Easy Lock-Free Indexing in Non-Volatile Memory

Tianzheng Wang[1]* Justin Levandoski[2] Per-Åke Larson[3]

[1]University of Toronto, [2]Microsoft Research, [3]University of Waterloo
[1]tzwang@cs.toronto.edu, [2]justin.levandoski@microsoft.com, [3]plarson@uwaterloo.ca

***Abstract*—Large non-volatile memories (NVRAM) will change the durability and recovery mechanisms of main-memory database systems. Today, these systems make operations durable through logging and checkpointing to secondary storage, and recover by rebuilding the in-memory database (records and indexes) from on-disk state. A main-memory database stored in NVRAM, however, can potentially recover instantly after a power failure. Modern main-memory databases typically use lock-free index structures to enable a high degree of concurrency. Thus NVRAM-resident databases need indexes that are both lock-free, persistent, and able to recover (almost) instantly after a crash. In this paper, we show how to easily build such index structures. A key enabling component of our scheme is a multi-word compare-and-swap operation, `PMwCAS`, that is lock-free, persistent, and efficient. `PMwCAS` significantly reduces the complexity of building lock-free indexes, which we illustrate by implementing both doubly-linked skip lists and the Bw-tree lock-free B+-tree for NVRAM. Experimental results show that `PMwCAS`'s runtime overhead is very low (∼4–6% under realistic workloads). This overhead is sufficiently low that the same implementation can be used for both DRAM and NVRAM resident indexes.**

I. Introduction

Fast, byte-addressable non-volatile memory (NVRAM) devices are currently coming online in the form of NVDIMM [1], Intel 3D XPoint [2], and STT-MRAM [3]. NVRAM blurs the distinction between memory and storage: besides being non-volatile and spacious, NVRAM provides close-to-DRAM performance and can be accessed by normal `load` and `store` instructions. Currently, several companies ship NVRAM-N DIMMs (flash and super-capacitor backed DRAM) with up to 32GB capacity per chip, making it an attractive medium for main-memory database workloads.

NVRAM potentially allows almost *instant* recovery of a database after a crash. A primary challenge is implementing persistent, high-performance indexes. Several systems implement lock-free indexes to fully exploit the hardware parallelism in modern servers: MemSQL uses lock-free skip lists [4], while Microsoft Hekaton uses the Bw-tree [5], a lock-free B+-tree.

Non-trivial lock-free data-structures are already tricky to design and implement in the volatile case. These implementations use atomic instructions such as compare-and-swap (`CAS`) to coordinate interaction among threads. However, these instructions operate on single words, and non-trivial data structures usually require atomic updates of multiple words (e.g., B+-tree splits). Implementing lock-free indexes on NVRAM in this manner is even more difficult: the same atomic instructions can still be used, but since the processor cache is volatile there must be a persistence protocol in place to ensure the data structure recovers correctly after a crash. The key is to make sure that a write is persisted on NVRAM before any dependent reads, otherwise the index might recover to an inconsistent state.

In this paper, we show how to build efficient lock-free indexes for NVRAM relatively easily. The key to our technique is a *persistent multi-word compare-and-swap* operation (`PMwCAS`) that provides atomic compare-and-swap semantics across arbitrary words in NVRAM. The operation itself is *lock-free and guarantees durability of its modifications, even across crashes*. `PMwCAS` greatly simplifies the implementation of lock-free data structures. Using `PMwCAS`, the developer simply specifies the memory words to modify along with the expected and desired values for each (similar to a single-word `CAS`). `PMwCAS` will either atomically install all new values or fail the operation without exposing a partially completed operation) to other threads. `PMwCAS` reserves three bits (for status information) in each target word; if three free bits are not available, the technique cannot be used. This has not been a problem in index implementations; target words have always been pointers, status words, or (small) counters.

The `PMwCAS` operation is the key contribution of this paper and the first implementation of a multi-word `CAS` operation for non-volatile memory. It is based on the volatile `MwCAS` operation by Harris et al [6] which we extend with persistence guarantees and transparent recovery. There are two other versions of volatile `MwCAS` operations [7], [8] but they are either slower and/or more complex than the version by Harris et al.

The `PMwCAS` operator has several features that make it attractive for lock-free programming in an NVRAM environment.

Persistence guarantees. `PMwCAS` guards against tricky bugs inherent in an NVRAM environment. For example, on persistent memory, updating a value v using a volatile `CAS` can lead to corruption. Since `CAS` does not guarantee persistence of v (CPU caches are not persistent), another thread might read v and take action (e.g., perform further writes) without guarantee that v will become durable before a crash. Our `PMwCAS` implementation ensures that readers only see persistent values.

Transparent and instant recovery. A big advantage of `PMwCAS` is that users can avoid application-specific recovery code completely. On restart after a crash, the `PMwCAS` library transparently recovers the data structure to a consistent state by completing or rolling back operations that were in flight at the time of the crash. Recovery time is proportional to the number of in-flight operations and virtually instant.

* Work performed while at Microsoft Research.

Simpler code. PMwCAS simplifies implementation of lock-free data structures. Non-trivial lock-free data structures usually requires atomically changing multiple words. This is complex and error-prone to achieve using single-word CAS. With PMwCAS, the implementation is almost as mechanical as a locked based one. In addition, the same implementation can be used in both volatile DRAM as well as NVRAM.

Simpler memory management. Lock-free programming requires careful memory reclamation protocols, since memory cannot be freed under mutual exclusion. Memory management is even more difficult in an NVRAM environment. For instance, unless care is taken, a new node that was allocated but not yet added to the index will be leaked when the system crashes. We have designed PMwCAS so that index implementations can easily piggyback on its memory recycling protocol to ensure that memory is safely reclaimed after the success (or failure) of an operation and even after a crash.

Robust performance (compared to HTM). Recent hardware transactional memory (HTM) [9] provides an alternative to PMwCAS as it could be used to atomically modify multiple NVRAM words. However, this approach is vulnerable to spurious aborts (e.g., caused by CPU cache size) and still requires application-specific recovery logic. As Section IV-E shows, MwCAS is only 3% to 12% slower then HTM under low contention and significantly faster under high contention.

To illustrate how PMwCAS simplifies index implementation in NVRAM, we describe the implementation of two index types.

- *A doubly-linked lock-free skip list*. Skip lists are used in a number of research and production main-memory databases, e.g., MemSQL [4]. We detail how to implement a persisted skip list with forward and backward links.
- *The Bw-tree [5]*. The Bw-tree is the lock-free B+-tree used by SQL Server Hekaton [10]. We detail how to use PMwCAS to simplify Bw-tree structure modifications that span multiple nodes.

We also provide an extensive experimental evaluation of our techniques. Using microbenchmarks, we show that PMwCAS performs robustly and is efficient even under high-contention. Under realistic workloads, the overhead for our persistence version of skip lists is only ∼1–3% compared with a volatile CAS based implementations; for Bw-tree, the overhead is 2–8%.

In the rest of this paper, Section II covers our assumptions and the PMwCAS interface. Section III describes a single-word persisted CAS as a basis for understanding PMwCAS (Section IV). Section V covers NVRAM management issues. Section VI describes our index implementations using PMwCAS, while Section VII provides experimental evaluation. Related work is covered in Section VIII, and Section IX concludes the paper.

II. Background

A. System Model

We assume a single-level store model where NVRAM is attached directly to the memory bus and the operating system provides access to (byte addressable) NVRAM through the memory mapping interface. This model was also adopted by several recent NVRAM based systems [11], [12], [13], [14], [15], [16]. A single node can have multiple processors. We assume that indexes and base data reside in NVRAM. The system may also contain DRAM which is used as working storage.

Access to NVRAM is cached by multiple levels of *volatile* private and shared CPU caches, and is subject to re-ordering by the processor for performance reasons. Special care must be taken to guarantee durability and ordering. This is typically done through a combination of cache write-backs and memory fences. In addition to memory fences and atomic 8-byte writes, we assume the ability to selectively flush or write-back a cache line, e.g., via the cache line write-back (CLWB) or cache line flush (CLFLUSH) instructions on Intel processors [9]. Both of these instructions flush the target cache line to memory but CLFLUSH also evicts the cache line. This increases the number of memory accesses which slow down performance.

B. The PMwCAS Operation

Our techniques rely on an efficient PMwCAS operator to atomically change multiple 8-byte words with persistence guarantees. The API for PMwCAS is:

- AllocateDescriptor(callback): Allocate a descriptor that will be used throughout the PMwCAS operation. The user can provide a custom 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): Similar to 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 (details in Section V).
- Descriptor::RemoveWord(address): Remove the word previously specified as part of the PMwCAS.
- PMwCAS(descriptor): Execute the PMwCAS and return true if succeeded.
- Discard(descriptor): Cancel the PMwCAS (only valid before calling PMwCAS).

The API is identical for both volatile and persistent MwCAS. Internally, PMwCAS provides all the needed persistence guarantees, without additional actions by the application.

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

The word entries in the descriptor are sorted on the address field to prevent deadlock. The first phase of PMwCAS in effect attempts to “lock” each target word. From concurrency control theory, deadlocks cannot occur if all “clients” acquire locks (or other resources) in the same order.

Memory management. To ensure memory safety in a lock-free environment, descriptors are recycled by the PMwCAS and Discard functions using epoch-based reclamation (see Section V). The user need not worry about descriptor memory. PMwCAS is most often used to update pointers to dynamically allocated memory. The callback parameter allows the user to piggyback on PMwCAS's epoch-based reclamation protocol. The callbacks are invoked once it is determined that memory behind each pointer is safe to be recycled. The user can provide a recycling policy (using ReserveEntry) to specify the circumstance under which a callback is invoked (e.g., recycling memory pointed to by old values after the PMwCAS succeeds).

In addition to memory recycling, the PMwCAS must correctly interact with the allocator and avoid leaking memory even if the system crashes in the middle of a PMwCAS operation. To handle this, ReserveEntry will return a pointer to the newly added entry's new value field, which can be given to a persistent memory allocator as the target location for storing the address of the allocated memory (similar to posix_memalign [17]). Section V discusses the details behind memory management.

III. A Persistent Single-Word CAS

To set the stage for describing our PMwCAS implementation, we first summarize an approach to building a single-word persistent CAS. To maintain data consistency across failures, a single-word CAS operation on NVRAM can proceed only if its target word's existing value is persistent in NVRAM. In general, inconsistencies may 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 be persisted. Such inconsistencies can be avoided by a *flush-on-read* principle: any load instruction must be preceded by a cache line flush (e.g., via CLFLUSH or CLWB [9]) to ensure that the word is persistent in NVRAM. Flush-on-read is straightforward to implement but sacrifices much performance. Fortunately, there is a way to drastically reduce the number of flushes.

CAS operates on word-aligned boundaries [18], so certain low-order bits in the operands are always zero. For example, the lower four bits are always zero if the operands are at least 4-byte aligned. Another source of such "vacant" bits is the result of the "canonical address" design employed by modern 64-bit x86 processors [9], where the microarchitecture only implements 48 address bits, leaving the higher 16 bits unused. These vacant bits can be used to help improve the performance of persistent CAS: a bit can be dedicated to indicate whether the value is guaranteed to be persistent. Throughout this paper, we call this the "dirty" bit and in an actual implementation it can be placed in either the high 16 bits or the low bits due to alignment requirements. If the dirty bit is clear, the word is guaranteed to be persistent; otherwise it *might* not be persistent.[1] Thus the protocol is that (1) a store always sets the dirty bit and (2) any thread accessing a word (either read/write) with the dirty bit set flushes it and then clears the dirty bit to avoid unnecessary, repetitive flushes.

[1]Some system events (e.g., cache eviction) could implicitly persist the word.

Algorithm 1 A persistent single-word CAS.

```
1  def pcas_read(address):
     word = *address
3    if word & DirtyFlag is not 0:
       persist(address, word)
5    return word & ~DirtyFlag

7  def persistent_cas(address, old_value, new_value):
     pcas_read(address)
9    # Conduct the CAS with dirty bit set on new value
     return CAS(address, old_value, new_value | DirtyFlag)
11
   def persist(address, value):
13   CLWB(address)
     SFENCE
15   CAS(address, value, value & ~DirtyFlag)
```

Algorithm 1 shows how persistent CAS can be built following this principle. The DirtyFlag is a word-long constant with only the dirty bit set. Before executing the final CAS at line 10, the caller must first make sure that the target word is durable by checking if the dirty bit is set and possibly flush the word using the CLWB [9] instruction (lines 3–4 and 13–14). Note that after issuing CLWB, a store fence (SFENCE) is needed to ensure correct write ordering and visibility of writes [9]. At line 15, a CAS must be used to clear the dirty bit as (1) there may be concurrent threads trying to also set the bit or (2) there may be concurrent threads attempting to change the word to another value. This step does not require a flush, however, since any read operation of words that might participate in the persistent CAS must be done through pcas_read in Algorithm 1.

Employing a dirty bit on the target word solves both problems of data consistency and performance. A thread can only read a target word after making sure the word is durable in NVRAM. Clearing the dirty bit after flushing avoids repetitive flushing, maintaining most benefits of write-back caching.

IV. Persistent Multi-Word CAS

We now discuss how to implement PMwCAS using the above principles. The key is persisting and correctly linearizing access to the information needed by the multi-word CAS.

A. PMwCAS Overview

Each PMwCAS operation uses a descriptor that describes the operation to be performed and tracks it status. Figure 1 (ignore the target words for now) shows the internals of a descriptor. It includes a status variable that tracks the operation's progress, an optional pointer to a callback function, and an array of *word descriptors*. In the status field we employ its most significant bit as a dirty bit to indicate whether it is modified/persisted in NVRAM, following the same flush-on-read principle described in Section III. Table I lists the possible status values and their meanings. The callback function is called when the descriptor is no longer needed and typically frees memory objects that can be freed after the operation has completed. The callback is not a raw function pointer (since the function may not map to the same address after a crash). Instead, we allocate an array for storing pointers to finalize callback functions and the array

TABLE I: Possible values and meaning of the status field in PMwCAS descriptor. Value of the dirty bit is omitted for brevity.

Value	Meaning
Undecided	The PMwCAS operation in progress.
Succeeded	The PMwCAS operation succeeded.
Failed	The PMwCAS operation failed.

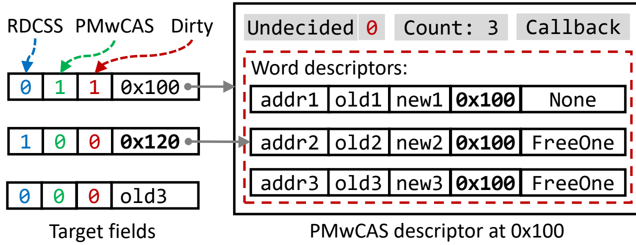


Fig. 1: Flag bits employed in target words (left) and an example PMwCAS descriptor (right). Callbacks and memory policy (word descriptor's right-most field) are discussed in Section V.

is filled in at startup. A descriptor then refers to a callback function by its position in the array instead of by its address.

A word descriptor contains (1) the target word's address, (2) the expected value to compare against, (3) the new value, (4) a back pointer to the containing descriptor, and (5) a memory deallocation 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.

As shown on the left side of Figure 1, we use three vacant bits to indicate whether a word contains a pointer to a word descriptor, a pointer to a PMwCAS descriptor, and whether the value might not be persisted. We refer to them respectively as RDCSSFlag, PMwCASFlag, and DirtyFlag, which are constants with only the corresponding bit set.

The example descriptor in Figure 1 is currently in the initial Undecided status and looking to change three words. All three word descriptors contain a back pointer to the descriptor at address 0x100 and policy specification.

Users of PMwCAS first allocate a descriptor using the API in Section II-B, and add per-word modifications using AddWord or ReserveEntry. After adding all modifications, the user persists the descriptor and performs the PMwCAS operation by issuing the PMwCAS command (or Discard if the user wishes to cancel). If the PMwCAS succeeded, the user is guaranteed that all the target words were updated atomically and the new values will persist across failures. On failure, the user is guaranteed that none of the updates are visible to other threads.

The PMwCAS operation executes in two phases:

- Phase 1: Install a descriptor pointer in all target words.
- 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.

For a successful execution, with its initial old value, each target word will be changed to carry in sequence:

1) A pointer to the corresponding word descriptor;
2) A pointer to the PMwCAS descriptor (if step 1 succeeds);

Algorithm 2 PMwCAS algorithm.

```
1  bool pmwcas(Descriptor md):
     st = Succeeded
3    for w in md.words (in sorted order on md.words.address):
     retry:
5      rval = install_mwcas_descriptor(w)
       if rval == w.old_value:
7        # Descriptor successfully installed
         continue
9      elif rval & PMwCASFlag is not 0:
         if rval & DirtyFlag is not 0:
11         persist(w.address, rval)
         # Clashed another on-going MwCAS, help it finish
13       pmwcas(rval.Address)
         goto retry
15     else
         st = Failed
17       break

19   # Persist all target words if Phase 1 succeeded
     if st == Succeeded:
21     for w in md.words:
         persist(w.address, md| PMwCASFlag | DirtyFlag)
23
     # Finalize the MwCAS's status
25   CAS(md.status, Undecided, st | StatusDirtyFlag)
     if md.status & DirtyFlag:
27     CLWB(&md.status)
       SFENCE
29     md.status &= ~DirtyFlag

31   # Install the final values
     for w in md.words:
33     v = md.status == Succeeded ? w.new_value : w.old_value
       expected = md | PMwCASFlag | DirtyFlag
35     rval = CAS(w.address, expected, v | DirtyFlag)
       if rval == md | PMwCASFlag:
37       CAS(w.address, expected & ~DirtyFlag, v)
       persist(w.address, v)
39   return md.status == Succeeded
```

3) The new value (if step 2 succeeds).

If any of the above steps fails, the PMwCAS operation will roll back and the target words will be recovered to carry their original old values.

Another thread may read a word that contains a descriptor pointer instead of a "regular" value. If so, the thread helps complete the referenced PMwCAS before continuing. The following sections describe how PMwCAS works in detail. Algorithm 2 shows the main PMwCAS function. Algorithm 3 gives the entry point for readers (pmwcas_read) and two helper functions: install_mwcas_descriptor is the entry point to install a pointer to a descriptor at a particular address, while complete_install allows the reader to help along to complete an in-progress PMwCAS.

B. Phase 1: Installing Descriptors

The PMwCAS first installs a pointer to the descriptor in each target word. Along the way, it or other reads may encounter another in-progress PMwCAS, for which it must help to complete (Section IV-B1). It then persists the descriptor pointer writes before determining the final operation status (Section IV-B2).

For each target word in the descriptor md, PMwCAS first

attempts to install a pointer to md in each word (Algorithm 2 lines 3–8). This is done by the install_mwcas_descriptor function (line 5), which is essentially a *double-compare single-swap* (RDCSS) operation [6]. An RDCSS applies change to a target word only if the values of two words (including the one being changed) match their specified expected values. That is, RDCSS requires an additional "expected" value to compare against (but not modify) than CAS does.

RDCSS is a software operation, and for our purpose of installing a PMwCAS descriptor pointer in a target word, we expect (1) the target word to contain the old value, and (2) the PMwCAS descriptor's status is Undecided. The "desired" new value for the RDCSS is a pointer to the PMwCAS descriptor. RDCSS is necessary to guard against subtle race conditions and maintain a linearizable sequence of operations on the same word. Specifically, we must guard against the installation of a descriptor for a completed PMwCAS (p_1) that might inadvertently overwrite the result of another PMwCAS (p_2), where p_2 should occur after p_1. This can happen if a thread t executing p_1 is about to install a descriptor in a target word a over an existing value v, but goes to sleep. While t sleeps, another thread may complete p_1 (given the cooperative nature of PMwCAS) and subsequently p_2 executes to set a back to v. If t were to wake up and try to overwrite v (the value it expects) in address a, it would overwrite the result of p_2, violating the linearizable schedule for updates to a. Using RDCSS to install a descriptor ensures not only that the target word contains the expected value but also that the status is Undecided, i.e., that the operation is still in progress.

Lines 1–13 of Algorithm 3 shows how RDCSS works in detail. The install_mwcas_descriptor function (i.e., the RDCSS operation) receives the address of a word descriptor as the sole parameter and returns the value found in the word. It first uses a CAS to install a pointer to the *word* descriptor in the target word (lines 2–4). If the target word already points to a word descriptor, the caller helps complete that RDCSS and then retries its own RDCSS (lines 5–8). If the CAS succeeds, it then sets the target word to point to the descriptor if status is Undecided (lines 10–12 and 15–18). If the PMwCAS has finished (status is Succeeded or Failed), the installation fails and the target word is reset to the old value.

Figure 1 shows an example where the RDCSS has successfully installed a pointer to the descriptor in the first target word. The PMwCAS and dirty bits are set to indicate that the word contains a descriptor pointer and its content might not be durable on NVRAM. The second target word, however, still points to its word descriptor at address 0x120[2]. So for this word the caller could be executing lines 5–12 of Algorithm 3. The last target word is yet to be changed and contains the old value.

At line 5 of Algorithm 2, install_mwcas_descriptor returns one of the following values when trying to install a pointer to descriptor md. (1) A regular value that equals the expected value, signalling success. (2) A regular value

[2]This is 40 bytes off of the start address of the full descriptor, given a word descriptor and status each takes 32 and 8 bytes, respectively.

Algorithm 3 Help-along and read routines for PMwCAS.

```
1  def install_mwcas_descriptor(wd):
     ptr = wd | RDCSSFlag
3  retry:
     val = CAS(wd.address, wd.old_value, ptr)
5    if val & RDCSSFlag is not 0:
       # Hit another RDCSS operation, help it finish
7      complete_install(val & AddressMask)
       goto retry
9
     if val == desc.old_value:
11     # Successfully installed the RDCSS descriptor
       complete_install(wd)
13   return val

15 def complete_install(wd):
     ptr = wd.mwcas_descriptor | PMwCASFlag | DirtyFlag
17   u = wd.mwcas_descriptor.status == Undecided
     CAS(wd.address, wd | RDCSSFlag, u ? ptr : wd.old_value)
19
   def pmwcas_read(address):
21 retry:
     v = *address
23   if v & RDCSSFlag:
       complete_install(v & AddressMask)
25     goto retry

27   if v & DirtyFlag:
       persist(address, v)
29     v &= ~DirtyFlag

31   if v & PMwCASFlag:
       pmwcas(v & AddressMask)
33     goto retry
     return v
```

that does *not* equal the expected value, signaling a lost race with another PMwCAS that installed a new value before our RDCSS could install the descriptor pointer. In this case the PMwCAS fails (lines 16–17). (3) The pointer to md, i.e., another thread successfully installed md. (4) A pointer to the descriptor of another PMwCAS, in which case we help to complete that operation (lines 9–14) before retrying the installation of md. In all cases, if the return value's dirty bit is set, we persist the word using the persist function defined in Algorithm 1.

1) Reading Affected Words: Phase 1 exposes descriptor pointers to any thread reading one of the target words. Similar to MwCAS [6], a thread does not directly read words that may contain a descriptor pointer. It instead calls pmwcas_read (Algorithm 3 lines 20–34) that reads the word and checks whether it contains a descriptor pointer. If so, the function then helps complete the operation using complete_install or persistent_mwcas (Algorithm 2) depending on the descriptor type. It then retries reading the word and returns when it contains a regular value. Similar to pcas_read in Algorithm 1, the reader must also flush the target word if the dirty bit is set, either on a descriptor pointer or normal value.

2) Precommit: Upon completing Phase 1, a thread persists the target words whose dirty bit is set (lines 20–22 of Algorithm 2). To ensure correct recovery, this must be done before updating status and advancing to Phase 2. We update status with CAS to either Succeeded or Failed (with the

dirty bit set) depending on whether Phase 1 succeeded or failed (line 25 of Algorithm 2). Next, the thread persists the `status` word and clears its dirty bit (lines 26–29 of Algorithm 2). Persisting the `status` field "commits" the operation, ensuring its effects survive even across power failures.

C. Phase 2: Completing the MwCAS

If Phase 1 succeeds, the `PMwCAS` is guaranteed to succeed, even if a failure occurs—recovery will roll forward with the new values recorded in the descriptor. If Phase 1 succeeded, Phase 2 installs the final values (with the dirty bit set) in the target words, replacing the pointers to the descriptor `md` (lines 32–38 of Algorithm 2). Since the final values are installed one by one using a `CAS`, it is possible that a crash in the middle of Phase 2 leaves some target words with new values, while others point to the descriptor. Another thread might have observed some of the newly installed values and make dependent actions (e.g., performing a `PMwCAS` of its own) based on the read. Rolling back in this case might cause data inconsistencies. Therefore, it is crucial to persist `status` before entering Phase 2. The recovery routine can then rely on the `status` field of the descriptor to decide if it should roll forward or backward. The next section provides details of the recovery process.

If the `PMwCAS` fails in Phase 1, Phase 2 becomes a rollback procedure by installing the old values (with the dirty bit set) in all target words containing a descriptor pointer.

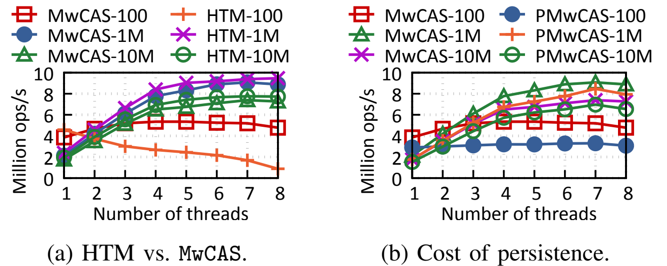
D. Recovery

Due to the two-phase execution of `PMwCAS`, a target word may contain a descriptor pointer or normal value after a crash. Correct recovery requires that the descriptor be persisted before entering Phase 1. The dirty bit in the `status` field is cleared because the caller has not started to install descriptor pointers in the target words; any failure that might occur before this point does not affect data consistency upon recovery.

We maintain a pool of descriptors in an application-specified NVRAM location. Upon restart from a failure, recovery starts by scanning the whole descriptor pool and processes each in-flight operation. As we will discuss in Section V, 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.

Recovery is straightforward: if a descriptor's `status` equals `Succeeded`, roll the operation forward; if it equals `Failed` or `Undecided`, roll the operation back; otherwise do nothing. For each descriptor `md`, we iterate over each target word and check if it contains a pointer to `md` or to the corresponding word descriptor. If either is the case, the old value is applied to the word if `md.status` equals `Undecided` or `Failed`; the new value is applied otherwise (i.e., when `md.status` equals `Succeeded`). We then free memory pointed to by the word descriptor's expected and desired values according to the specified policy (see Section V). The `status` field is then set to `Free` and the descriptor is ready for reuse.

In summary, using a fixed pool of descriptors enables the recovery procedure to easily find all in-flight `PMwCAS` operations after a crash. Persisting the descriptor before entering Phase 1 ensures that the operation can be correctly completed and persisting the `status` field after Phase 1 makes it possible to correctly decide whether to roll the operation forward or back. In terms of performance, an implementation of BzTree (a B+-Tree that uses the `PMwCAS` for persistence on NVRAM) recovered in 145 μs after crashing with 48 worker threads executing the YCSB workload [19]. Such performance illustrates that `PMwCAS` helps support near instantaneous recovery when used to implement non-trivial access methods.



(a) HTM vs. `MwCAS`. (b) Cost of persistence.

Fig. 2: Microbenchmark results that compare HTM based and software MwCAS (a), and show persistence overhead (b).

E. Discussion

`MwCAS` and transactional memory (TM) [20] provides similar guarantees. Recent hardware TM (Intel TSX [9]) has made it more appealing as TSX overcomes the high overhead in software TM. But it is not directly applicable to NVRAM: in some processor implementations TSX always aborts transactions that flush cache lines using `CLWB` [9]. Nevertheless, HTM can be used for Phase 1 of `MwCAS`, simplifying its implementation. Next we use microbenchmarks evaluate both approaches, and show the cost of persistence added by `PMwCAS`. In Section VII we conduct more realistic index benchmarks.

We use a TSX-enabled workstation equipped with an Intel Xeon E3-1245 v3 processor clocked at 3.4GHz and 32GB of RAM. The processor has four physical cores (eight hyperthreads) with 256KB/1MB/8MB L1/L2/L3 caches, respectively. Although the workstation has very limited parallelism, it is enough for us to reason about the relative merits of different designs. We expect similar conclusions on larger machines. The workload tries to modify multiple random words in a fixed-size array using `MwCAS`, `PMwCAS` or HTM based `MwCAS` (denoted as "HTM"). We employ three high bits in each of these words for the flags needed by `PMwCAS` and `MwCAS`.

Figure 2a compares `MwCAS` and HTM-based `MwCAS`. We vary the array size (100–10 million) to adjust contention level: smaller arrays indicate higher contention. Note that the 100-entry array occupies only 800 bytes of memory and will be fully cache-resident. Therefore, with 1–2 threads (very low contention) HTM-100 and MwCAS-100 could outperform the other variants where data may not be fully cache-resident. The 1M and 10M arrays occupy 8 and 80 MB of memory, respectively. As a result, MwCAS-1M/HTM-1M outperform MwCAS-10M/HTM-10M because most entries in the 1M

array are cached. Overall, HTM outperforms MwCAS under low contention (1M and 10M arrays) by ∼3–12% but falls behind by up to more than 80% with 8 threads on the 100-entry array. These results confirm the well-known observation that TSX is vulnerable to high contention [21].

Figure 2b shows PMwCAS's persistence cost by comparing it with MwCAS. PMwCAS caused ∼25–40% slow down under high contention (100-entry array). Note that this is the worst scenario and exaggerates the overhead, as the array is so small that it will be completely cache-resident. The results also exhibited similar trends for numbers between 100, 1M, and 10M arrays to those discussed previously. The overhead for 1M and 10M arrays is ∼7–17% and ∼6–15%, respectively.[3] We also observed that on a larger machine with 64 threads (details in Section VII, it does not support TSX) that PMwCAS adds a constant amount of overhead on top of MwCAS, without impeding its scalability. These results show that PMwCAS itself is not an expensive operation, compared to MwCAS. Section VII continues our evaluation using index benchmarks.

V. Storage Management

The NVRAM space stores both descriptors and user data, i.e., the data structures being maintained, in our case, indexes. However, indexes cannot use PMwCAS efficiently without proper storage management of dynamically allocated NVRAM. Words modified by PMwCAS often store pointers to memory acquired from a persistent allocator [22], [23]. The memory allocated should be owned by either the allocator or the data structure and not be left "hanging" after a crash. We designed PMwCAS to help avoid such memory leaks. Next we first detail descriptor management, and then discuss how PMwCAS ensures safe transfer of memory ownership.

A. Descriptor Management

We maintain a pool of descriptors in a dedicated area on NVRAM. The pool need not be big: it only needs to be large enough to support a maximum number of concurrent threads accessing a data structure (usually a small multiple of the hardware thread count). This scheme has several benefits. First, it aids recovery by having a single location to quickly identify PMwCAS operations that were in progress during a crash. Second, it gives more flexibility on storage management. The descriptor pool and data areas can be managed differently, depending on the user's choice, e.g., using different allocation strategies.

Allocation. Most lock-free data structures (including non-trivial ones like the Bw-Tree and a doubly-linked skip list) only require a handful (2–4) of words to be changed atomically. We thus fix the maximum number of target addresses in each descriptor. This allows us to treat the descriptor pool as a fixed sized array. With this scheme we can also support various descriptor size classes, with each class maintaining a different number of max target addresses. In this case we maintain a fixed-size array for each class. We divide descriptor allocation lists into per-thread partitions and only allow threads to "borrow" from other partitions if its list is depleted.

Descriptor recycling. One thorny issue in lock-free environments is detecting when memory can be safely reclaimed. In our case, we must be sure that no thread dereferences a pointer to a descriptor (swapped out in Phase 2) before we reclaim its memory. We use an epoch-based resource management approach [24] to recycle descriptors. Any thread must enter an epoch before dereferencing descriptors. The epoch value is a global value maintained by the system and advanced by user-defined events, e.g., by memory usage or physical time. After Phase 2, when the descriptor pointer has been removed from all target addresses, we place its pointer on a garbage list along with the value of the current global epoch, called the *recycle epoch*. The descriptor remains on the garbage list until all threads have exited epochs with values less than the descriptor's recycle epoch. This is sufficient to ensure that no thread can possibly dereference the current incarnation of the descriptor and it is free to reuse. The descriptor being removed from the garbage list first transitions to the Free status. It remains so and does not transition into the Undecided status until it is ready to conduct another PMwCAS. Employing the Free status aids recovery: without it, a crash happened during descriptor initialization will cause the recovery routine to wrongfully roll forward or back.

A nice feature of having a descriptor pool is that garbage lists need not be persistent: they are only needed for safety during multi-threaded execution. Recovery, being single threaded, can scan the entire descriptor pool and does not need to worry about other concurrent threads accessing and changing descriptors.

B. User Data Management

We assume the memory area for user data is managed by a persistent memory allocator, which must be carefully crafted to ensure safe transfer of memory ownership. The problem is best explained by the following C/C++ statement for allocating eight bytes of memory: `void *p = malloc(8);`. At runtime, the statement is executed in two steps: (1) the allocator reserves the requested amount of memory and (2) stores the address of the allocated memory in p. Step (2) transfers the ownership of the memory from the allocator to the application. When step 2 finishes, the application owns the memory. A naive implementation that simply stores the address in p could leak memory if a failure happens before p is persisted in NVRAM or if p is in DRAM. After a crash, the system could end up in a state where a memory block is "homeless" and cannot be reached from neither the application nor the allocator.

One solution is breaking the allocation process into two steps: *reserve* and *activate*, which allocates memory and transfers its ownership to the application, respectively [22]. The allocator ensures crash consistency internally for the reservation step. This is opaque to the application and is not the focus of this paper. However, the application must carefully interact with the allocator in the activation process, through an interface (provided by the allocator) that is similar to posix_memalign [17]

[3] We use a separate background process to maintain a shared memory space as "NVRAM." The application attaches to it upon start. This allows us to test recovery, but costs extra resources, as shown by performance drop with eight threads for PMwCAS variants. With real NVRAM such drop will not ensue.

TABLE II: Recycle policies and example usages. With epoch-based reclamation, the same epoch manager provides pointer stability for both data and descriptor. The "free" operations only happen after no thread is using the memory to be recycled.

Policy	Meaning	Example Usage
None	No recycling needed.	Change non-pointer values.
FreeOne	Free the old (or new) value memory if the PMwCAS succeeded (or failed).	Install a consolidated page in the Bw-tree.
FreeNewOnFailure	Free the new value memory if PMwCAS failed; do nothing if succeeded.	Insert a node into a linked list.
FreeOldOnSuccess	Free the old value memory if PMwCAS succeeded; do nothing if failed.	Delete a node from a linked list.

which accepts a reference of the target location for storing the address of the allocated memory. This design is employed by many existing NVRAM systems [13], [15], [22], [23], [25]. The application owns the memory only after the allocator has successfully persisted the address of the newly allocated memory in the provided reference.

Building a safe and correct persistent allocator is out of the scope of this paper. Our focus is making PMwCAS work with existing allocators that expose the above interface, to guarantee safe memory ownership transfer. Without PMwCAS, a lock-free data structure would use the persistent CAS primitive described in Section III and must handle possible failures in step 2. Since this approach does not guarantee safe transfer of memory ownership, it could significantly increase code complexity.

1) Safe Memory Ownership Transfer in PMwCAS: To avoid memory leaks we use PMwCAS descriptors as temporary owners of allocated memory blocks until they are incorporated into the application data structure. As described earlier, we assume an allocation interface similar to posix_memalign [17] that passes a reference of the target location for storing the address of the allocated memory. In our case we require that the application pass to the allocator the address of the new_value field in the word descriptor of the target word. Memory is owned by the descriptor after the allocator has persistently stored the address of the memory in the new_value field.

During recovery, the memory allocator runs its recovery procedure first. We assume that allocator recovery results in every pending allocation call being either completed or rolled back. As a result, all the "delivery addresses" contain either the address of an allocated memory block or a null pointer. After the allocator's recovery phase, we begin PMwCAS's recovery mechanism to roll forward or back in-flight PMwCAS operations as described in Section IV-D.

Reclamation. Lock-free data structures must support safe memory reclamation, given that deallocation is not protected by mutual exclusion. In other words, threads can dereference a pointer to a memory block even after it has been removed from a data structure [26]. By allowing the application to piggyback on our descriptor recycling framework, we free the application from implementing its own memory reclamation mechanism.

In lock-free implementations, memory chunks pointed to by the old_value or new_value fields normally do not acquire new accesses if the PMwCAS succeeded or failed, respectively. We allow an application to specify a *memory recycling policy* for each target word. The policy defines how the memory pointed to by the old_value and new_value fields should be handled when the PMwCAS concludes and no thread can dereference the corresponding memory (based on the epoch

```
1. palloc(p1, size);
2. palloc(p2, size);
        +
Complex, error-prone
   recovery code
```

(**a**) CAS-based approach.

```
1. Descriptor *d = AllocateDescriptor();
2. p1 = d->ReserveEntry(a1, o1, FreeOne);
3. palloc(p1, size);
4. p2 = d->ReserveEntry(a2, o2, FreeOne);
5. palloc(p2, size);
```

(**b**) With PMwCAS, no custom recovery logic.

Fig. 3: Comparison of CAS and PMwCAS based approaches.

safety guarantee discussed previously). The policy is stored in an additional field in the word descriptor. The different recycling options are described in Table II.

Instead of specifying per-word policies, the application can provide a customized "finalize" function to be called when a descriptor is about to be recycled. This is useful for applications that need more control over the memory deallocation process. For example, instead of simply calling free() on a memory object, an object-specific destructor needs to be called.

Example. Figure 3 shows an example of allocating and installing two words using (a) a single-word persistent CAS and (b) PMwCAS. In Figure 3(b), the application first allocates a PMwCAS descriptor (line 1) and then reserves a slot in the descriptor using ReserveEntry (lines 2 and 4). ReserveEntry works exactly the same as AddWord except that it does not require the application to pass the new value and will return a reference (pointer) to the new_value field of the newly added entry. The reference is further fed to the allocator (lines 2 and 5) for memory allocation. The application also specifies a FreeOne recycle policy when calling ReserveEntry: if the PMwCAS succeeded, then the memory pointed to by the old_value field will be freed (respecting epoch boundaries); otherwise the new_value will be freed.

2) Discussion: Our approach frees the application from implementing its own memory recycling mechanism, which tends to be complex and error-prone. Typically, the application only needs to (1) allocate a PMwCAS descriptor, (2) initialize each word descriptor using ReserveEntry and specify a recycling policy, (3) pass a reference to the newly-allocated entry's new_value field to the allocator, (4) initialize the newly allocated memory object and, when all word descriptor have been filled in, (4) execute the PMwCAS. When the PMwCAS concludes, the dynamically-allocated memory associated with it will be recycled according to the recycle policy specified. No application-specific code is needed.

PMwCAS makes it easier to transform a volatile data structure into a persistent one without having to write application-specific recovery code; the only requirement is the application use PMwCAS to transform the underlying data structure from

one consistent state to another, which is usually the case. Upon recovery, PMwCAS's recovery procedure proceeds as Section IV-D describes, and deallocates memory that is no longer needed. ***The application needs no explicit recovery code, nor memory management code during recovery***. The only limitation is that one must use PMwCAS even if the operation is single-word in nature for safe memory ownership transfer. For the volatile case, however, one is free to use single-word CAS to avoid the overhead of maintaining descriptors.

VI. Case Studies

Now we demonstrate the use of PMwCAS to simplify the implementation of two highly concurrent indexes on NVRAM: a doubly-linked skip list [27] and the Bw-tree [5]. We use key-sequential access methods since they are ubiquitous (all databases need to support range scans efficiently). They also require non-trivial efforts to achieve high performance. Of course, the use of PMwCAS applies beyond indexing; one can use it to ease the implementation of any lock-free protocol that requires atomically updating multiple arbitrary memory words.

A. Doubly-Linked Skip List

Overview. A skip list can be thought of as multiple levels of linked lists. The lowest level maintains a linked list of all records in key-sequential order. Higher level lists consist of a sparser subsequence of keys than levels below. Search starts from the top level of a special head node, and gradually descends to the desired key at the base list in logarithmic time. To implement a lock-free singly-linked (unidirectional) skip list, a record is inserted into the base list using a CAS. At this point the record is visible since it will appear in a search of the base list. If the new key must be promoted to higher-level lists, this can be done lazily [27].

Implementation complexity. It is straightforward to implement a lock-free singly-linked skip list, but it comes at a price: reverse scan is often omitted or supported inefficiently. Some systems "remembers" the predecessor nodes in a stack during forward scan and use it to guide a reverse scan. A more natural solution is making the skip list *doubly-linked*, with a *next* and *previous* pointer in each node. While efficient, this requires complex hand-in-hand CAS operations at each level [28].

Using CAS to implement lock-free doubly-linked skip lists is complicated and error-prone. The state-of-the-art method first uses a CAS to insert a node at each level, setting up only the *next* pointers. A second phase then tries to install the *previous* pointers using a series of CAS operations [28]. The complexity comes from the second phase having to detect races with simultaneous inserts and deletes that interfere with the installation of the *previous* pointer. If such a race is detected, the implementation must fix up and retry the operation. A majority of the code from this approach is dedicated handling such races. Earlier designs [29], [30], [31] often sacrifice features (e.g., deletion) for easier implementation.

Implementation using PMwCAS. In our implementation, each node points to its predecessor and successor in the same level, and to the lower level node in the same tower. Inserting (deleting) a node involves first inserting (deleting) in the base level, and then inserting (deleting) upper level nodes containing the record key. For a volatile implementation, one can use PMwCAS (with persistence guarantees disabled) to atomically install a node n in each doubly-linked list by atomically updating the *next* pointer at n's predecessor and *previous* pointer at n's successor. Compared to our CAS based implementation, the PMwCAS based variant reduced the lines of code by 24% and is almost as easy as a lock-based one, evidenced by a 43% reduction on cyclomatic complexity.[4]

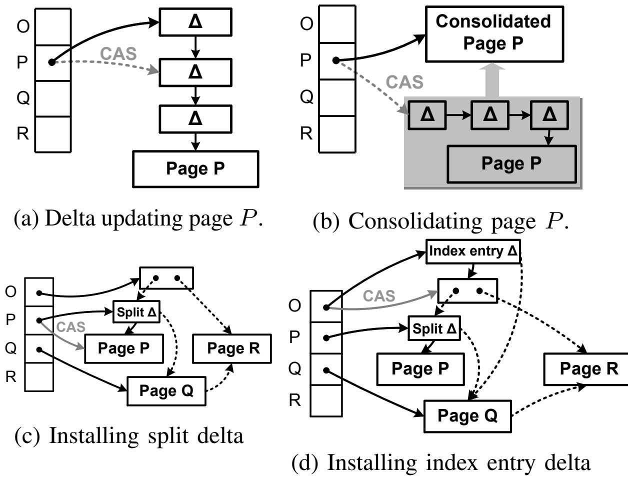
The transition from volatile to persistent implementation is seamless. The core insert/delete logic remains the same, but with additional memory management code described in Section V-B. All index nodes must be allocated using a persistent allocator to ensure persistence and proper ownership handoff. Since PMwCAS always transforms the skip list from one consistent state to another, we use the default recovery and memory reclamation mechanisms (Section V) to maintain data consistency across failures. No customized recovery code is needed. For a new node insertion, we use the FreeNewOnFailure policy to ensure the new node is reclaimed if the PMwCAS fails. For delete, we use FreeOldOnSuccess to recycle the deleted node after the PMwCAS succeeds.

B. The Bw-Tree

Overview. The Bw-tree [5] is a lock-free B+-tree. It maintains a mapping table that maps logical page identifiers (LPIDs) to virtual addresses. All links between Bw-tree nodes are LPIDs; a thread traversing the index must use the mapping table to translate LPIDs to pointers. The Bw-tree uses copy-on-write to update pages. An update creates a *delta record* describing the update and prepends it to the target page. Deltas are installed using a CAS that replaces the current page address in the mapping table with the address of the delta. Figure 4a depicts a delta update to page P; the dashed line represents P's original address, while the solid line represents P's new address. Pages are consolidated once a number of deltas accumulate on a page to prevent degradation of search performance. Consolidation involves creating a new compact (search-optimized) page with all deltas applied that replaces the old page version using a CAS (Figure 4b).

Implementation complexity. Structure modification operations (SMOs) such as page splits and merges cause complexity in the Bw-tree, since they introduce changes to more than one page and we cannot update multiple arbitrary nodes using a CAS. The Bw-tree breaks an SMO into a sequence of atomic steps; each step is installed using a CAS to a single page. Figure 4c depicts the two-phase split for a page P. Phase 1 selects a separator key K, generates a new sibling page Q and installs a "split delta" on P that logically describes the split and provides a side-link to the new sibling Q. Phase 2 inserts K into the parent node O by posting a delta containing (K, LPID) with a CAS. Deleting and merging pages in the Bw-tree follow a similar process with three atomic steps (details covered in [5]).

[4] A quantitative measure of the number of linearly independent code paths [32].



(a) Delta updating page P.
(b) Consolidating page P.
(c) Installing split delta
(d) Installing index entry delta

Fig. 4: Bw-tree lock-free page update and split. Split is broken into two atomic CAS operations on the mapping table.

While highly concurrent, the Bw-tree contains several subtle race conditions as a result of the SMO protocol. For example, threads can observe "in progress" SMOs, so the implementation must detect and handle such conflicts. A Bw-tree thread that encounters a partial SMO will "help along" to complete it before continuing with its own operation. Also, in-progress SMOs can "collide," and without care lead to index corruption. A prime example is that simultaneous splits and merges on the same page could collide at the parent. This happens, for instance, when a thread t_1 sees an in-progress split of a page P with new sibling Q and attempts to help along by installing a new key/pointer pair for Q at a parent O. In the meantime, another thread t_2 could have deleted Q and already removed its entry at O (which was installed by another thread t_3). In this case t1 must be able to detect the fact that Q was deleted and avoid modifying O. A large amount of code (and thought) is dedicated to detecting and handling subtle cases like these.

Implementation using PMwCAS. We use `PMwCAS` to simplify Bw-tree's SMO protocol and reduce the subtle races just described. The approach "collapses" the multi-step SMO into a single `PMwCAS`. We use page split as a running example; a page delete/merge follows a similar approach. For a volatile implementation, a split of page P first allocates a new sibling page, along with memory for *both* the split and index deltas. It then uses `PMwCAS` (with persistence disabled) to atomically install the split delta on P and the index delta at the parent. The split may trigger further splits at upper levels, in which case we repeat this process for the parent. `PMwCAS` allows us to cut all the help-along code in the `CAS` based implementation and reduces cyclomatic complexity of SMOs by 24%.

For a persistent implementation, similar to the skip list case, the logic for the SMOs remains the same, but the code must conform to correct memory-handling procedures. For each new page allocated (the new page Q along with split and index deltas), we reserve a slot in the descriptor and pass the persistent allocator a reference to the reserved slot's `new_value` field. For reclamation, we use `FreeNewOnFailure` to ensure the new memory is recycled if the `PMwCAS` fails.

Some Bw-tree SMOs are single-word in nature, e.g., installing a delta or consolidating a page. In the volatile case, we can safely use `CAS` as long as it does not use `PMwCAS`'s flag bits. But in the persistent case using `CAS` loses the safe persistence guarantee as the transfer of memory ownership will be unknown to the descriptor. Therefore, we use `PMwCAS` even for single-word updates for the persistent Bw-tree implementation.

VII. Evaluation

A. Experimental Setup

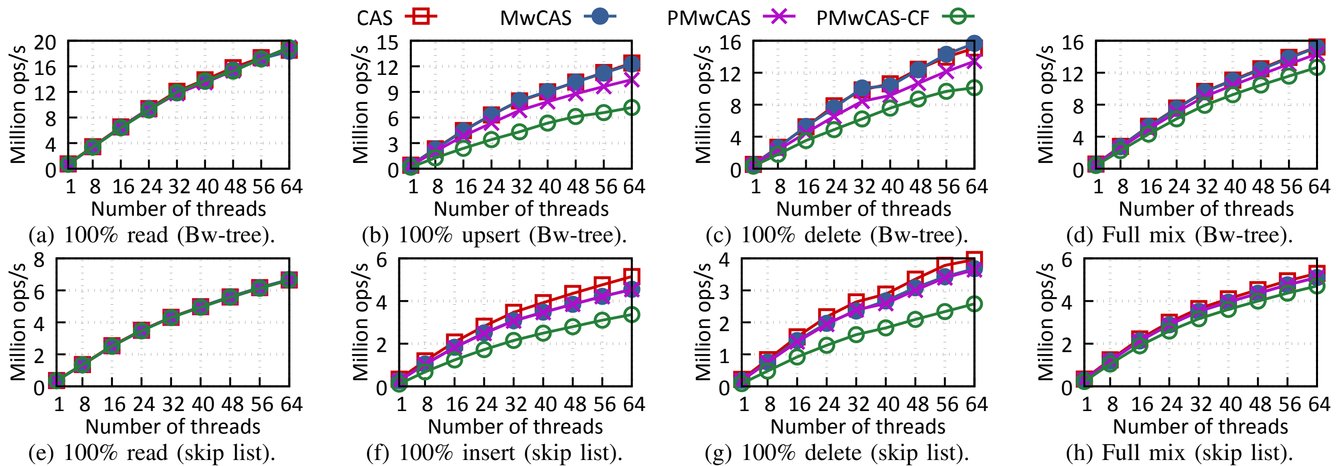
The goal of our experiments is to evaluate the impact of synchronization mechanism on index structures. We run experiments on a quad-socket machine with four Intel Xeon E5-4620 processors clocked at 2.2GHz and 512GB of main memory. Each CPU has eight physical cores and 256KB/2MB/16MB L1/L2/L3 caches, respectively. With hyper-threading the server gives in total 64 hardware threads. We target flash-backed NVDIMMs [1] which exhibits exactly the same performance characteristics as DRAM at runtime. So we run all experiments in normal DRAM. Each run is repeated three times and we report the average across these runs.

Intel is expected to release 3D XPoint DIMMs during 2018. At the time of writing, we do not have access to it. While the underlying memory technology is slower than DRAM, the overall effect on performance is difficult to estimate. The write latency, as seen by an application, should be the same as DRAM because data only has to make it to the memory controller's write buffer to be "safe" (though an application may generate more write traffic to ensure that modifications are persisted). Read latency, as seen by an application, is expected to be 3-5x higher than DRAM but some of this will be mitigated by caches; to what extent depends on cache and working set sizes. A more detailed evaluation will have to wait until real hardware becomes available.

B. Workloads and Index Implementations

We implement the Bw-tree and skip list as standalone record stores that support insert/upsert, delete, get, and scan. We vary the percentage of each operation in the benchmark to test individual and mixed operations. The mixed workload follows a 4:1 read/write ratio and a 4:1 point read/range scan ratio (20% write, 64% get and 16% scan). We use 8-byte keys, 8-byte values, and initialize both indexes with 10 million records. The `PMwCAS` uses a descriptor size class of four (the max size needed for both indexes) and a decriptor pool size of 1K.

Following the designs described in Section VI, we implemented and evaluate three variants of the Bw-tree and doubly-linked skip list using `CAS`, volatile `MwCAS`, and `PMwCAS`. The `CAS` and `MwCAS` variants are completely volatile. While the `PMwCAS` variant guarantees persistence, it does not issue any `CLWB` instruction as described in previous algorithms because the processor does not support it. To give an *upper bound* of the persistence overhead, we show the performance of a "`PMwCAS-CF`" variant, which is the same as `PMwCAS` except that it issues `CLFLUSH` to "persist" target words. `CLFLUSH` evicts the cache line while writing it back to NVRAM, and future



(a) 100% read (Bw-tree). (b) 100% upsert (Bw-tree). (c) 100% delete (Bw-tree). (d) Full mix (Bw-tree).

(e) 100% read (skip list). (f) 100% insert (skip list). (g) 100% delete (skip list). (h) Full mix (skip list).

Fig. 5: Bw-tree (a–d) and skip list (e–h) performance with 8-byte keys and 8-byte values.

processors are expected to feature the CLWB instructions that do not evict the cache line [9]. With PMwCAS-CF, we measure the worst-case overhead for persistence.

C. Bw-Tree Results

This section compares the impact of synchronization primitives on the Bw-tree. Figures 5a–5c show the results for read, upsert and delete operations, respectively. All the variants show similar performance for the read-only workload (Figure 5a). Since the workload is read-only, compared to the CAS-based variant, the only extra work needed by MwCAS and PMwCAS is setting/checking the flags bits. Such overhead is minuscule; we observed similar results for skip list in Section VII-D.

Figure 5b plots the throughput for upserts as we vary the number of concurrent threads on the x-axis. As the figure shows, MwCAS is relatively cheap for multi-word operations. The only SMO that changes multiple words atomically in an upsert-only workload is page split. Recall that for volatile MwCAS in the Bw-tree, we employ CAS internally to conduct single-word operations. Therefore, the performance difference exhibited by CAS and MwCAS is purely due to multi-word operations. As the figure shows, the overhead of MwCAS is small (at most ∼2%). Supporting persistence using PMwCAS requires one use single-entry PMwCAS, adding ∼15% of overhead compared to MwCAS, regardless of the amount of concurrency across the x-axis. Therefore, PMwCAS's persistence machinery adds a fixed amount of overhead and does not affect the scalability of MwCAS. Finally, by comparing the performance of PMwCAS and PMwCAS-CF, we observed that CLFLUSH could degrade throughput by more than 30%. This is the worst case scenario and underlines the need of a high-performance cache line write-back mechanism, such as CLWB.

Figure 5c summarizes the delete-only benchmark, where each thread deletes an equal amount of randomly-chosen records, leaving an empty tree when all threads finish. The overall trend is similar to what we observed in the upsert-only benchmark, however, the margin is smaller. Figure 5d gives the throughput for the mixed workload. Since the workload has more reads, it exhibits smaller differences among the evaluated variants. Compared to the best-performing CAS variant, PMwCAS incurs on average ∼6% of overhead. We believe the significant ease of programming efforts justify such low overhead.

These results show that for the Bw-tree, MwCAS imposes little overhead (but largely simplifies the implementation). PMwCAS adds persistence without changing the index code for an existing MwCAS based variant, and does so by adding a small overhead.

D. Skip List Results

The skip list experiments use the same workload settings as those used in the Bw-tree experiments. Figures 5e–5g depict the throughput for individual operations. The relative trends between different synchronization primitives are similar to those for the Bw-tree. All variants perform similarly under the pure-read workload. For workloads that consist of inserts and/or deletes (Figures 5f–5g), CAS is consistently faster than MwCAS by an average of ∼12%, while in the Bw-tree experiments MwCAS and CAS have similar performance. The reason is that unlike the Bw-tree in which insert/delete involves only appending a delta record that can be done via a single-word CAS, both insert and delete must atomically change *two* pointers atomically at each level, thus making the skip list unable to use CAS internally for inserts or deletes and save the cost of maintaining MwCAS descriptors and the flag bits. The PMwCAS variant again exhibits similar trend to that in the Bw-tree experiments. Compared to the CAS-based variant, they perform on average 13% slower. With a realistic workload (Figure 5h), such difference becomes smaller (∼3–8%). Similar to what we have seen in previous Bw-tree experiments, the CLFLUSH impact is large, stressing the need for instructions like CLWB.

VIII. Related Work

Multi-word CAS. We demonstrate MwCAS's usefulness on lock-free indexes for both volatile and non-volatile memory. To our knowledge, PMwCAS is the first MwCAS proposal for NVRAM. We based it on the volatile MwCAS by Harris et al. [6] because of its simplicity and good performance. Other MwCAS designs exist [7], [8], but are either considerably more complex or do not perform as well as the design by [6].

Transactional memory. Compared to software TM [33], HTM exhibits much lower overhead, but often comes with

constraints that limit its usefulness, such as limited transaction size [21]. We also explored HTM drawbacks in Section IV-E.

Lock-free indexes. Lock-free indexes are usually built with single-word CAS and must deal with various races as a result concurrent accesses. We focused on easing the difficulty of implementing doubly-linked skip list [27], [28], [34], [35] and the Bw-tree [5]. Masstree [36] is a trie of B+-trees designed to achieve good cache behavior and scalability. ART [37], [38] is also a trie-based memory-optimized index.

Persistent indexes. Chen et al. [39] presented and analyzed metrics for B+-trees on NVRAM. The concurrent CDDS [14] B-tree avoids logging using global version numbers. The wB+-Tree [11] reduces the amount of cache line flushes by only keeping the leaf nodes sorted. NV-Tree [16] also only keeps the leaf nodes sorted but re-constructs internal index nodes when needed. The FPTree [13] is a hybrid index that puts leaf nodes in NVRAM and internal nodes unsorted in DRAM.

NVRAM systems. Most systems use logging and require custom recovery logic [12], [15], [40]. In contrast, PMwCAS employs the dirty bit design and avoids logging using pooled descriptors. Izraelevitz et al. [41], [42] proposed a framework for reasoning about the correctness of data structures in NVRAM and a single-word CAS based mechanism for building persistent data structures. PMwCAS can work with any NVRAM allocator [22], [23], [43], [25] that provides proper interfaces to guarantee safe transfer of memory ownership.

IX. Conclusion

Building lock-free data structures is a challenging task, especially for NVRAM. Traditional approaches rely on single-word CAS and implement custom recovery logic in addition to handling complex races. Our contribution is a persistent multi-word compare-and-swap (PMwCAS) primitive that can atomically change multiple 8-byte words in a lock-free manner. PMwCAS provides a middle ground between CAS and lock-based programming, with the former's high performance and the latter's ease of use. PMwCAS provides safe persistence guarantees, and frees the developer from devising complex and error-prone recovery logic needed in existing NVRAM systems. Moreover, the same index implementation can be used for both the DRAM and NVRAM resident indexes. We adapted the volatile CAS-based Bw-tree and doubly-linked skip list to NVRAM using PMwCAS. The result is competitive performance with persistence guarantees, transparent recovery support, as well as code that is easy to maintain and reason about.

References

[1] Viking Technology, "Viking DDR4 NVDIMM," 2017, http://www.vikingtechnology.com/products/nvdimm/ddr4-nvdimm/.

[2] R. Crooke and M. Durcan, "A revolutionary breakthrough in memory technology," *Intel 3D XPoint launch keynote*, 2015.

[3] M. Hosomi *et al.*, "A novel nonvolatile memory with spin torque transfer magnetization switching: spin-RAM," *IEEE IEDM*, pp. 459–462, 2005.

[4] A. Prout, "The story behind MemSQL's skiplist indexes," 2014, http://blog.memsql.com/the-story-behind-memsqls-skiplist-indexes/.

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

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

[7] P. H. Ha and P. Tsigas, "Reactive multi-word synchronization for multiprocessors," *J. Instruction-Level Parallelism*, vol. 6, 2004.

[8] H. Sundell, "Wait-free multi-word compare-and-swap using greedy helping and grabbing," *International Journal of Parallel Programming*, vol. 39, no. 6, pp. 694–716, 2011.

[9] Intel Corporation, "Intel® 64 and IA-32 architectures software developer's manuals," 2016.

[10] C. Diaconu *et al.*, "Hekaton: SQL Server's memory-optimized OLTP engine," in *SIGMOD*, 2013, pp. 1243–1254.

[11] S. Chen and Q. Jin, "Persistent B+-trees in non-volatile main memory," *PVLDB*, vol. 8, no. 7, pp. 786–797, Feb. 2015.

[12] J. Coburn *et al.*, "NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," *ASPLOS*, 2011.

[13] I. Oukid *et al.*, "FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory," *SIGMOD*, pp. 371–386, 2016.

[14] S. Venkataraman *et al.*, "Consistent and durable data structures for non-volatile byte-addressable memory," *FAST*, 2011.

[15] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," *ASPLOS*, pp. 91–104, 2011.

[16] J. Yang *et al.*, "NV-Tree: Reducing consistency cost for NVM-based single level systems," *FAST*, pp. 167–181, 2015.

[17] IEEE and The Open Group, "The Open Group Base Specifications Issue 7, IEEE Std 1003.1," 2016.

[18] Microsoft, "InterlockedCompareExchange function," *MSDN*, 2017.

[19] J. Arulraj, J. Levandoski, U. F. Minhas, and P.-A. Larson, "BzTree: A high-performance latch-free range index for non-volatile memory," *PVLDB*, vol. 11, no. 5, Jan. 2018.

[20] D. B. Lomet, "Process structuring, synchronization, and recovery using atomic actions," in *ACM LDRS*, 1977, pp. 128–137.

[21] D. Makreshanski, J. Levandoski, and R. Stutsman, "To lock, swap, or elide: On the interplay of hardware transactional memory and lock-free indexing," *PVLDB*, vol. 8, no. 11, pp. 1298–1309, Jul. 2015.

[22] Intel Corporation, "NVM library," *http://www.pmem.io*, 2016.

[23] D. Schwalb, T. Berning, M. Faust, M. Dreseler, and H. Plattner, "nvm malloc: Memory allocation for NVRAM," *ADMS*, 2015.

[24] H. T. Kung and P. L. Lehman, "Concurrent manipulation of binary search trees," *ACM TODS*, vol. 5, no. 3, pp. 354–382, Sep. 1980.

[25] I. Oukid *et al.*, "Memory management techniques for large-scale persistent-main-memory systems," *PVLDB*, 2017.

[26] M. M. Michael, "Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects," *IEEE TPDS*, vol. 15, no. 6, pp. 491–504, 2004.

[27] W. Pugh, "Skip lists: A probabilistic alternative to balanced trees," *Commun. ACM*, vol. 33, no. 6, pp. 668–676, Jun. 1990.

[28] H. Sundell and P. Tsigas, "Lock-free deques and doubly linked lists," *JPDC*, vol. 68, no. 7, pp. 1008–1020, Jul. 2008.

[29] H. Attiya and E. Hillel, "Built-in coloring for highly-concurrent doubly-linked lists," *DISC*, pp. 31–45, 2006.

[30] M. Greenwald, "Two-handed emulation: How to build non-blocking implementations of complex data-structures using DCAS," *PODC*, 2002.

[31] J. D. Valois, "Lock-free data structures," Ph.D. dissertation, RPI, 1995.

[32] T. J. McCabe, "A complexity measure," *ICSE*, 1976.

[33] N. Shavit and D. Touitou, "Software transactional memory," *PODC*, 1995.

[34] K. Fraser, "Practical lock-freedom," Ph.D. dissertation, University of Cambridge, 2004.

[35] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," *DISC*, pp. 300–314, 2001.

[36] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," *EuroSys*, pp. 183–196, 2012.

[37] V. Leis, A. Kemper, and T. Neumann, "The adaptive radix tree: ARTful indexing for main-memory databases," *ICDE*, pp. 38–49, 2013.

[38] V. Leis, F. Scheibner, A. Kemper, and T. Neumann, "The ART of practical synchronization," *DaMoN*, pp. 3:1–3:8, 2016.

[39] S. Chen, P. B. Gibbons, and S. Nath, "Rethinking database algorithms for phase change memory," *CIDR*, 2011.

[40] A. Chatzistergiou *et al.*, "REWIND: recovery write-ahead system for in-memory non-volatile data-structures," *PVLDB*, 2015.

[41] J. Izraelevitz, H. Mendes, and M. L. Scott, "Linearizability of persistent memory objects under a full-system-crash failure model," *DISC*, 2016.

[42] ——, "Brief announcement: Preserving happens-before in persistent memory," *SPAA*, 2016.

[43] K. Bhandari *et al.*, "Makalu: Fast recoverable allocation of non-volatile memory," *OOPSLA*, pp. 677–694, 2016.