Easy Lock-Free Indexing in Non-Volatile Memory

Tianzheng Wang*
University of Toronto
tzwang@cs.toronto.edu

Justin Levandoski
Microsoft Research
justin.levandoski@microsoft.com

Per-Ake Larson
University of Waterloo
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 and persistent. 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. The `PMwCAS` operation 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 the runtime overhead introduced by `PMwCAS` is very low (~4–6% under realistic workloads). This overhead is sufficiently low that the same implementation can be used for both DRAM-resident indexes and NVRAM-resident indexes. Not requiring separate implementations for both cases is a salient feature of our design and greatly reduces the cost of code maintenance.

1 INTRODUCTION

Fast, byte-addressable non-volatile memory (NVRAM) devices are currently coming online in the form of NVDIMM [28, 39], Intel 3D XPoint [7], and STT-MRAM [14]. 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 will change how high-performance main-memory databases implement durability and recovery. Today, DRAM is the primary home for records in a main-memory database. However to guarantee durability, the system takes checkpoints and logs updates to secondary storage, much like a disk-based system. During recovery the in-memory database (records and indexes) is rebuilt from the on-disk state. Persisting both records and indexes on NVRAM, however, opens up the opportunity for almost *instant* recovery that requires only a small amount of work (bounded by the number of active transactions during the crash) before the database is online and active. Much recent work has investigated such an approach [5, 30, 38].

A primary challenge in this space is implementing persistent, high-performance indexes. Several commercial main-memory systems implement lock-free indexes to be able to fully exploit the hardware parallelism in modern machines: MemSQL uses lock-free skip lists [31], while Microsoft Hekaton uses the Bw-tree [22], a lock-free B+-tree. In this paper we show how lock-free indexes can be made persistent without sacrificing their high performance.

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., for B+-tree page splits and merges). 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, while NVRAM is durable, 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 the modified words. `PMwCAS` greatly simplifies the implementation of lock-free data structures. Using `PMwCAS`, the developer specifies the memory words to modify along with the expected and desired values for each (similar to a single-word `CAS`). The `PMwCAS` operation will either atomically install all new values or fail the operation without exposing intermediate state (e.g., a partially completed operation) to the user. This behavior is guaranteed across a power failure as well.

The `PMwCAS` operation is the main contribution of this paper and what enables us to build high-performance, lock-free indexes for NVRAM. Furthermore, persistence across failures is guaranteed without requiring any logging or special recovery logic in the index code (Section 4.4). `PMwCAS` is 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 [13] to which we added persistence guarantees and support for recovery. There are two other versions of volatile `MwCAS` operations [11, 35] but they are either slower and/or more complex than the version by Harris et al.

To show how `PMwCAS` eases engineering complexity and code maintenance, we detail the implementation of two high-performance lock-free indexes.

*Work performed while at Microsoft Research.

MSR Tech Report, August 2017, Redmond, WA, USA
2017. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$0.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

- *A doubly-linked lock-free skip list.* Skip lists are used in a number of research and production main-memory databases, e.g., MemSQL [31]. We detail how to implement a persisted skip list that uses forward and backward links for dual-direction range scans.
- *The Bw-tree [22].* The Bw-tree is the lock-free B+-tree used by SQL Server Hekaton [8]. The complexity in the Bw-tree stems from breaking structure modifications (e.g., node splits) that touch multiple nodes into separate atomic steps, each done using a single-word `CAS`.

Using `PMwCAS`, we substantially reduced code complexity for both indexes. For example, we cut lines of code for the skip list by ~24%. We substantially reduced the code path of the Bw-tree by eliminating code that addresses race conditions when a thread sees an incomplete node split or merges initiated by another thread. Such simplification makes the code almost as mechanical as using lock-based programming, but with the performance benefits of lock-free data indexes.

We also provide an extensive experimental evaluation of our techniques. Using microbenchmarks, we show that `PMwCAS` performs robustly and is efficient even under highly-contended workloads. When running 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%. The overhead is sufficiently low that the same implementation can be used for both volatile DRAM-resident indexes *and* NVRAM-resident indexes, greatly reducing the cost of code maintenance.

The rest of this paper is organized as follows. Section 2 covers our system assumptions and the `PMwCAS` interface and behavior. Section 3 describes a single-word persisted `CAS` operation as a basis for understanding the full `PMwCAS` operation covered in Section 4. Section 5 covers issues related to NVRAM management. Section 6 describes our index implementations using `PMwCAS`, while Section 7 provides our experimental evaluation. Related work is covered in Section 8, and Section 9 concludes the paper.

2 BACKGROUND

2.1 System Model

As illustrated in Figure 1, we assume a system with a single-level store where NVRAM is attached directly to the memory bus. This model was also adopted by several recent NVRAM based systems [5, 6, 29, 38, 40, 41]. 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 [16]. 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. While the `CLWB` instruction currently ships on current Intel CPUs [16], it appears to not yet be fully implemented.

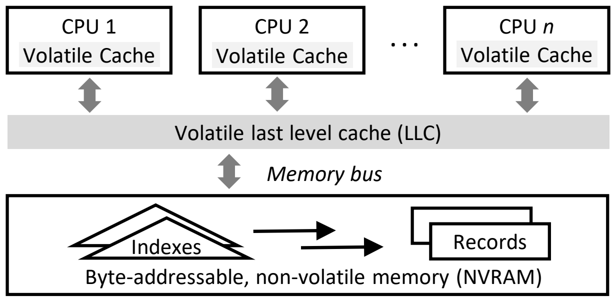


Figure 1: Single-level system with NVRAM on the memory bus. Access to NVRAM is cached by multiple levels volatile CPU caches. Both indexes and data reside in NVRAM.

2.2 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 = default)`: 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 5).
- `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`). No specified word will be modified.

The API is identical for both volatile and persistent `MwCAS`. Under the hood, `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. It can use `RemoveWord` 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. 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 kept in sorted order on the address field to prevent deadlock. During execution of the `PMwCAS`, the first phase in effect attempts to "lock" each target word. From concurrency control theory we know that 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 5). The user need not worry about descriptor memory. PMwCAS is most often used to update pointers to dynamically allocated memory. The callback parameter is provided if the user wishes 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 also specify 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 [15]). Section 5 discusses the details behind memory management.

2.3 The Benefits of PMwCAS

The PMwCAS operator has several salient features that make it attractive for lock-free programming in an NVRAM environment, especially for implementing high-performance indexes.

Easier programming. PMwCAS greatly simplifies the design and implementation of high performance lock-free code. The two indexing techniques we consider in this paper (double-linked skip list and the Bw-tree) are much easier to implement by using PMwCAS. They—and non-trivial lock-free code in general—require atomic operations that span multiple words. In our experience, implementing atomic operations that require updating multiple words using only single-word CAS often results in complex and subtle code that is very hard to reason about. With PMwCAS, the implementation is almost as mechanical as a locked based one.

Persistence guarantees. PMwCAS guards against tricky persistence 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 readers only see persistent values.

Less code with transparent recoverability. PMwCAS allows for the same index implementation to be used in both volatile DRAM as well as NVRAM with hardly any change. This reduces code complexity, simplifies code maintenance, and more importantly, allows one to transform a volatile data structure to a persistent one *without* application-specific recovery code. Internally, PMwCAS ensures crash consistency, as long as the application's use of PMwCAS transforms the data structure from one consistent state to another. This has always been the case in our experience with the two indexing structures and is easy to achieve for most applications.

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, since subtle leaks might occur if the system crashes in the midst of an operation. For instance, a new node that was allocated but not yet added to the index will be leaked when the system crashed, unless care is taken. Index implementations can easily piggyback on the lock-free recycling protocol used by PMwCAS to ensure that memory is safely reclaimed after the success (or failure) of the operation and even after a crash.

Robust compared to HTM. Recent hardware transactional memory (HTM) [16] 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 that is potentially complex. HTM also has the potential of easing the implementation of MwCAS itself: one could use a single HTM transaction for MwCAS's Phase 1, but again we found this approach does not perform robustly; software-based MwCAS yields similar but much more robust performance, as Section 7.4 details. We therefore focus on pure software-based implementation in this paper.

3 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 [16]) 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.

Most CAS operations operate on word-aligned pointers, so certain lower bits in the operands are always zero. For example, the lower two bits are always zero if the address is at least 4-byte aligned. Modern 64-bit x86 processors employ a "canonical address" design [16], 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. 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.

Algorithm 1 shows how single-word 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 [16] instruction (lines 3–4 and 13). Note that at line 14, 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

[1]Various system events (e.g., cacheline replacement) could implicitly persist the word.

Algorithm 1 A persistent single-word CAS.

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

def persistent_cas(address, old_value, new_value):
  pcas_read(address)
  # Conduct the CAS with dirty bit set on new value
  return CAS(address, old_value, new_value | DirtyFlag)

def persist(address, value):
  CLWB(address)
  CAS(address, value, value & ~DirtyFlag)
```

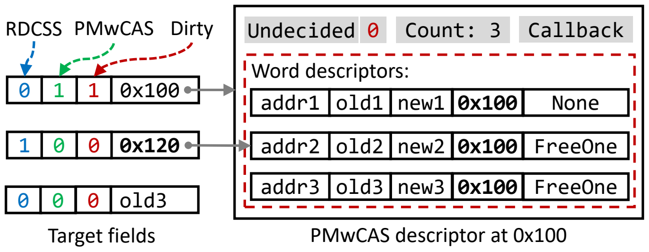

RDCSS
PMwCAS
Dirty
0 1 1 0x100
1 0 0 0x120
0 0 0 old3
Target fields
Undecided 0
Count: 3
Callback
Word descriptors:
addr1 old1 new1 0x100 None
addr2 old2 new2 0x100 FreeOne
addr3 old3 new3 0x100 FreeOne
PMwCAS descriptor at 0x100

Figure 2: Flag bits employed in target fields (left) and an example PMwCAS descriptor (right). Callbacks and memory policy (word descriptor's right-most field) are discussed in Section 5.

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 field 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.

4 PERSISTENT MULTI-WORD CAS

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

Users of PMwCAS first allocate a descriptor using the API in Section 2.2, and add per-word modifications using either the AddWord (in the case of 8-byte updates) or ReserveEntry (to install pointers to memory blocks larger than 8-bytes). The user performs the operation by issuing the PMwCAS command (or Discard if they wish to cancel). If the PMwCAS operation acknowledges success, the users is guaranteed that all the target words were updated atomically and will persist across power failures. On failure, the user is guaranteed that none of the updates are visible to other threads.

4.1 PMwCAS Overview

Our design is based on the volatile multi-word CAS (MwCAS) by Harris et al [13] which we enhance to work correctly also when data is stored in NVRAM.

The PMwCAS operator uses a descriptor that describes the operation to be performed and tracks it status. Figure 2 (ignore the target fields 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*. 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 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. The policy field 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.

The example descriptor in Figure 2 is currently in the initial Undecided status and looking to change three words at addr1, addr2, and addr3. All three word descriptors contain a back pointer to the descriptor at address 0x100 and policy specification.

The execution of a PMwCAS operation consists of two phases:

- Phase 1: Install a pointer to the descriptor in all target addresses.
- Phase 2: If Phase 1 succeeded, Phase 2 then installs the new values in all target addresses. If Phase 1 failed, then Phase 2 resets any target word that points to the descriptor back to its old value.

Another concurrent 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 more detail. Algorithms 2 provides the entry point to PMwCAS . Since PMwCAS is cooperative, Algorithm 3 provides the entry point for readers (pmwcas_read), along with two common 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.

4.2 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 4.2.1). It then ensures persistence of the descriptor pointer writes before determining the final operation status (Section 4.2.2).

For each target word in the descriptor mdesc, PMwCAS first attempts to install a pointer to mdesc in each target word (Algorithm 2 lines 3–8). The installation uses a two-phase process called RDCSS [13] that performs a double compare and a single swap to install the descriptor. RDCSS is necessary to guard against subtle race conditions and maintain a linearizable sequence of operations on the same target address. Specifically, we must guard against the

Algorithm 2 PMwCAS.

```
    bool PMwCAS(mdesc):
2     st = Succeeded
      for w in mdesc.words:
4     retry:
        rval = install_mwcas_descriptor(w)
6       if rval == w.old_value or rval & AddressMask == mdesc:
          # Descriptor successfully installed
8         continue
        elif rval & MwCASFlag is not 0:
10        if rval & DirtyFlag is not 0:
            persist(w.address, rval)
12        # Clashed another on-going MwCAS, help it finish
          persistent_mwcas(rval.Address)
14        goto retry
        else
16        st = Failed
          break
18
      # Persist all target fields if Phase 1 succeeded
20    if st == Succeeded:
        for w in mdesc.words:
22        persist(w.address, mdesc | MwCASFlag | DirtyFlag)

24    # Finalize the MwCAS's status
      CAS(mdesc.status, Undecided, st | StatusDirtyFlag)
26    if mdesc.status & DirtyFlag:
        CLWB(&mdesc.status)
28      mdesc.status &= ~DirtyFlag

30    # Install the final values
      for w in mdesc.words:
32      val = mdesc.status == Succeeded ? w.new_value : w.old_value
        expected = mdesc | MwCASFlag | DirtyFlag
34      rval = CAS(w.address, expected, val | DirtyFlag)
        if rval == mdesc | MwCASFlag:
36        CAS(w.address, expected & ~DirtyFlag, val)
        persist(w.address, val)
38    return mdesc.status == Succeeded
```

Algorithm 3 Read and help-along routines for PMwCAS.

```
    def pmwcas_read(address):
2   retry:
      v = *address
4     if v & RDCSSFlag:
        complete_install(v & AddressMask)
6       goto retry

8     if v & DirtyFlag:
        persist(address, v)
10      v &= ~DirtyFlag

12    if v & MwCASFlag:
        persistent_mwcas(v & AddressMask)
14      goto retry
      return v
16
    def install_mwcas_descriptor(word):
18    ptr = word | RDCSSFlag
    retry:
20    val = CAS(word.address, word.old_value, ptr)
      if val & RDCSSFlag is not 0:
22      # Hit a descriptor, help it finish
        complete_install(val & AddressMask)
24      goto retry

26    if val == desc.old_value:
        # Successfully installed the conditional CAS descriptor
28      complete_install(word)
      return val
30
    def complete_install(wdesc):
32    mwcas_ptr = wdesc.mwcas_descriptor | MwCASFlag | DirtyFlag
      u = wdesc.mwcas_descriptor.status == Undecided
34    CAS(word.address, val, u ? mwcas_ptr : wdesc.old_value)
```

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 address 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 actually be overwriting 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.

The function install_mwcas_descriptor (lines 17–29 of Algorithm 3) is invoked for each target word in the PMwCAS descriptor. It receives the address of a word descriptor as the sole parameter and returns the value found in the target word. It first uses a single-word CAS to install a pointer to the *word* descriptor (with the RDCSSFlag flag set) in the target word (lines 18–20). If the target word already points to a word descriptor, the caller helps complete the corresponding RDCSS and then retries its own RDCSS (lines 21–24). If the CAS succeeds, it proceeds to set the target word to point to the descriptor if status is Undecided (lines 26–28 and 31–34). If the PMwCAS has finished (status contains Succeeded or Failed), the installation fails and the target word is reset to the old value. Note that at line 12, we toggle the dirty bit when installing the descriptor to ensure correct recovery (discussed later).

Figure 2 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 field contains a descriptor pointer and the field content might not be durable on NVRAM. The second target address, however, still points to its word descriptor whose address is 0x120 [2]. Therefore, for this field the caller could be executing lines 21–28 of Algorithm 3. The last target field is yet to be changed and still contains the old value.

The result of the call to install_mwcas_descriptor (at line 5 of Algorithm 2) returns one of the following values when trying to install a pointer to descriptor mdesc. (1) A regular value that equals the expected old value, signalling success. (2) A regular value that does *not* equal the expected old 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 value to mdesc, meaning another thread successfully completed the installation. (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 mdesc.

[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.

In all cases, if the return value's dirty bit is set, we persist the field using the `persist` function defined in Algorithm 1.

4.2.1 Reading Affected Words. Phase 1 exposes pointers to a full descriptor or individual word descriptors to any thread reading one of the target words. Similar to the volatile `MwCAS` [13], a thread does not directly read words that may contain a descriptor pointer but instead calls `pmwcas_read` (lines 1–15 in Algorithm 3). `pmwcas_read` reads the word and checks whether it contains a descriptor pointer. If it does, the function then helps complete the operation by calling `complete_install` (lines 29–34 in Algorithm 3) or `persistent_mwcas` (Algorithm 2) depending on the descriptor type. It then retries reading the field and returns when the field contains a regular value. As shown on the left side of Figure 2, we use three vacant bits to indicate whether a word contains a pointer to a word descriptor, a pointer to a descriptor, and whether the value might not be persisted. They are represented in Algorithm 3 by `RDCSSFlag`, `MwCASFlag`, and `DirtyFlag`, which are constants with only the corresponding bit set. Similar to the `pcas_read` function 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.

4.2.2 Precommit. Upon completing Phase 1, a thread then 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 the `status` field and advancing to Phase 2. We update the `status` field using `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 26–28 of Algorithm 2). Persisting the `status` field "commits" the operation, ensuring its effects survive even across power failures.

4.3 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 `mdesc` (lines 31–37 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 fields 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.

4.4 Recovery

Due to the two-phase execution of `PMwCAS`, a target address 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 fields; any failure that might occur before this point does not affect data consistency upon recovery.

We maintain a pool of descriptors within the NVRAM address space at a location predefined by the application. 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 5, 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 quite straightforward: if a descriptor's `status` field 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 field 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 (details in section 5). 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.

5 STORAGE MANAGEMENT

The NVRAM space is used for storing descriptors and user data, i.e., the data structures being maintained, in our case, indexes. Words modified by `PMwCAS` often store pointers to memory acquired from a persistent allocator [17, 33]. 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.

5.1 Descriptor Management

We maintain a pool of descriptors in a dedicated area on NVRAM. The descriptor 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. We provide a `PMwCAS` space analysis in Appendix B.

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 [19] 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 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 to guarantee 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.

5.2 User Data Management

We assume the memory area for user data is managed by a persistent memory allocator. The allocator 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) store the address of the allocated memory in `p`. Step (2) transfers the ownership of the memory block 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 [17]. The allocator ensures crash consistency internally for the reservation step, which 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` [15] 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 [17, 29, 33, 40]. 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. Instead, we focus on making `PMwCAS` work with existing allocators that expose the above activation interface, to guarantee safe memory ownership transfer. Without `PMwCAS`, a lock-free data structure would use the persistent `CAS` primitive described in Section 3 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.

5.2.1 *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` [15] 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 block 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 4.4.

Reclamation. Lock-free data structures must support some form of 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 [27]. 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 safety guarantee discussed previously). The policy is stored in an additional field in the word descriptor. The different recycling options are described in Table 1.

Rather than providing customized per-word policies, the application can provide a customized "finalize" function that will be called when a descriptor is about to be recycled. This is useful in scenarios where the application needs 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 8-byte words using (a) a single-word persistent `CAS` and

Table 1: 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.

```
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.

Figure 3: Allocating and installing two 8-byte words using single-word CAS (a) and PMwCAS (b).

(b) PMwCAS. As shown 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 AddEntry 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 both old_value fields will be freed (respecting epoch boundaries); otherwise the new_values will be freed.

5.2.2 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 4.4 describes, and deallocates memory that is no longer needed. The application needs no explicit recovery code, nor memory management code during recovery. The limitation of our approach 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. Another issue is support for relative vs. absolute pointers, which we discuss in Appendix A.

6 CASE STUDIES

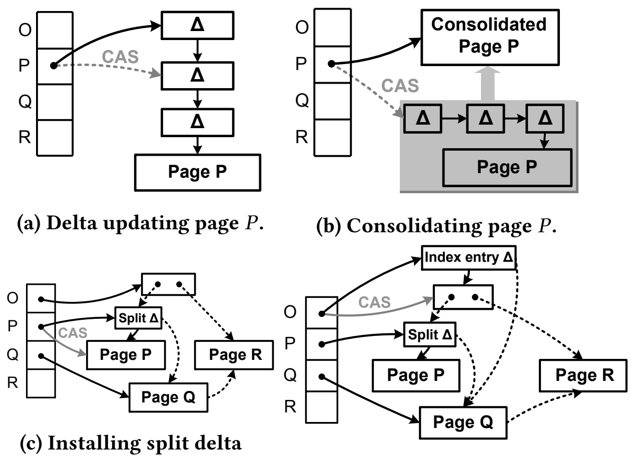
Now we demonstrate the use of PMwCAS to simplify the implementation of highly concurrent indexes on NVRAM. We focus on two lock-free range indexes: a doubly-linked skip list [32] and the Bw-tree [22] used in Microsoft Hekaton [8]. We use key-sequential access methods since they are ubiquitous (all databases need to support range scans efficiently). They also require non-trivial implementation effort to achieve high performance; these implementations are usually lock-free in modern main-memory systems. 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.

6.1 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 single-word 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 [32].

Implementation complexity. While a lock-free singly-linked skip list is easy to implement, it comes at a price: reverse scan is often omitted or supported inefficiently. Some systems "remembers" the predecessor nodes in a stack during forward scans and use it to guide a reverse scan. A more natural way to support reverse scan is making the skip list *doubly-linked*, with a *next* and *previous* pointer in each node. While efficient, this approach requires complex hand-in-hand CAS operations list at each level [36].

Common solutions to implementing lock-free doubly-linked skip lists using a single-word CAS are complicated and error-prone. The state-of-the-art method first inserts a record at each level as if inserting into a singly linked list (making a predecessor point to its new successor). A second phase then tries to install *previous* pointers from successor to new predecessor using a series of CAS operations [36]. The complexity of this approach 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 [1, 10, 37] often sacrifice features (e.g., deletion) for easier implementation.



(a) Delta updating page *P*.

(b) Consolidating page *P*.

(c) Installing split delta

(d) Installing index entry delta

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

Implementation using PMwCAS. We build the doubly-linked skip list using multiple levels of lock-free doubly-linked lists [36]. 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 specifying the two pointers to atomically update: the *next* pointer at *n*'s predecessor and *previous* pointer at *n*'s successor. Compared to our CAS based implementation, we reduced the lines of code by 24%. PMwCAS makes the implementation almost as easy as a lock-based one, evidenced by a 43% reduction on cyclomatic complexity [26].

The transition from volatile to persistent implementation on NVRAM is seamless. The core insert/delete logic remains the same, but with additional memory management code described in Section 5.2. If inserting a node, the implementation must allocate the node using a persistent allocator to ensure persistence and proper ownership handoff. Upon allocating the PMwCAS descriptor, we add the appropriate values to its new_value and old_value fields. Since PMwCAS always transforms the skip list from one consistent state to another, we can use the default recovery and memory reclamation mechanisms (Section 5) to maintain data consistency across failures. No special-purpose recovery routine is needed. For a new node insertion, one can use the "FreeNewOnFailure" policy to ensure the new node memory is reclaimed in case the PMwCAS fails. For delete, one would specify the "FreeOldOnSuccess" policy to recycle the deleted node after the PMwCAS succeeds.

6.2 The Bw-Tree

Overview. The Bw-tree [22] 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, meaning a thread traversing the index must use the mapping table to translate each LPID to a page pointer. 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 single-word 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 delta updates 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 single-word 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 an appropriate 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 follows a similar process with three atomic steps (details covered in [22]).

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 the 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 the Bw-tree 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 can then use the 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.

MwCAS allows us to cut all the help-along code in the CAS based implementation and reduces cyclomatic complexity of SMOs by 24%. MwCAS makes the code much easier to reason about and less error-prone with a simpler control flow.

The transition from volatile to persistent implementation is seamless. The logic for the SMOs remains the same. However, in addition the code must conform to memory-handling procedures described in Section 5.2, starting with allocating a PMwCAS descriptor. Then, for each new memory 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 memory reclamation, we use the "FreeNewOnFailure" policy that will recycle this new memory if the PMwCAS fails. The process of merging two pages works similarly to the split, by specifying the required mapping table entries to change and relying on PMwCAS's memory safety guarantee.

Certain Bw-tree operations are single-word in nature, e.g., installing a delta record or consolidating a page. In the volatile case, we can safely use CAS in the presence of PMwCAS as long as the flag bits needed by PMwCAS are not used by CAS. But in the persistent case installing delta records with CAS loses the safe persistence guarantee provided by PMwCAS 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.

7 EVALUATION

7.1 Experimental Setup

The goal of our experiments is to evaluate the impact of synchronization mechanism on index structures. We run index 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 also use microbenchmarks to compare MwCAS and PMwCAS's raw performance, and evaluate an HTM based MwCAS (detailed in Section 7.4). Unfortunately, the quad-socket server does not support HTM, so for microbenchmarks we use a workstation that has Intel's hardware transactional memory extension (TSX) enabled. The workstation is equipped with an Intel Xeon E3-1245 v3 processor clocked at 3.4GHz and 32GB of main memory. The processor has four physical cores (eight hyper-threads) with 256KB/1MB/8MB L1/L2/L3 caches, respectively. Although the workstation for HTM experiments has very limited parallelism, we show that such configuration is enough for us to reason about the relative merits of different designs. We expect similar conclusions on larger machines.

We target flash-backed NVDIMMs [39]. At runtime, they exhibit exactly the same performance characteristics as DRAM, so we run all experiments in normal DRAM. In certain runs, we use the CLFLUSH instruction to "persist" words in NVRAM. CLFLUSH evicts the cache line while writing it back to NVRAM, and future processors are expected to feature instructions such as CLWB [16] that do not evict the cache line. With CLFLUSH, we actually measure the worst-case overhead for persisting words. Each run is repeated three times and we report the average across these runs.

7.2 Workloads

Microbenchmarks. We stress test PMwCAS to understand its performance characteristics under write-heavy scenarios with varying degrees of contention. The microbenchmark operates on a fixed-size, statically-allocated array, and tries to modify multiple random words using a single PMwCAS (or its volatile counterpart).

We also measure the overhead imposed by PMwCAS's persistence guarantee over volatile MwCAS. Intel TSX [16] provides an efficient way to atomically change multiple words with low overhead. We built an HTM-based MwCAS that uses an HTM transaction to finish Phase 1 (denoted as "HTM"), and compare it with software MwCAS.

Index benchmarks. Both the Bw-tree and skip list are implemented 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, consisting of 20% of writes (upserts for the Bw-tree, inserts for skip list), 64% of get and 16% of scan. We use 8-byte keys, 8-byte values, and initialize both indexes with ten million records.

7.3 Index Implementations

We evaluate implementations of the Bw-tree and skip list based on four synchronization primitives described as follows.

CAS. The CAS variants follow the protocols described in Section 6 and are completely volatile without persistence guarantees.

MwCAS. Certain Bw-tree operations (update, delete, and page consolidation) only require single-word CAS,we still use CAS for these operations for optimized performance. We use MwCAS for other SMOs that change multiple words. We did not use the same approach for the MwCAS-based skip list as both insert and delete operations involve atomically changing at least two words.

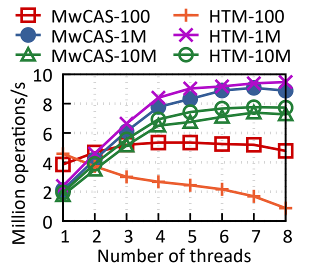
Persistent CAS (PCAS). Without application-specific code, the PCAS primitive described in Section 3 does not guarantee safe transfer of memory ownership or recoverability. For fair comparison, we implemented PCAS-based indexes using PMwCAS that always conducts a single-word MwCAS. We use PMwCAS's safe memory ownership guarantee for memory allocation and recycling, however, this does not provide recoverability without custom recovery logic.

PMwCAS. This variant guarantees safe persistence and recoverability. To ensure correctness and consistency, we use single-entry PMwCAS for update, delete and page consolidation in the Bw-tree.

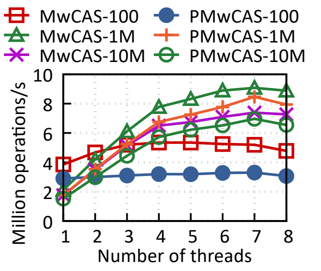
PCAS-CF and PMwCAS-CF. These two variants are exactly the same as PCAS and PMwCAS, except that they use CLFLUSH to "persist" the word that is being changed. CLFLUSH evicts the cache line during write-back, thus is not an ideal choice for persisting word in NVRAM. Nevertheless, CLFLUSH should give an upper bound for the overhead caused by persisting words in NVRAM.

7.4 Comparison with HTM-based MwCAS

Figure 5a compares the throughput (number of successful operations per second) of volatile MwCAS and HTM-based MwCAS using our microbenchmark that changes four random words. We vary the array size (100–10 million) to adjust contention level: smaller arrays indicate higher contention. As the figure shows, 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 a 100-entry array. Such result confirms the well-known observation that TSX is vulnerable to high contention [24]. Software-based MwCAS performs robustly across all contention levels and outperforms HTM under high contention. Therefore, we focus on evaluating software-based volatile MwCAS and PMwCAS in the rest of this section.



(a) HTM outperforms MwCAS under low contention (e.g., 10M array), but is vulnerable to high contention (100-entry array).



(b) High contention exaggerates the cost of persistence, but it can be as low as < 10% when data is not fully cached by CPU (~8MB).

Figure 5: Microbenchmark results that compare HTM based and software MwCAS (a), and show persistence overhead (b).

7.5 Cost of Persistence

Now we evaluate the overhead added by PMwCAS on top of volatile MwCAS to support persistence. Figure 5b shows the performance of MwCAS and PMwCAS modifying four 8-byte words randomly-chosen from pre-allocated arrays of varying sizes. The y-axis shows the number of successful MwCAS operations per second. Under high contention (100-entry array), adding persistence caused ~25–40% slow down. High contention exaggerates the persistence overhead as the array is so small that it will be completely cache-resident. Thus, this is a worst-case scenario and uncommon in real life workloads. On arrays with 1M and 10M entries, the overhead is ~7–17% and ~6–15%, respectively. As we discuss later, PMwCAS exhibits very small overhead (~6%) under realistic workloads, following the trend shown by larger arrays here. Although not shown here, we also measured the cost of persistence in PMwCAS on the quad-socket machine with up to 64 threads. The result shows a similar trend for scalability and persistence overhead. Our persistence mechanism adds a constant amount of overhead on top of volatile MwCAS, without impeding its scalability.

7.6 The Bw-Tree

This section compares the impact of different synchronization primitives, starting with individual index operations. Figures 6a–6c show the results for read, upsert and delete operations, respectively. All the variants show similar performance for the read-only workload (Figure 6a). 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. As the figure shows, such overhead is minuscule; we observed similar results for skip list in Section 7.7.

Figure 6b 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 needs to change 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. PCAS performs similarly to PMwCAS as we use single-entry PMwCAS for safe memory ownership transfer. But PCAS is not safe without custom recovery logic. Finally, by comparing the performance between PMwCAS vs. PMwCAS-CF, as well as PCAS vs. PCAS-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.

In the delete-only benchmark, each thread deletes an equal amount of randomly-chosen records, leaving an empty tree when all threads finish. As Figure 6c shows, the overall trend and relative differences between synchronization primitives are similar to what we observed in the upsert-only benchmark, however, the margin is smaller. Figure 6d gives the throughput for a realistic, mixed workload (including both point reads and range scans). 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.

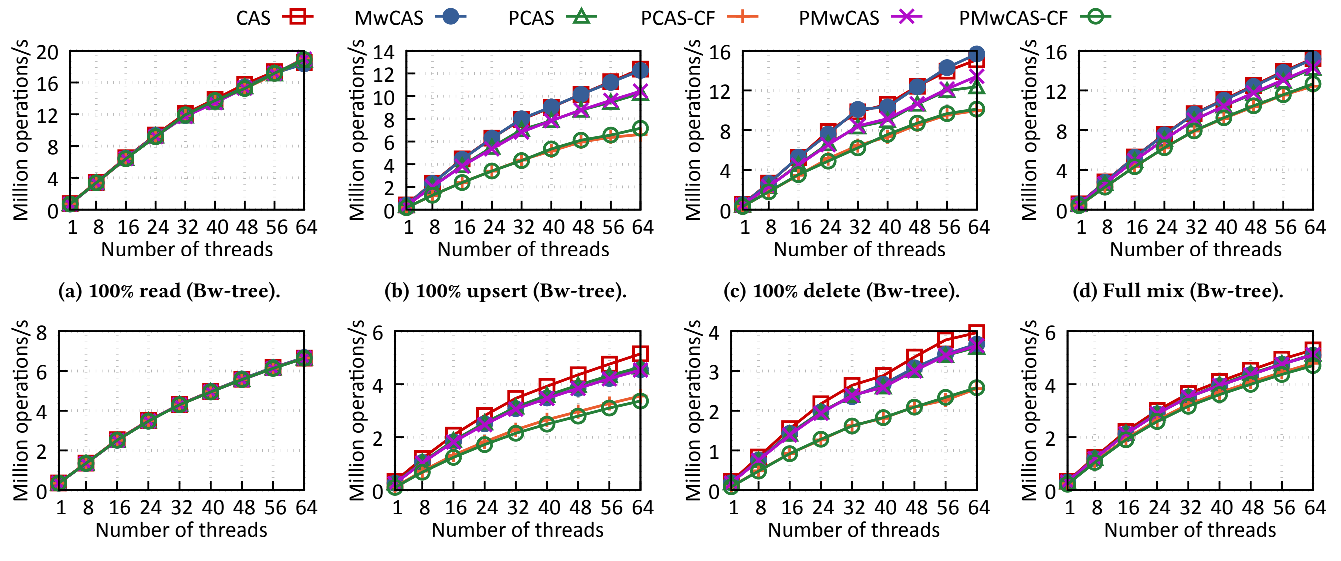
These results show that for the Bw-tree, MwCAS imposes little overhead (but largely simplifies the implementation). PMwCAS adds persistence without changing the code for an existing MwCAS based variant, and does so by adding a small overhead.

7.7 Skip List

The skip list experiments use the same workload settings as those used in Bw-tree experiments. Figures 6e–6g 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 6f–6g), 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 persistent variants using PCAS and PMwCAS again exhibit 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 6h), such difference becomes smaller (~3–8%). Again, the impact of CLFLUSH remains large for skip list, as we have seen in previous experiments, stressing the need for instructions such as CLWB.

8 RELATED WORK

Multi-word CAS. MwCAS simplifies lock-free programming while still maintaining high performance. This makes MwCAS useful in building of high-performance data structures. We have demonstrated its usefulness on lock-free indexes for both volatile and non-volatile memory. To the best of our knowledge, our PMwCAS is the first implementation of MwCAS for NVRAM. We based it on the



(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).

Figure 6: Bw-tree (a–d) and skip list (e–h) performance with 8-byte keys and 8-byte values.

volatile MwCAS designed by Harris et al [13] because of its simplicity and good performance. Other MwCAS designs exist [11, 35], however, they are either considerably more complex or do not perform as well as the design by Harris et al [13].

Transactional memory. MwCAS and transactional memory [23] have a lot in common: all operations in an MwCAS or a transaction either happen in their entirety or not at all. Most work focuses on software transactional memory [34], however, its high overhead has prevented wide adoption. HTM [18, 42] exhibits much lower overhead and has the potential of simplifying lock-free design, but often comes with constraints that limit its usefulness, such as spurious aborts due to transaction size [24]. We also explored HTM drawbacks in Section 7.4.

Lock-free indexes. The adoption of large main memory and multi-core CPUs has heightened the need for high-performance, lock-free indexes, usually built with single-word CAS. These implementations must deal with various races as a result concurrent accesses from different threads. We focused on easing the difficulty of implementing doubly-linked skip lists [9, 12, 32, 36] and the Bw-tree [22], two indexing techniques used widely in practice. We also showed that using PMwCAS, the volatile implementation of these indexes can seamlessly port to NVRAM environments. Other state-of-the-art indexing techniques include Masstree [25]: a trie of B+-trees designed to achieve good cache behavior and scalability. ART [20, 21] is also a trie-based memory-optimized index.

Persistent indexes. NVRAM has lead to a number of persistent indexing proposals. Much work has focused on enhancing the lifetime of NVRAM. Chen et al [4] presented and analyzed metrics for B+-trees on NVRAM. The CDDS [38] B-tree is a persistent and concurrent tree that uses versioning to avoid logging, but relies on global version numbers. The wB+-Tree [5] reduces the amount of cache line flushes by only keeping the leaf nodes sorted. NV-Tree [41] also only keeps the leaf nodes sorted but re-constructs internal index nodes when needed. The FPTree [29] is a hybrid index that puts leaf nodes in NVRAM and internal nodes unsorted in DRAM, with fingerprints to aid the search of internal nodes. It uses HTM for DRAM nodes and fine-grained locks for leaf nodes to avoid HTM transaction aborts due to cache line flushes.

NVRAM runtime and systems. Most NVRAM proposals rely on logging for correct persistence [3, 6, 17, 40], requiring extra efforts for recovery. In contrast, our approach employs the dirty bit design and avoids logging through use of pooled descriptors. An important line of work is NVRAM allocators [2, 17, 33] that must give the application proper interfaces to handle allocations/deallocations and avoid leaking memory. PMwCAS can work with any persistent allocator that provides such interfaces to guarantee safe transfer of memory ownership.

9 CONCLUSION

Building lock-free data structures is a challenging task, especially for upcoming NVRAM that will change the way main-memory databases implement lock-free indexes. Traditional approaches rely on single-word CAS and must devise 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 specified by the developer in a lock-free manner. PMwCAS provides a middle ground between single-word CAS and lock-based programming, with the former's high performance and the latter's ease of use. PMwCAS also gives safe persistence guarantees, and often times 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, as PMwCAS has an identical API to the volatile MwCAS. 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 much easier to maintain and reason about.

REFERENCES

[1] Hagit Attiya and Eshcar Hillel. 2006. Built-In Coloring for Highly-Concurrent Doubly-Linked Lists. *DISC* (2006), 31–45.

[2] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. 2016. Makalu: Fast Recoverable Allocation of Non-volatile Memory. *OOPSLA* (2016), 677–694.

[3] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Viglas. 2015. REWIND: Recovery Write-Ahead System for In-Memory Non-Volatile Data-Structures. *PVLDB* (2015).

[4] Shimin Chen, Phillip B Gibbons, and Suman Nath. 2011. Rethinking Database Algorithms for Phase Change Memory. *CIDR* (2011).

[5] Shimin Chen and Qin Jin. 2015. Persistent B+-trees in Non-volatile Main Memory. *PVLDB* 8, 7 (Feb. 2015), 786–797.

[6] Joel Coburn et al. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. *ASPLOS* (2011), 105–118.

[7] Rob Crooke and Mark Durcan. 2015. A Revolutionary Breakthrough in Memory Technology. *Intel 3D XPoint launch keynote* (2015).

[8] Cristian Diaconu et al. 2013. Hekaton: SQL Server's Memory-Pptimized OLTP Engine. In *SIGMOD*. 1243–1254.

[9] Keir Fraser. 2004. *Practical lock-freedom*. Ph.D. Dissertation. University of Cambridge.

[10] Michael Greenwald. 2002. Two-handed Emulation: How to Build Non-blocking Implementations of Complex Data-structures Using DCAS. *PODC* (2002).

[11] Phuong Hoai Ha and Philippas Tsigas. 2004. Reactive Multi-word Synchronization for Multiprocessors. *J. Instruction-Level Parallelism* 6 (2004). http://www.jilp.org/vol6/v6paper3.pdf

[12] Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-Lists. *DISC* (2001), 300–314.

[13] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. 2002. A Practical Multi-word Compare-and-Swap Operation. *DISC* (2002), 265–279.

[14] M. Hosomi et al. 2005. A novel nonvolatile memory with spin torque transfer magnetization switching: spin-ram. *IEEE International Electron Devices Meeting (IEDM)* (2005), 459–462.

[15] IEEE and The Open Group. 2016. The Open Group Base Specifications Issue 7, IEEE Std 1003.1. (2016).

[16] Intel Corporation. 2016. Intel® 64 and IA-32 Architectures Software Developer's Manuals. (2016).

[17] Intel Corporation. 2016. NVM Library. *http://www.pmem.io* (2016).

[18] Christian Jacobi, Timothy Slegel, and Dan Greiner. 2012. Transactional Memory Architecture and Implementation for IBM System Z. *MICRO* (2012), 25–36.

[19] H. T. Kung and Philip L. Lehman. 1980. Concurrent Manipulation of Binary Search Trees. *ACM TODS* 5, 3 (Sept. 1980), 354–382.

[20] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The Adaptive Radix Tree: ARTful Indexing for Main-memory Databases. *ICDE* (2013), 38–49.

[21] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of Practical Synchronization. *DaMoN*, Article 3 (2016), 3:1–3:8 pages.

[22] Justin Levandoski, David Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. *ICDE* (2013), 302–313.

[23] D. B. Lomet. 1977. Process Structuring, Synchronization, and Recovery Using Atomic Actions. In *Proceedings of an ACM Conference on Language Design for Reliable Software*. 128–137.

[24] Darko Makreshanski, Justin Levandoski, and Ryan Stutsman. 2015. To Lock, Swap, or Elide: On the Interplay of Hardware Transactional Memory and Lock-free Indexing. *PVLDB* 8, 11 (July 2015), 1298–1309.

[25] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-value Storage. *EuroSys* (2012), 183–196.

[26] Thomas J. McCabe. 1976. A Complexity Measure. *ICSE* (1976).

[27] Maged M. Michael. 2004. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distrib. Syst.* 15, 6 (2004), 491–504.

[28] Netlist. 2016. Storage Class Memory. (2016). http://www.netlist.com/products/Storage-Class-Memory/HybriDIMM/default.aspx.

[29] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. *SIGMOD* (2016), 371–386.

[30] Ismail Oukid, Wolfgang Lehner, Thomas Kissinger, Thomas Willhalm, and Peter Bumbulis. 2015. Instant Recovery for Main Memory Databases. In *CIDR*.

[31] Adam Prout. 2014. The Story Behind MemSQL's Skiplist Indexes. (2014). http://blog.memsql.com/the-story-behind-memsqls-skiplist-indexes/.

[32] William Pugh. 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM* 33, 6 (June 1990), 668–676.

[33] David Schwalb, Tim Berning, Martin Faust, Markus Dreseler, and Hasso Plattner. 2015. nvm malloc: Memory Allocation for NVRAM. *ADMS* (2015).

[34] Nir Shavit and Dan Touitou. 1995. Software Transactional Memory. *PODC* (1995).

[35] Håkan Sundell. 2011. Wait-Free Multi-Word Compare-and-Swap Using Greedy Helping and Grabbing. *International Journal of Parallel Programming* 39, 6 (2011), 694–716. https://doi.org/10.1007/s10766-011-0167-4

[36] Håkan Sundell and Philippas Tsigas. 2008. Lock-free Deques and Doubly Linked Lists. *JPDC* 68, 7 (July 2008), 1008–1020.

[37] John D. Valois. 1995. *Lock-free Data Structures*. Ph.D. Dissertation. Rensselaer Polytechnic Institute.

[38] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory. *FAST* (2011).

[39] Viking Technology. 2013. Viking ArxCis-NV NVDIMM. (2013). http://www.vikingtechnology.com/arxcis-nv.

[40] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. *ASPLOS* (2011), 91–104.

[41] Jun Yang et al. 2015. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. *FAST* (2015), 167–181.

[42] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. 2013. Performance Evaluation of Intel® Transactional Synchronization Extensions for High-performance Computing. *SC*, Article 19 (2013), 19:1–19:11 pages.

A RELATIVE VS. ABSOLUTE POINTERS

We assume the operating system provides access to (byte addressable) NVRAM through the memory mapping interface. The application maps a specific area of NVRAM into its address space using system calls such as `mmap()` (POSIX) and `MapViewOfFileEx()` (Windows). However, the OS cannot guarantee addresses remain identical across remappings. `mmap` and `MapViewOfFileEx` allow an application to specify a target virtual address but the desired address range may already be in use. The application would have to retry (thus possibly restart) or use relative pointers (offsets off the starting address of the NVRAM space) instead of absolute pointers. Relative pointers must be "swizzled" (add offset and base address) at runtime but the cost is typically minimal.

B TOTAL SPACE OVERHEAD

The total space overhead of our `PMwCAS` machinery depends on three factors: the three signal bits (`RDCSS`, `PMwCAS`, and `Dirty`) we "steal" for each target word, the size of the descriptors, and the number of descriptors. First, the signal bits are static. However, if the system cannot spare three bits outside the "canonical" 48-bit address space (on a 64-bit system), our scheme will not work. Second, the size of each descriptor depends on the number of target words in the `PMwCAS` operation. To implement state-of-the-art lock-free indexes, we expect the number of target words to be small: in practice one only needs to update a handful of words (2–4) atomically. Finally, the number of descriptors depends on the number of parallel `PMwCAS` operations executing in parallel; this is bounded by the number of hardware threads available on the machine, plus a small amount of slack space to account for descriptors that are not active but awaiting recycling. In our implementation of both the Bw-tree and skip lists, we encountered negligible size overhead for the `PMwCAS` machinery.