CLWB. Since the mixed workload has more reads, it exhibits smaller differences among the evaluated variants. Compared to the best-performing CAS variant, PMwCAS incurs on average ∼4–6% of overhead. We believe the significant ease of programming efforts justifies such low overhead.

Finally, an implementation of the aforementioned BzTree recovered in 145μs after crashing with 48 worker threads executing the YCSB workload [1]. Such performance illustrates that PMwCAS helps support near instantaneous recovery in non-trivial data structures.

3. Conclusion

Lock-free data structures are hard to build in NVRAM. Traditional approaches handle complex races using single-word instructions and must implement custom recovery logic. Our contribution is PMwCAS, a software primitive that can atomically change multiple 8-byte words in a lock-free manner with persistence guarantees and transparent recovery. The result is code that is easy to maintain and reason about (almost as mechanical as a lock-based one), but matches lock-free performance. Moreover, the same implementation can be used for both volatile and persistent data structures.

References

[1] J. Arulraj, J. Levandoski, U. F. Minhas, and P.-A. Larson. BzTree: A high-performance latch-free range index for non-volatile memory. *PVLDB*, 11(5), Jan. 2018.

[2] T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. *DISC*, pages 265–279, 2002.

[3] J. Levandoski, D. Lomet, and S. Sengupta. The Bw-Tree: A B-tree for new hardware platforms. *ICDE*, pages 302–313, 2013.

[4] T. Wang, J. Levandoski, and P.-A. Larson. Easy lock-free indexing in non-volatile memory. *ICDE*, 2018.