Efficient Multi-word Compare and Swap*

Rachid Guerraoui
EPFL, Switzerland
rachid.guerraoui@epfl.ch

Alex Kogan
Oracle Labs, USA
alex.kogan@oracle.com

Virendra J. Marathe
Oracle Labs, USA
virendra.marathe@oracle.com

Igor Zablotchi[1]
EPFL, Switzerland
igor.zablotchi@epfl.ch

Abstract

Atomic lock-free multi-word compare-and-swap (MCAS) is a powerful tool for designing concurrent algorithms. Yet, its widespread usage has been limited because lock-free implementations of MCAS make heavy use of expensive compare-and-swap (CAS) instructions. Existing MCAS implementations indeed use at least $2k+1$ CASes per k-CAS. This leads to the natural desire to minimize the number of CASes required to implement MCAS.

We first prove in this paper that it is impossible to "pack" the information required to perform a k-word CAS (k-CAS) in less than k locations to be CASed. Then we present the first algorithm that requires $k+1$ CASes per call to k-CAS in the common uncontended case. We implement our algorithm and show that it outperforms a state-of-the-art baseline in a variety of benchmarks in most considered workloads. We also present a durably linearizable (persistent memory friendly) version of our MCAS algorithm using only 2 persistence fences per call, while still only requiring $k+1$ CASes per k-CAS.

2012 ACM Subject Classification Theory of computation → Concurrent algorithms

Keywords and phrases lock-free, multi-word compare-and-swap, persistent memory

Acknowledgements This work has been supported in part by the European Research Council (ERC) Grant 339539 (AOC).

1 Introduction

Compare-and-swap (CAS) is a foundational primitive used pervasively in concurrent algorithms on shared memory systems. In particular, it is used extensively in *lock-free* algorithms, which avoid the pitfalls of blocking synchronization (e.g., that employs locks) and typically deliver more scalable performance on multicore systems. CAS conditionally updates a memory word such that a new value is written if and only if the old value in that word matches some expected value. CAS has been shown to be universal, and thus can implement any shared object in a non-blocking manner [34]. This primitive (or the similar load-linked/store-conditional (LL/SC)) is nowadays provided by nearly every modern architecture.

CAS does have an inherent limitation: it operates on a single word. However, many concurrent algorithms require atomic modification of multiple words, thus introducing significant complexity (and overheads) to get around the 1-word restriction of CAS [10, 19, 25, 26, 41, 48].

* A shorter version of this paper will appear in the proceedings of DISC '20.
[1] This work was done when the author was an intern at Oracle Labs.

As a way to address the 1-word limitation, the research community suggested a natural extension of CAS to multiple words—an atomic multi-word compare-and-swap (MCAS). MCAS has been extensively investigated over the last two decades [4, 5, 19, 25, 26, 33, 34, 47, 55]. Arguably, this work partly led to the advent of the enormous wave of Transactional Memory (TM) research [31, 32, 36]. In fact, MCAS can be considered a special case of TM. While MCAS is not a silver bullet for concurrent programming [21, 35], the extensive body of literature demonstrates that the task of designing concurrent algorithms becomes much easier with MCAS. Not surprisingly, there has been a resurgence of interest in MCAS in the context of persistent memory, where the persistent variant of MCAS (PMCAS) serves as a building block for highly concurrent data structures, such as skip lists and B+-trees [6, 58], managed in persistent memory.

Existing lock-free MCAS constructions typically make heavy use of CAS instructions [4, 33, 47], requiring between 2 and 4 CASes per word modified by MCAS. That resulting cost is high: CASes may cost up to 3.2× times more cycles than load or store instructions [18]. Naturally, algorithm designers aim to minimize the number of CASes in their MCAS implementations.

Toward this goal, it may be tempting to try to "pack" the information needed to perform the MCAS in fewer than k memory words and perform CAS only on those words. We show in this paper that this is impossible. While this result might not be surprising, the proof is not trivial, and is done in two steps. First, we show through a bivalency argument that lock-free MCAS calls with non-disjoint sets of arguments must perform CAS on non-disjoint sets of memory locations, or violate linearizability. Building on this first result, we then show that any lock-free, disjoint-access-parallel k-word MCAS implementation admits an execution in which some call to MCAS must perform CAS on at least k different locations. (Our impossibility result focuses on *disjoint-access-parallel* (DAP) algorithms, in which MCAS operations on disjoint sets of words do no interfere with each other. DAP is a desirable property of scalable concurrent algorithms [39].)

We also show, however, in the paper that MCAS can be "efficient". We present the first MCAS algorithm that requires $k+1$ CAS instructions per call to k-CAS (in the common uncontended case). Furthermore, our construction has the desirable property that reads do not perform any writes to shared memory (unless they encounter an ongoing MCAS operation). This is to be contrasted with existing MCAS constructions (in which read operations do not write) that use at least $3k+1$ CASes per k-CAS. Furthermore, we extend our MCAS construction to work with persistent memory (PM). The extension does not change the number of CASes and requires only 2 persistence fences per call (in the common uncontended case), comparing favorably to the prior work that employs $5k+1$ CASes and $2k+1$ fences [58].

Most previous MCAS constructions follow a multi-phase approach to perform a k-CAS operation *op*. In the first (*locking*) phase, *op* "locks" its designated memory locations one by one by replacing the current value in those locations with a pointer to a *descriptor* object. This descriptor contains all the information necessary to complete *op* by the invoking thread or (potentially) by a helper thread. In the second (*status-change*) phase, *op* changes a status flag in the descriptor to indicate successful (or unsuccessful) completion. In the third (*unlocking*) phase, *op* "unlocks" those designated memory locations, replacing pointers to its descriptor with new or old values, depending on whether *op* has succeeded or failed.

In order to obtain lower complexity, our algorithm makes two crucial observations concerning this unlocking phase. First, this phase can be deferred off the critical path with no impact on correctness. In our algorithm, once an MCAS operation completes, its descriptor is left in place until a later time. The unlocking is performed later, either by another MCAS operation locking the same memory location (and thus effectively eliminating the cost of unlocking for *op*) or during the memory reclamation of operation descriptors. (We describe a delayed memory reclamation scheme that employs epochs and amortizes the cost of reclamation across multiple operations.)

Our second, and perhaps more surprising, observation is that deferring the unlocking phase allows the *locking* phase to be implemented more efficiently. In order to avoid the ABA problem, many existing algorithms require extra complexity in the locking phase. For instance, the well-known Harris et al. [33] algorithm uses the atomic *restricted double-compare single-swap* (*RDCSS*) primitive (that requires at least 2 CASes per call) to conditionally lock a word, provided that the current operation was not completed by a helping thread. Naively performing the locking phase using CAS instead of RDCSS would make the Harris et al. algorithm prone to the ABA problem (we provide an example in the full version of our paper [27]). However, in our algorithm, we get ABA prevention "for free" by using a memory reclamation mechanism to perform the unlocking phase, because such mechanisms already need to protect against ABA in order to reclaim memory safely.

Deferring the unlocking phase allows us to come up with an elegant and, arguably, simple MCAS construction. Prior work shows, however, that the correctness of an MCAS construction should not be taken for granted: for instance, Feldman et al. [22] and Cepeda et al. [14] describe correctness pitfalls in MCAS implementations. In this paper, we carefully prove the correctness of our construction. We also evaluate our construction empirically by comparing to a state-of-the-art MCAS implementation and showing superior performance through a variety of benchmarks (including a production quality B+-Tree [6]) in most considered scenarios.

We note that the delayed unlocking/cleanup introduces a trade-off between higher MCAS performance (due to fewer CASes per MCAS, which also leads to less slow-down due to less helping) and lower read performance (because of the extra level of indirection reads have to traverse when encountering a descriptor left in place after a completed MCAS). One may argue that it also increases the amount of memory consumed by the MCAS algorithm. Regarding the former, our evaluation shows that the benefits of the lower complexity overcome the drawbacks of indirection in all workloads that experience MCAS contention. Furthermore, we propose a simple optimization to mitigate the impact of indirection in reads. As for the latter, we note that much like any lock-free algorithm, the memory consumption of our construction can be tuned by performing memory reclamation more (or less) often.

The rest of the paper is organized as follows. In Section 2 we describe our model. In Section 3 we present our impossibility result. Sections 4 and 5 detail our MCAS algorithms for volatile and persistent memory. Section 6 elaborates our lazy memory reclamation scheme. Section 7 presents the results of our experimental evaluation. We review related work in Section 8 and conclude in Section 9. Some content has been moved to the optional appendices: Appendix A.1 provides the ABA example for the naive simplification of the Harris et al. algorithm; Appendix A.2 proves the correctness of our volatile algorithm; Appendix A.3 presents the persistent version of our algorithm; Appendices A.4 through Appendix A.6 provide additional details regarding our memory management scheme for volatile and persistent memory and regarding an optimization to improve read performance; Appendix A.7 contains additional performance graphs and Appendix A.8 gives additional details regarding related work on non-blocking MCAS.

2 System Model

2.1 Volatile Memory

We assume a standard model of asynchronous shared memory [37], with basic atomic *read*, *write* and *compare-and-swap* (CAS) operations. The latter receives three arguments—an address, an expected value and a new value; it reads the value stored in the given address and if it is equal to the expected value, atomically stores the new value in the given address, returning the indication of success or failure.

Using those atomic operations, we implement an atomic MCAS operation with the following semantics. The MCAS operation receives an array of tuples, where each tuple contains an address, an expected value and a new value. For ease of presentation, we assume the size of the array is a known constant N. (In practice, the size of the array can be dynamic, and different for every MCAS operation.) The MCAS operation reads values stored in the given addresses, and if they all are equal to respective expected values, atomically writes new values to the corresponding address and returns an indication of success. Otherwise, if at least one read value is different from an expected one, the MCAS operation returns an indication of failure. We also provide a custom implementation of a read operation from a memory location that can be a target of an MCAS operation (which, in the most general case, can be any shared memory location).

Our MCAS implementation is *linearizable* [37]. This means, informally, that each (read or MCAS) operation appears to take effect instantaneously at some point in time in the interval during which the operation executes. In terms of progress, our MCAS implementation is *non-blocking*. That is, a lack of progress of any thread (e.g., due to the suspension or failure of that thread) does not prevent other threads from applying their operations. Furthermore, the MCAS implementation guarantees *lock-freedom*. That is, given a set of threads applying operations, it guarantees that, eventually, at least one of those threads will complete its operation.

Similar to many non-blocking algorithms, our design makes use of operation descriptors, which store information on existing MCAS operations, including the status of the operation and the array of tuples with addresses and values. We assume each word in the shared memory can contain either a regular value or a pointer to such a descriptor. A similar assumption has been made in prior work on MCAS [22, 33, 56, 58]. In practice, a single (e.g., least significant) bit can be used to distinguish between the two.

Initialization of the descriptor is done before invocation of the MCAS operation. We assume that all the addresses in the descriptor are sorted in a monotonic total order. This assumption is crucial for the liveness property of our algorithm, buy can be easily lifted by explicitly sorting the array of tuples by corresponding addresses before an MCAS operation is executed.

2.2 Persistent Memory

We extend the model in Section 2.1 with standard assumptions about PM [15, 17, 24, 40]. We assume the system is equipped with persistent shared memory that can be accessed through the same set of atomic primitives (read, write and CAS). The system may also be equipped with DRAM to be used as transient storage. As in previous work [40], we assume that the overall system can crash at any time and possibly recover later. On such a full-system crash, we assume that the contents of persistent memory—but not those of processor caches, registers or volatile memory—are preserved. Moreover, threads that are active at the time of the crash are assumed to be lost forever and replaced by new threads in case of recovery. After a full-system crash but before the system recovers and resumes normal execution, we assume a *recovery* routine may be executed, in order to bring persistent memory-resident objects to a consistent state. The recovery routine can be executed in a single thread, and thus it does not have to be thread-safe. Another full-system crash, however, may occur during the recovery routine.

As is standard practice [15, 17, 58], we assume that a-priori there is no guarantee on when and in what order cache lines are written back to persistent memory. We assume the existence of two primitives to enforce such write backs. The first primitive is `PERSISTENT_FLUSH(addr)`, which takes as argument a memory location and asynchronously writes the contents of that location to persistent memory. Multiple invocations of this primitive are not ordered with respect to each other and thus several flushes can proceed in parallel. Concrete examples of this primitive are `clflushopt` and `clwb` [38]. The second primitive is `PERSISTENT_FENCE()`, which

stalls the CPU until any pending flushes are committed to persistent memory. A concrete example of this primitive is `sfence` [38]. LOCK-prefixed instructions such as CAS also act as persistent fences [38]. Since persistent flushes do not stall the CPU, whereas persistent fences do, the cost of writing to persistent memory is dominated by the latter instructions and we consider the cost of the former to be negligible.

Regarding initialization, we assume descriptor contents are made persistent before invocation of MCAS.

The safety criterion we use when working with persistent memory is durable linearizability [40]. Informally, an implementation of an object is durably linearizable if it is linearizable and has the following additional properties in case of a full-system crash and recovery: (1) all operations that completed before the crash are reflected in the post-recovery state and (2) if some operation *op* that was ongoing at the time of the crash is reflected in the post-recovery state, then so are all the operations on which *op* depends (i.e., operations whose effects *op* observed and thus need to be linearized before *op*).

3 Impossibility

In this section we show that any lock-free disjoint-access-parallel (DAP) implementation of MCAS requires at least one CAS per modified word. Consider a call to k-CAS($addr_1$,...,$addr_k$, [old and new values]). We call $addr_1$,...,$addr_k$ the *set of targets* of the call. We also define the *range* of the call in an execution E to be the set of locations on which CAS (single-word CAS) is performed, successfully or not, during the call in E. Intuitively, we say that an MCAS implementation is *DAP* if non-conflicting calls to k-CAS do not access the same memory locations; for the formal definition, see [39].

▶ **Definition 1** (Star Configuration). *We say that a set $\{c_0,...,c_\ell\}$ of calls to k-CAS are in a* star configuration *if (1) the sets of targets of c_0 and c_i are non-disjoint for all $i \in \{1,...,\ell\}$, and (2) the sets of targets of c_i and c_j are disjoint for all $i \neq j \in \{1,...,\ell\}$.*

An example of a star configuration for $\ell = k$ is the following set of calls $\mathcal{C} = \{c_0,...,c_k\}$, where we omit old and new values for ease of notation and we assume that addresses $a_i^{(j)}$ are all distinct:

- c_0: k-CAS($a_1^{(0)}$,...,$a_k^{(0)}$)
- c_1: k-CAS($a_1^{(0)}$,$a_2^{(1)}$,...,$a_k^{(1)}$). Call c_1's set of targets intersects that of c_0 in $a_1^{(0)}$.
- c_i, $1 \leq i \leq k$: k-CAS($a_1^{(i)}$,...,$a_i^{(0)}$,...,$a_k^{(i)}$). Call c_i's set of targets intersects that of c_0 in $a_i^{(0)}$ and is disjoint from the set of targets of c_j for all $j \neq i, j \neq 0$.

In this section, we assume without loss of generality that all calls in $\mathcal{C}$ have the correct old values for their target addresses and that each new value is distinct from its respective old value. Under these assumptions, in every execution it must be that either c_0 succeeds and all c_1,...,c_k fail, or that c_0 fails and all c_1,...,c_k succeed.

We say that a state S of an implementation $\mathcal{A}$ is c_0-valent with respect to (*wrt*) some subset $C \subseteq \mathcal{C}$ if, for any call $c_i \in C$, in any execution starting from S in which only c_0 and c_i take steps, c_0 succeeds. Similarly, we say that a state S is C-valent *wrt* c_0 if, for any call $c_i \in C$, in any execution starting from S in which only c_0 and c_i take steps, c_0 fails. We say that a state is univalent *wrt* c_0 and C if it is c_0-valent or C-valent; otherwise it is bivalent *wrt* c_0 and C. A state is critical *wrt* c_0 and C when (1) it is bivalent *wrt* c_0 and C and (2) if any process in $\{c_0\} \cup C$ takes a step, the state becomes univalent *wrt* c_0 and C.

Note that the initial state of $\mathcal{A}$ must be bivalent *wrt* c_0 and any non-empty subset of $\mathcal{S}$.

▶ **Lemma 2.** *Consider a lock-free implementation $\mathcal{A}$ of k-CAS and let $\mathcal{C}=\{c_0,...,c_\ell\}$ be a star configuration of calls to k-CAS. Then there exists an execution E of $\mathcal{A}$ such that, for all $i\geq 1$, the ranges of c_0 and c_i in E are non-disjoint.*

Proof. We follow a bivalency proof structure. We construct an execution in which process p_i performs call c_i, $i\geq 0$. For ease of notation, we say that "call c_i takes a step" to mean "process p_i takes a step in its execution of c_i".

The execution proceeds in stages. In the first stage, as long as some call in $\mathcal{C}$ can take a step without making the state univalent *wrt* c_0 and any non-empty subset of $\mathcal{C}$, let that call take a step. If the execution runs forever, the implementation is not lock-free. Otherwise, the execution enters a state S where no such step is possible, which must be a critical state *wrt* c_0 and some subset $C_1\subseteq\mathcal{C}\setminus\{c_0\}$. We choose C_1 to be maximal, i.e., state S is not critical *wrt* c_0 and any subset of $\mathcal{C}\setminus C_1$ (otherwise, add that subset to C_1).

We prove in Lemma 3 below that c_0 and all calls in C_1 are about to perform CAS on some common location l_1. We let c_0 perform that CAS step, bringing the protocol to state S'. By our choice of C_1 as maximal, S' must be bivalent *wrt* c_0 and any subset of $\mathcal{C}\setminus C_1$. The execution now enters the second stage, in which we let calls in $\mathcal{C}\setminus C_1$ take steps until they reach a critical state *wrt* c_0 and some subset $C_2\subseteq\mathcal{C}\setminus C_1$. By induction, we can show that eventually c_0 will have reached critical points *wrt* all calls in $\mathcal{C}$. At the end of the execution, we resume each process in $\mathcal{C}\setminus c_0$ for one step; they were each about to perform a CAS step on some location on which c_0 has already performed a CAS step. Thus, in this execution, all calls in $\mathcal{C}\setminus c_0$ have performed a CAS on a common location with c_0. ◀

▶ **Lemma 3.** *Consider a lock-free implementation $\mathcal{A}$ of k-CAS and let $\mathcal{C}=\{c_0,...,c_k\}$ be a star configuration of calls to k-CAS. If S is a critical state of $\mathcal{A}$ wrt c_0 and some subset $C\subseteq\mathcal{C}$, then in S, c_0 and all calls in C are about to perform a* CAS *step on a common location l.*

Proof. From S, we consider the next steps of c_0 and any $c_i\in C$:

Case 1 One of the calls is about to read; assume *wlog* it is c_0. Consider two possible scenarios. First scenario: c_i moves first and runs solo until it returns (c_i must succeed because c_i took the first step). Second scenario: c_0 moves first and reads, then c_i runs solo until it returns (c_i must fail because c_0 took the first step). But the two scenarios are indistinguishable to c_i, thus c_i must either succeed in both or fail in both, a contradiction.

Case 2 Both calls are about to write. In this case, they must be about to write to the same register r, otherwise their writes commute. First scenario: c_0 writes r, then c_i writes r, then c_i runs solo until it returns (c_i must fail since c_0 took the first step). Second scenario: c_i writes r and then runs solo until it returns (c_i must succeed since c_i took the first step). But the two scenarios are indistinguishable to c_i, since its write to r obliterated any potential write by c_0 to r, so c_i must either succeed in both scenarios or fail in both; a contradiction.

Case 3 c_0 is about to CAS and c_i is about to write (or vice-versa). In this case, their operations must be to the same memory location r (otherwise they commute). First scenario: c_0 CASes r, then c_i writes to r and then runs solo until c_i returns (c_i must fail since c_0 took the first step). Second scenario: c_i writes to r and then runs solo until it returns (c_i must succeed since c_i took the first step). But the two scenarios are indistinguishable to c_i, since its write to r obliterated any preceding CAS by c_0 to r; thus c_i must either succeed in both scenarios or fail in both; a contradiction.

Case 4 Both calls are about to CAS. In this case, they must be about to CAS the same location, otherwise their CASes commute. ◀

▶ **Theorem 4.** *Consider a lock-free disjoint-access-parallel implementation $\mathcal{A}$ of k-CAS in a system with $n > k$ processes. Then there exists some execution E of $\mathcal{A}$ such that in E some call to k-CAS performs CAS on at least k locations.*

Proof. We prove the theorem by contradiction. We first assume that calls to k-CAS perform CAS on *exactly* $k-1$ locations and derive a contradiction; we later show how assuming that k-CAS performs CAS on *at most* $k-1$ locations also leads to a contradiction.

We construct an execution E in which two concurrent but non-contending k-CAS calls (i.e., two k-CAS calls with disjoint sets of targets) perform CAS on the same location, thus contradicting the disjoint-access-parallelism (DAP) property and proving the theorem.

Let $c_0,...,c_k$ be $k+1$ calls to k-CAS in a star configuration. By Lemma 2, there exists an execution E of $\mathcal{A}$ such that, for all $i \geq 1$, the ranges of c_0 and c_i in E are non-disjoint.

Let $l_1,...,l_{k-1}$ be the range of c_0. By Lemma 2, in E the range of c_1 must intersect that of c_0 in at least one location; assume *wlog* it is l_1. Furthermore, the range of c_2 must also intersect that of c_0 in at least one location; moreover, due to the DAP property, the intersection must contain some location other than l_1, since c_1 and c_2 have disjoint sets of targets. By induction, we can show that the range of each call $c_i, i \in \{1,2,...,k-1\}$ intersects the range of c_0 in l_i. However, the range of c_k must also intersect the range of c_0 in some location other than $l_1,...,l_{k-1}$, due to the DAP property. We have reached a contradiction.

If we now assume that calls to k-CAS perform CAS on $k-1$ *or fewer* locations, then we also reach a similar contradiction as above. In fact, if some call c_i performs CAS on strictly fewer than $k-1$ locations, this may cause the contradiction to occur before call c_k, as c_i now has fewer locations to choose from in order intersect with the range of c_0 in some location that is not in the ranges of $c_1,...,c_{i-1}$. ◀

4 Volatile MCAS with $k+1$ CAS

In this section we describe our MCAS construction for volatile memory. Our algorithm uses $k+1$ CAS operations in the common uncontended case, and does not involve cleaning up after completed MCAS operations. In Section 6 we describe a memory management scheme that can be used to clean up after completed MCAS operations as well as for reclaiming or reusing operation descriptors employed by the algorithm.

4.1 High-level Description

As is standard practice [30, 33, 56], our MCAS construction supports two operations: MCAS and `read`. Similarly to most MCAS algorithms [30, 33, 56], the MCAS operation uses operation descriptors that contain a set of addresses (the *target* addresses or words), and *old* and *new* values for each target address. In addition, each operation descriptor contains a *status* word indicating the status of the corresponding MCAS operation.

The MCAS operation proceeds in two stages. In the first stage, we attempt to install a pointer to the operator descriptor in each memory word targeted by the MCAS operation. If we succeed to install the pointer, we say that the target address is *owned* (or *locked*) by the descriptor. The first stage ends when all target addresses are owned by the descriptor, or if we find a target address with a value different from the expected one. In the second stage, we *finalize* the MCAS operation by atomically changing its status to indicate its success or failure, depending on whether the first stage was successful (i.e., all target addresses have been locked). The `read` operation returns the current value at an address, either by reading it directly from the target address or by reading the appropriate value from a descriptor of a completed MCAS operation installed in that

Listing 1 Data structures used by our algorithm

```
struct WordDescriptor {
    void* address;
    uintptr_t old;
    uintptr_t new;
    MCASDescriptor* parent; };

enum StatusType { ACTIVE, SUCCESSFUL, FAILED };

struct MCASDescriptor {
    StatusType status;
    size_t N;
    WordDescriptor words[N]; };
```

Listing 2 The `readInternal` auxiliary function, used by our algorithm.

```
1  readInternal(void* addr, MCASDescriptor *self) {
2  retry_read:
3    val = *addr;
4    if (!isDescriptor(val)) then return <val,val>;
5    else { // found a descriptor
6      MCASDescriptor* parent = val->parent;
7      if (parent != self && parent->status == ACTIVE) {
8        MCAS(parent);
9        goto retry_read;
10     } else {
11       return parent->status == SUCCESSFUL ?
12           <val,val->new> : <val,val->old>; }  }  }
```

address. If either MCAS or `read` encounter another MCAS in progress (e.g., when they attempt to read the current value in the target address), they first help that MCAS operation to complete.

4.2 Technical Details

Structures and Terminology. We describe the structures used by our algorithm and explain the terminology. Pseudocode for the structures is shown in Listing 1. An `MCASDescriptor` describes an MCAS operation. It contains a status field, which can be `ACTIVE`, `SUCCESSFUL` or `FAILED`, the number `N` of words targeted by the MCAS and an array of `WordDescriptor`s for those words. These `WordDescriptor`s are the *children* of the `MCASDescriptor`, who is their *parent*. We say that an `MCASDescriptor` (and the MCAS it describes) is *active* if its status is `ACTIVE` and *finalized* otherwise.

The `WordDescriptor` contains information related to a given word as target of an MCAS operation: the word's address in memory, its expected value and the new intended value. The `WordDescriptor` also contains a pointer to the descriptor of its parent MCAS operation. As described later, the pointer is used as an optimization for fast lookup of the status field in the `MCASDescriptor`, and can be eliminated.

Algorithm. Both MCAS and `read` operations rely on the auxiliary `readInternal` function shown in Listing 2. The `readInternal` function takes an address `addr` and an `MCASDescriptor` `self` (called the *current descriptor*) and returns a tuple. The tuple contains two values (which might be identical), and, intuitively, represent the contents in the given (target) address and the actual value the former represents. More specifically, `readInternal` reads the content of the given `addr` (Line 3). If `addr` does not point to a descriptor (this is determined by the `isDescriptor` function; see below), the returned tuple contains two copies of the contents of `addr` (Line 4). If `addr` points to an active `WordDescriptor` whose parent is not the same

Listing 3 Our main algorithm. Commands in *italic* are related to memory reclamation (discussed in a later section).

```
13 read(void* address) {
14   epochStart();
15   <content, value> = readInternal(address, NULL);
16   epochEnd();
17   return value; }
18
19 MCAS(MCASDescriptor* desc) {
20   epochStart();
21   success = true;
22   for wordDesc in desc->words {
23 retry_word:
24     <content, value> = readInternal(wordDesc.address, desc);
25     // if this word already points to the right place, move on
26     if (content == &wordDesc) continue;
27     // if the expected value is different, the MCAS fails
28     if (value != wordDesc.old) { success = false; break;  }
29     if (desc->status != ACTIVE) break;
30     // try to install the pointer to my descriptor; if failed,  retry
31     if (!CAS(wordDesc.address, content, &wordDesc)) goto retry_word; }
32   if (CAS(&desc.status, ACTIVE, success ? SUCCESSFUL : FAILED)){
33     // if I finalized this descriptor, mark it for reclamation
34     retireForCleanup(desc); }
35   returnValue = (desc.status == SUCCESSFUL);
36   epochEnd();
37   return returnValue; }
```

as `self`, then `readInternal` helps the other (MCAS) operation to complete (Line 8) and then restarts (Line 9). Therefore, the role of the `self` pointer is to avoid an (MCAS) operation to help itself recursively. If `addr` points to a finalized descriptor, the tuple returned by `readInternal` contains the pointer to the descriptor and the final value, corresponding to the status of the descriptor (Line 12). Finally, if `addr` points to a descriptor whose parent is equal to `self`, then `readInternal` returns the pointer to that descriptor (Line 12, a value is also returned in the tuple in this case, but is disregarded; see below).

Listing 3 provides the pseudo-code for the `read` and MCAS operations. The pseudo-code includes extensions relevant to memory management (in *italics*), whose discussion is deferred to Section 6.

The `read` operation is simply a call to `readInternal` with a `self` equal to `null` as the current operation descriptor (Line 15).

The MCAS operation takes as argument an `MCASDescriptor` and returns a boolean indicating success or failure. As mentioned above, the operation proceeds in two stages. In the first stage, MCAS attempts to take ownership of (or *acquire*) each target word (Lines 22–31). To this end, for each `WordDescriptor` w in its `words` array, we start by calling `readInternal` on w's target address `addr` (Line 24; as described above, this handles any helping required in case another active operation owns `addr`). If `addr` is already owned by the current MCAS, we move on to the next word (Line 26). Otherwise, if the current value at `addr` does not match the expected value of w, the MCAS cannot succeed and thus we can skip the next `WordDescriptor`s and go to the second stage (Line 28). If the values do match, we re-check if the operation is still active (line 29); otherwise we go to the second stage—this prevents a memory location from being re-acquired by the current operation *op* in case *op* was already finalized by a helping thread. Finally, we attempt to take ownership of `addr` through a CAS (Line 31). Note that the failure of this CAS might mean that another thread has concurrently helped this MCAS to lock the target word. Therefore, we simply retry taking ownership on

this target word, rather than failing the MCAS operation (Line 31).

In the second stage (Lines 32–34), MCAS finalizes the descriptor by atomically changing its status from `ACTIVE` to `SUCCESSFUL` (if all word acquisitions were successful in stage one) or to `FAILED` (otherwise).

Our pseudocode assumes the existence of the `isDescriptor` function, which takes a value and returns `true` if and only if the value is a pointer to a `WordDescriptor`. This function can be implemented, for instance, by designating a low-order *mark bit* in a word to indicate whether it contains a pointer to a descriptor or not [33, 58]. Whenever we make an address point to a descriptor (e.g., Line 31) or convert the contents of a word into a pointer to descriptor (e.g., Line 6), we also set or unset the mark bit, respectively. In the interest of clarity, we do not show the implementation of `isDescriptor` or the code for marking/unmarking pointers.

We give a proof of correctness for our algorithm in the Appendix.

5 Persistent MCAS with $k+1$ CAS and 2 Persistent Fences

We discuss the modifications required to make our volatile MCAS algorithm work with persistent memory. The extra instructions are shown underlined in Listings 4 and 5 in the Appendix.

In the `MCAS` function (Listing 4), after all target locations have been successfully acquired, we add one persistent flush per target word and one persistent fence overall. The persistent fence ensures that all target locations persistently point to their respective `WordDescriptors` before attempting to modify the status.

When finalizing the status, we mark the status with a special `DirtyFlag`. This flag indicates that the status is not yet persistent. We then perform a persistent flush and fence after the status has been finalized. This ensures that the finalized status of the descriptor is persistent before returning from the MCAS. Finally, we unset the `DirtyFlag` with a simple store (line 28); this store cannot create a race with the CAS that finalizes the status (line 23) because that CAS must fail (the status must be already finalized if some thread is already attempting to unset the dirty flag (line 28)).

We also modify the `readInternal` function (Listing 5) such that, when an operation *op* encounters another operation *op*′ whose status is finalized but still has the `DirtyFlag` set, *op* helps *op*′ persist its status and unsets the `DirtyFlag` on *op*′ status.

Our modifications enforce the following invariants. First, at the time when a descriptor becomes finalized, its acquisitions of target locations are persistent. Second, at the time when an MCAS operation returns, its finalized status is persistent. Third, when a read or MCAS operation *op* returns, all operations on which *op* depends are finalized and their statuses are persistent. With these invariants, we can argue that our persistent MCAS is correct. By correctness we refer to lock-freedom (liveness) and durable linearizability (safety). Lock-freedom is clearly preserved by our additions, thus we focus on durable linearizability. We examine the point in time when a full-system crash may occur during the execution of an MCAS operation *op*. There are two possibilities to consider:

1. If the crash occurs before *op*'s status was finalized and made persistent, then we know that no operation *op*′ which observed the effects of *op* could have returned before the crash; otherwise, *op*′ would have helped *op* and persisted its status. In this case, neither *op* nor any such *op*′ will be linearized before the crash; during recovery, their effects will be rolled back by reverting any acquired locations to their old values.
2. If the crash occurs after *op*'s status was finalized and made persistent, then *op* is linearized before the crash. During recovery, any locations still acquired by *op* will be detached and given either their new or old values (depending on *op*'s success or failure status), as specified

in *op*'s descriptor.

In sum, the recovery procedure of our algorithm is as follows. The recovery goes through each operation descriptor D. If D's status is not finalized, then we roll D back by going through each target location ℓ of D; if ℓ is acquired by D (i.e., points to D), then we write into ℓ its old value, as specified in D. If D's status is finalized, then we detach D and install final values; we go through each target location ℓ of D; if ℓ is acquired by D and D was successful (resp. failed), then we write into ℓ the new (resp. old) value as specified in D.

6 Memory Management

The MCAS algorithm has been presented so far under the assumption that no memory is ever reclaimed. For practical considerations, however, one should to be able to reclaim and/or reuse MCAS operation descriptors. While efficient memory management of concurrent data structures remains an active area of research (see, e.g., [3, 12, 20, 54, 59]), here we describe one possible mechanism suitable for an MCAS implementation. We briefly outline the mechanism here and defer its full description, as well as optimizations for persistent memory and efficient reads, to Appendices A.4 through A.6.

We note that the life cycle of an operation descriptor comprises several phases. Once its status is no longer `ACTIVE`, the (finalized) descriptor cannot be recycled just yet as certain memory locations can point to it. Therefore, we need first to *detach* such a descriptor by replacing the pointers to the descriptor (using CAS) with actual values (respective to whether the corresponding MCAS has succeeded or failed) in affected memory locations. Only after that, a detached descriptor can be recycled, provided no concurrently running thread holds a reference to it. Note that CASes in the detachment phase are necessary only for those affected memory locations that still point to the to-be-detached descriptor, which, as our evaluation shows, is rare in practice.

Our scheme keeps track of two categories of descriptors: (1) those that have been finalized but not yet detached and (2) those that have been detached but to which other threads might still hold references. Similar to RCU approaches [44, 45], we use thread-local epoch counters to track threads' progress and infer when a descriptor can be moved from category (1) to category (2), and when a descriptor from category (2) can be reclaimed.

7 Evaluation

7.1 Experimental Setup

We evaluate our algorithm on a 2-socket Intel Xeon machine with two E5-2630 v4 processors operating at 3.1 GHz. Each processor has 10 cores, each core has 2 hardware threads (40 hardware threads total). Each experimental run lasts 5 seconds; shown values are the average of 5 runs. We base our evaluation on the framework available from the authors of PMwCAS [53, 58].

The baseline of our evaluation is the volatile version of PMwCAS [53, 58], a state-of-the-art implementation of the Harris et al. [33] algorithm. Like the Harris et al. algorithm, volatile PMwCAS requires $3k+1$ CASes per k-CAS. We use PMwCAS as our baseline since (1) it has recent, openly available and well-maintained code and (2) it is to our knowledge the only other MCAS algorithm in which readers do not write to shared memory in the common uncontended case.

PMwCAS implements an optimization of the Harris et al. algorithm: it marks pointers with a special *RDCSS* flag instead of allocating a distinct RDCSS descriptor. However, we found that this optimization made the PMwCAS algorithm incorrect, due to an ABA vulnerability. In our evaluation, we fixed the PMwCAS implementation to allocate and manually manage RDCSS descriptors.

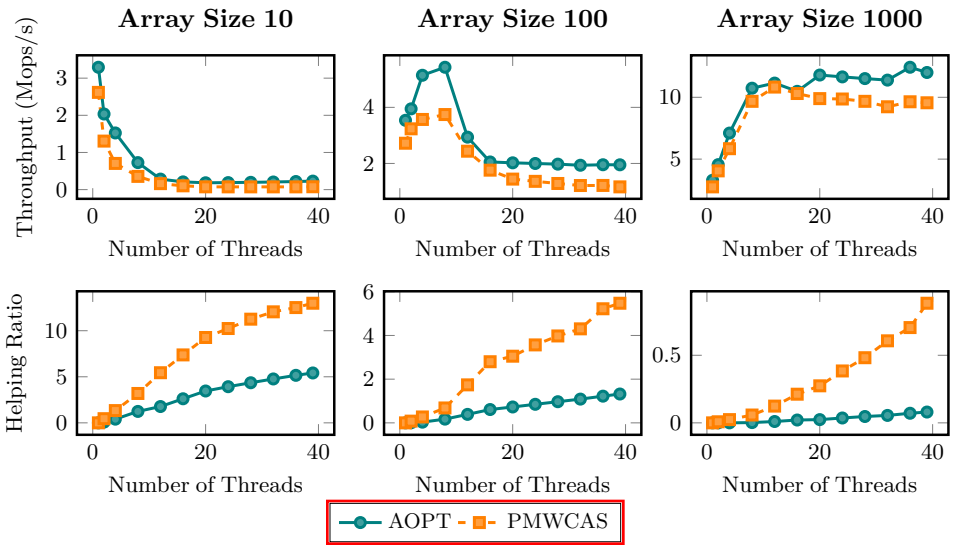


Figure 1 Array benchmark. Top row shows throughput (higher is better), bottom row shows helping ratio (lower is better). Each column corresponds to a different array size (10, 100 and 1000, respectively).

Our evaluation uses three benchmarks: an *array benchmark* in which threads perform MCAS-based read-modify-write operations at random locations in an array, a *doubly-linked list benchmark*, in which threads perform MCAS-based operations on a list implementing an ordered set, and a *B+tree benchmark* in which threads perform MCAS-based operations on a B+-tree. The first two benchmarks are based on the implementation available in [53], and the third is based on PiBench [52] and BzTree [6, 13]. We note, however, that we modified the benchmark in [53] so all threads operate on the same key range (rather than having each thread using a unique set of keys), so we could induce contention by controlling the size of the key range.

In each experiment, we vary the number of threads from 1 to 39 (we reserve one hardware thread for the main thread). Threads are assigned according to the default settings in the evaluation frameworks used [13, 52, 53]. In the array and list benchmarks, threads are assigned in the following way: we first populate the first hardware thread of each core on the first socket, then on the second socket, then we populate the second hardware thread on each core on the first socket, and finally the second hardware thread on each core on the second socket. The B+-tree benchmark uses OpenMP [49], which dictates thread assignment; it also employs a scalable memory allocator [1].

7.2 Array Benchmark

The benchmark consists of each thread performing the following in a tight loop: reading k locations at random from the array ($k=4$ in our experiments), computing a new value for each location, and attempting to install the new values using an MCAS.

In this benchmark we measure two quantities. The first is throughput: the number of read-modify-write operations completed successfully per time unit. The second metric is the *helping ratio*. We measure the helping ratio by dividing the number of *ongoing* MCAS operations encountered (and helped) during read or MCAS operations by the total number of MCAS operations. A higher helping ratio thus means more operations are slowed down due to the need to help other, incomplete MCAS operations.

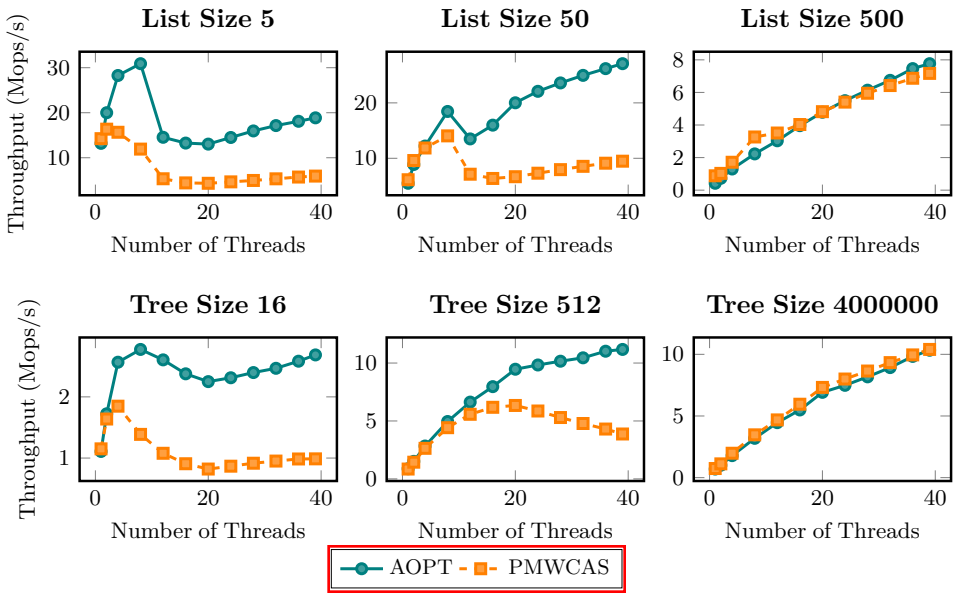


Figure 2 Top row: Doubly-linked list benchmark (80% reads) with different initial list sizes (5, 50 and 500 elements). Bottom row: B+-tree benchmark (80% reads) with different initial tree sizes (16, 512 and 4000000 elements).

We run the benchmark with three array sizes (10, 100, and 1000) in order to capture different contention levels. The results of this benchmark are shown in Figure 1 (our algorithm is denoted *AOPT* in all figures in this section).

The top row of Figure 1 shows that our algorithm outperforms PMwCAS at every contention level and at every thread count, including in single-threaded mode. This can be explained by two related factors. First, our algorithm has a lower CAS complexity ($k+1$ CASes per k-CAS for our algorithm compared to $3k+1$ for PMwCAS). Second, as a consequence of its lower complexity, in our algorithm there is a shorter "window" for each MCAS operation to interfere with other operations by forcing them to help.

To illustrate the second factor above, we examine the helping ratios of the two algorithms (bottom row of Figure 1). We observe that the helping ratio of our algorithm is considerably lower than that of PMwCAS. This means that, on average, each operation helps (and is slowed down by) fewer MCAS operations in our algorithm than in PMwCAS.

In order to quantify the impact of descriptor cleanup on performance in our algorithm, we also measure the *detaching ratio*: the number of CASes performed in order to detach (in the sense of Section 6) finalized MCAS descriptors, divided by the total number of completed MCAS operations. We find the detaching ratio to be less than 0.001 for every thread count and array size. This is because finalized MCAS descriptors are constantly being replaced by ongoing MCAS operations, and thus recycling these detached descriptors requires no CASes. We conclude that the vast majority of our MCAS operations do not incur any cleanup CASes.

7.3 Doubly-linked List Benchmark

In this benchmark we operate on a shared ordered set object implemented from a doubly-linked list. The list supports search and update (insert and delete) operations. Insertions are done using 2-CAS and deletions are done using 3-CAS. We initialize the list by inserting a predefined

(configurable) number of nodes. During the benchmark, each thread selects an operation type (search, insert or delete) at random, according to a configurable distribution; the thread also selects a value at random; it then performs the selected operation with the selected value.

We perform this benchmark with three initial list sizes (5, 50 and 500 elements). The operation distribution is: 80% reads, 20% updates (in all our experiments, updates are evenly distributed among insertions and deletions). As is standard practice, the initial size of the list is half of the key range. Results are shown in the top row of Figure 2. We also ran experiments with 50%, 98%, and 100% reads; to improve readability, performance graphs for these less representative cases are deferred to Appendix A.7.

Our algorithm outperforms PMwCAS for list sizes 5 and 50 by 2.6× and 2.2× on average, respectively. This shows that under high and moderate contention, our algorithm's faster MCAS operations (due to the double effect of lower complexity and lower helping ratio) compensate for its slower read operations (due to the extra level of indirection), even in read-heavy workloads. In the low contention case (list size 500), PMwCAS outperforms our algorithm at low thread counts and is outperformed at high thread counts. On average, PMwCAS outperforms our algorithm by 1.2×. Under low contention, operations have a low probability to conflict on the same element and thus the lower read complexity of PMwCAS has a stronger impact on performance than the lower MCAS complexity of our algorithm.

7.4 B+-tree Benchmark

In this benchmark we operate on a B+-tree which supports search and update (insert and delete) operations. Insertions and deletions use k-CAS, where k may vary, e.g., depending on whether the operation led to nodes being split or merged.

Similar to the previous benchmark, we initialize the B+-tree with a configurable number of entries; threads then select operations and values at random. We perform the benchmark with 80% reads and three initial tree sizes (16, 512, and 4000000). As for the previous benchmark, performance graphs for the 50%, 98% and 100% reads cases are shown in Appendix A.7. As before, the initial size of the tree is half of the key range. Results are shown in the bottom row of Figure 2.

We observe a similar behavior to the previous benchmark. Our algorithm outperforms PMwCAS under high and medium contention (because it performs fewer CASes and triggers less helping) and is slightly outperformed under low contention (where helping no longer plays a major role).

8 Related Work

Lock- and wait-free implementations of MCAS. Our algorithm shares similarities with previous work [33, 58]: as has become standard practice, it uses operation descriptors and a three-phase design (locking, status-change and unlocking). However, our algorithm introduces key differences with respect to previous work: it defers the unlocking phase and combines it with the reclamation of descriptors, without compromising correctness. This deferment has a triple beneficial effect on complexity: (1) it removes k CASes from the critical path, (2) it allows these CASes to be amortized across several operations, and (3) it removes the onus of ABA-prevention from the locking phase, thus shaving off k further CASes from the latter.

Table 1 summarizes the differences between our algorithm and existing non-blocking MCAS implementations, while the detailed treatment of each of the numerous prior efforts is deferred to Appendix A.8. The results in Table 1 reflect the number of CASes per MCAS operation required for correctness by each algorithm in the common uncontended case. We note that previous MCAS implementations perform descriptor cleanup immediately after applying

Table 1 Comparison of non-blocking MCAS implementations in terms of the number of CAS instructions required, whether readers perform writes to shared memory or expensive atomic instructions, and the number of persistent fences (all per k-word MCAS, in the uncontended case).

	CASes	Readers write	P. fences
Israeli and Rappoport [39]	$3k+2$	Yes	N/A
Anderson and Moir [4]	$3k+2$	Yes	N/A
Moir [47]	$3k+4$	Yes	N/A
Harris et al. [33]	$3k+1$	No	N/A
Ha and Tsigas [29, 30]	$2k+2$	Yes	N/A
Attiya and Hillel [8]	$6k+2$	N/A	N/A
Sundell [56]	$2k+1$	Yes	N/A
Feldman et al. [22]	$3k-1$	Yes	N/A
Wang et al. [58] (volatile)	$3k+1$	No	N/A
Wang et al. [58] (persistent)	$5k+1$	No	$2k+1$
Our algorithm	$k+1$	No	2

MCAS, and it is not clear how to separate cleanup from these algorithms while preserving correctness. If we take the cleanup cost into consideration for our algorithm as well, its theoretical (worst-case) complexity becomes $2k+1$, the same as some of the previous work. As our experiments in Section 7 demonstrate, however, the number of CASes in the cleanup phase is negligible in practice. Furthermore, we highlight the fact that unlike most previous work, including the one that employs $2k+1$ CASes, readers in our case do not write into the shared memory in the common case, even when cleanup is considered.

General techniques. Transactional memory (TM) [36, 55] can be seen as the most general approach to providing atomic access to multiple objects. It allows a block of code to be designated as a transaction and thus executed atomically, with respect to other transactions. Thus, TM is strictly more general than MCAS. This generality comes at a cost: software implementations of transactional memory (STM) have prohibitive performance overheads, whereas hardware support (HTM) is subject to spurious aborts and thus only provides "best-effort" guarantees. Prior work on nonblocking STMs [23, 43] share goals similar to our work; namely reduction of overheads in the critical path. However, these works (i) either employ k extra cleanup CASes [23] on the critical path, incurring precisely the overheads we avoid in our work, or (ii) employ a vastly more complex "stealing" framework to avoid overheads from the critical path [43].

As any concurrent object, MCAS can be implemented using a universal construction [34], but such an implementation is not disjoint-access-parallel and has high overhead.

Prior Work on Persistent MCAS. Pavlovic et al. [50] provide an implementation of MCAS for persistent memory which differs from ours in the progress guarantee (theirs is blocking) and hardware assumpions (theirs uses HTM). In their algorithm, a transaction is used to atomically verify expected values and acquire ownership of all target locations. In case of success, the new values are written non-transactionally. Reads that encounter a location owned by an MCAS operation block until the location is no longer owned.

Wang et al. [6, 58] introduce the first lock-free persistent implementation, based on the algorithm of Harris et al. [33]. The main differences with respect to our algorithm are outlined in Table 1. This algorithm uses a per-word dirty flag to indicate that the word is not yet guaranteed to be written to persistent memory. Operations encountering a set dirty flag will persist the associated word and then unset the flag. This technique avoids unnecessary

persistent flushes, but uses 2 extra CAS instructions per target location in order to manipulate the dirty flag. In total, this algorithm uses $5k+1$ CAS instructions for a k-word MCAS in the uncontended case. Their implementation does not use explicit persistent fences; instead, it relies on the CAS instructions that are already required to unset the dirty flag to also enforce ordering among write backs [38]. Their original algorithm uses $2k+1$ such "CAS-fences", but we believe it can be modified to only require 3 persistent fences.

In our work we use the recent durable linearizability correctness condition [40], which assumes a full-system crash-recovery model, but other models of persistent memory can be explored in this context [2, 9, 16, 28, 51].

9 Conclusion

Atomic multi-word primitives significantly simplify concurrent algorithm design, but existing implementations have high overhead. In this paper, we propose a simple and efficient lock-free algorithm for multi-word compare-and-swap, designed for both volatile and persistent memory. The complimentary lower bound shows that the complexity of our algorithm, as measured in the number of CASes in the uncontended case, is nearly optimal.

References

1 Yehuda Afek, Dave Dice, and Adam Morrison. Cache index-aware memory allocation. In *Proceedings of the International Symposium on Memory Management (ISMM)*, page 55–64. Association for Computing Machinery, 2011.

2 Marcos K Aguilera and Svend Frølund. Strict linearizability and the power of aborting. Technical Report HPL-2003-241, HP Labs, 2003.

3 Dan Alistarh, William M. Leiserson, Alexander Matveev, and Nir Shavit. Forkscan: Conservative Memory Reclamation for Modern Operating Systems. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017*, pages 483–498, 2017.

4 James H. Anderson and Mark Moir. Universal Constructions for Multi-object Operations. In *14th Annual ACM Symposium on Principles of Distributed Computing*, pages 184–193, 1995.

5 James H. Anderson, Srikanth Ramamurthy, and Rohit Jain. Implementing Wait-free Objects on Priority-based Systems. In *16th Annual ACM Symposium on Principles of Distributed Computing*, pages 229–238, 1997.

6 Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. BzTree: A High-Performance Latch-free Range Index for Non-Volatile Memory. In *44th International Conference on Very Large Data Bases*, 2018.

7 ASCYLIB, a concurrent-search data-structure library with over 40 implementations of linked lists, hash tables, skip lists, binary search trees, queues, and stacks. `https://github.com/LPD-EPFL/ASCYLIB`, 2018.

8 Hagit Attiya and Eshcar Hillel. Highly concurrent multi-word synchronization. *Theor. Comput. Sci.*, 412(12-14):1243–1262, 2011.

9 Ryan Berryhill, Wojciech M. Golab, and Mahesh Tripunitara. Robust shared objects for non-volatile main memory. In *19th International Conference on Principles of Distributed Systems, OPODIS 2015*, pages 20:1–20:17, 2015.

10 Anastasia Braginsky and Erez Petrank. A Lock-free B+Tree. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 58–67, 2012.

11 Trevor Brown, Faith Ellen, and Eric Ruppert. Pragmatic Primitives for Non-blocking Data Structures. In *ACM Symposium on Principles of Distributed Computing, PODC '13*, pages 13–22, 2013.

12 Trevor Alexander Brown. Reclaiming Memory for Lock-Free Data Structures: There Has to Be a Better Way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 261–270, 2015.

13 Bztree: a high-performance latch-free range index for non-volatile memory. https://github.com/wangtzh/bztree. 2019.

14 Diego Cepeda, Sakib Chowdhury, Nan Li, Raphael Lopez, Xinzhe Wang, and Wojciech Golab. Toward linearizability testing for multi-word persistent synchronization primitives. In *23rd International Conference on Principles of Distributed Systems, OPODIS 2019*, volume 153, pages 19:1–19:17, 2019.

15 Nachshon Cohen, Rachid Guerraoui, and Igor Zablotchi. The Inherent Cost of Remembering Consistently. In *Proceedings of the 30th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2018*, 2018.

16 Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin C. Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009*, pages 133–146, 2009.

17 Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. Log-Free Concurrent Data Structures. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018*, 2018.

18 Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 33–48, 2013.

19 David Detlefs, Christine H. Flood, Alex Garthwaite, Paul Martin, Nir Shavit, and Guy L. Steele, Jr. Even Better DCAS-Based Concurrent Deques. In *14th International Conference on Distributed Computing*, pages 59–73, 2000.

20 Dave Dice, Maurice Herlihy, and Alex Kogan. Fast non-intrusive memory reclamation for highly-concurrent data structures. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, pages 36–45, 2016.

21 Simon Doherty, David L. Detlefs, Lindsay Groves, Christine H. Flood, Victor Luchangco, Paul A. Martin, Mark Moir, Nir Shavit, and Guy L. Steele, Jr. DCAS is Not a Silver Bullet for Nonblocking Algorithm Design. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 216–224, 2004.

22 Steven D. Feldman, Pierre LaBorde, and Damian Dechev. A Wait-Free Multi-Word Compare-and-Swap Operation. *International Journal of Parallel Programming*, 43(4):572–596, 2015.

23 Keir Fraser. *Practical lock-freedom.* PhD thesis, University of Cambridge, UK, 2004.

24 Michal Friedman, Maurice Herlihy, Virendra J. Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018*, pages 28–40, 2018.

25 Michael Greenwald. Non-blocking synchronization and system design. *ph.d. thesis, stanford university*, 1999.

26 Michael Greenwald. Two-handed emulation: how to build non-blocking implementation of complex data-structures using DCAS. In *21st Annual ACM Symposium on Principles of Distributed Computing*, pages 260–269, 2002.

27 Rachid Guerraoui, Alex Kogan, Virendra J. Marathe, and Igor Zablotchi. Efficient multi-word compare and swap. *ArXiv preprint arXiv:???*, 2020. URL: http://arxiv.org/???, arXiv:???

28 Rachid Guerraoui and Ron R. Levy. Robust Emulations of Shared Memory in a Crash-Recovery Model. In *24th International Conference on Distributed Computing Systems (ICDCS 2004)*, pages 400–407, 2004.

29 Phuong Hoai Ha and Philippas Tsigas. Reactive Multi-Word Synchronization for Multiprocessors. In *12th International Conference on Parallel Architectures and Compilation Techniques (PACT 2003)*, pages 184–193, 2003.

30 Phuong Hoai Ha and Philippas Tsigas. Reactive Multi-word Synchronization for Multiprocessors. *J. Instruction-Level Parallelism*, 6, 2004.

31 Tim Harris, James R. Larus, and Ravi Rajwar. *Transactional Memory: 2nd Edition.* Morgan & Claypool, 2010.

32 Timothy L. Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 388–402, 2003.

33 Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A Practical Multi-word Compare-and-Swap Operation. In *16th International Conference on Distributed Computing*, pages 265–279, 2002.

34 Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

35 Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software Transactional Memory for Dynamic-sized Data Structures. In *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*, pages 92–101, 2003.

36 Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. In *20th Annual International Symposium on Computer Architecture*, pages 289–300, 1993.

37 Maurice Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

38 Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual Combined. `https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf`, 2018.

39 Amos Israeli and Lihu Rappoport. Disjoint-Access-Parallel Implementations of Strong Shared Memory Primitives. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 151–160, 1994.

40 Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *Distributed Computing - 30th International Symposium, DISC 2016*, pages 313–327, 2016.

41 Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. The Bw-Tree: A B-tree for New Hardware Platforms. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, pages 302–313, 2013.

42 Victor Luchangco, Mark Moir, and Nir Shavit. Nonblocking k-compare-single-swap. In *SPAA 2003: Proceedings of the Fifteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 314–323, 2003.

43 Virendra J. Marathe and Mark Moir. Toward high performance nonblocking software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2008, Salt Lake City, UT, USA, February 20-23, 2008*, pages 227–236. ACM, 2008.

44 Paul E. McKenney. Is parallel programming hard, and, if so, what can you do about it?, 2017.

45 Paul E. McKenney and John D. Slingwine. Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *Parallel and Distributed Computing and Systems*, 1998.

46 Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004.

47 Mark Moir. Transparent Support for Wait-Free Transactions. In *11th International Workshop on Distributed Algorithms*, pages 305–319, 1997.

48 Aravind Natarajan and Neeraj Mittal. Fast Concurrent Lock-free Binary Search Trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 317–328, 2014.

49 The OpenMP API specification for parallel programming. `https://www.openmp.org/`, 2019.

50 Matej Pavlovic, Alex Kogan, Virendra J. Marathe, and Tim Harris. Persistent Multi-Word Compare-and-Swap. In *ACM Symposium on Principles of Distributed Computing*, 2018.

51 Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency: Semantics for byte-addressable nonvolatile memory technologies. *IEEE Micro*, 35(3):125–131, 2015.

52 Benchmarking framework for index structures on persistent memory. `https://github.com/wangtzh/pibench`, 2019.

53 Persistent multi-word compare-and-swap (PMwCAS) for NVRAM. `https://github.com/microsoft/pmwcas`, 2019.

54 Manuel Pöter and Jesper Larsson Träff. *Stamp-it*, amortized constant-time memory reclamation in comparison to five other schemes. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018*, pages 413–414, 2018.

55 Nir Shavit and Dan Touitou. Software Transactional Memory. In *14th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

56 Håkan Sundell. Wait-Free Multi-Word Compare-and-Swap Using Greedy Helping and Grabbing. *International Journal of Parallel Programming*, 39(6):694–716, 2011.

57 Shahar Timnat, Maurice Herlihy, and Erez Petrank. A Practical Transactional Memory Interface. In *Euro-Par 2015: Parallel Processing - 21st International Conference on Parallel and Distributed Computing*, pages 387–401, 2015.

58 Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. Easy Lock-Free Indexing in Non-Volatile Memory. In *34th IEEE International Conference on Data Engineering*, 2018.

59 Haosen Wen, Joseph Izraelevitz, Wentao Cai, H. Alan Beadle, and Michael L. Scott. Interval-based memory reclamation. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–13, 2018.

A APPENDIX

A.1 Replacing RDCSS with CAS in Harris et al. algorithm leads to ABA

Consider two memory locations a_1 and a_2 with initial values v_1 and v_2 respectively. Consider two 2-CAS operations op and op' which operate on a_1 and a_2. op has old values v_1 and v_2 and new values v_1' and v_2', respectively; op' has old values v_1' and v_2' and new values v_1 and v_2, respectively. Let D be op's descriptor.

1. op executes solo and performs the CAS to make a_1 point to D, then pauses immediately before the CAS to acquire a_2.
2. op' executes solo: it first helps op complete, changing the values of a_1 and a_2 to v_1' and v_2' respectively; then op' performs its own changes, modifying a_1's and a_2's values back to v_1 and v_2, respectively.
3. op resumes, successfully acquires a_2, performs the status-change CAS on D, then performs unlocking CASes on a_1 and a_2. The CAS on a_1 will fail, and a_1's value will remain v_1. The CAS on a_2 will succeed, changing its value to v_2'.
4. The values of a_1 and a_2 are now incompatible with any linearization of op and op'.

A.2 Correctness of Volatile $k+1$ algorithm

In this section we argue that our MCAS algorithm is linearizable and lock-free. We give preliminary invariants before showing the main results.

▶ **Lemma 5.** *Once an* MCAS *descriptor is finalized, its status never changes again.* The status can only be modified through the CAS at Line 32, whose expected value is `ACTIVE`. If the CAS succeeds, the new value of the status can only be `SUCCESSFUL` or `FAILED`, thus any subsequent attempt to change the status will fail.

▶ **Lemma 6.** *An* MCAS *descriptor is finalized by at most one thread.* This follows from the fact that a descriptor is finalized through a CAS and the fact that an MCAS descriptor cannot change status after being finalized (Lemma 5).

▶ **Lemma 7.** *If at least one thread attempts to finalize a descriptor d, some thread will successfully finalize d.* The initial status of a descriptor is `ACTIVE`. Any thread attempting to finalize a descriptor does so through the CAS at Line 32, with expected value `ACTIVE`. Thus, at least one CAS finds the status to be `ACTIVE` and successfully changes it.

▶ **Lemma 8.** *An* MCAS *descriptor d is finalized as successful only if some thread observed all target locations of d to be acquired by d.* This is because the status is changed to successful only if the `success` variable is true at Line 32. This only happens if some thread completed the for-loop over all of d's `WordDescriptor`s without exiting the loop at Line 28. The only two ways for a thread to move to the next `WordDescriptor` in the loop is if the thread sees the current target location was already acquired by d (Line 26) or if the thread successfully acquired the current target location for d (Line 31). In both cases the thread observed the target location to be acquired by d.

▶ **Lemma 9.** *An* MCAS *descriptor d is finalized as failed only if some thread observed a target location of d to contain a different value than its expected value in d.* This is because the only way for the status to be changed to failed is if the `success` variable is false. This only happens if some thread observed the current value of a target location is different from its expected value in Line 28.

▶ **Lemma 10.** *After a location l becomes acquired by some operation op, l will never become un-acquired again.* This is because the only instruction that modifies a location l is the acquire CAS at Line 31.

We say that an operation op_1 *helps* an MCAS operation op_2 if op_1 calls MCAS with op_2's descriptor in Line 8.

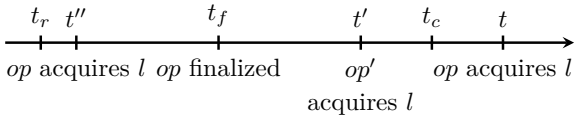
▶ **Lemma 11.** *After a location l becomes acquired by some operation op, no operation $op' \neq op$ will acquire l before op becomes finalized.* Assume by contradiction that op' acquires l after op acquires l and while op is still active. Consider op' last call to `readInternal` (Line 24) before the successful acquisition of l. During this call, op' must have observed that l is owned by op (otherwise; if op had acquired l after the call to `readInternal`, the acquisition CAS would have failed). Moreover, op was active during that call to `readInternal` by op'. Thus, op' helped op before returning from `readInternal`, finalizing op in the process. Thus op cannot be active at the time of the acquisition, a contradiction.

We say that a location l is *re-acquired* by operation op at time t if (1) l becomes acquired by op at time t, (2) there exists time $t' < t$ such that l became acquired by $op' \neq op$ at time t', and (3) there exists time $t'' < t'$ such that l became acquired by op at time t''.

▶ **Lemma 12.** *A location cannot be re-acquired.* Assume the contrary and let t be the earliest time when any location is re-acquired in a given execution E. Let l be that location and op be the operation re-acquiring it. This means that l became acquired by op at some time t'', then became acquired by some $op' \neq op$ at time $t' > t$ and then later became acquired by op at time $t > t'$ (the times in this lemma are represented in the figure below, for convenience). By Lemma 11, op must have become finalized at some time $t_f < t'$ (t_f is unique by Lemma 5).

Now consider the thread T which acquires l on behalf of op at time t. T does so through the CAS at line 31. Since op becomes finalized at time t_f, T must have performed the status check at line 29 at some time $t_s < t_f$ (otherwise T would have exited from the for loop without acquiring l). Let $t_r < t_s$ be the last time when T performed `readInternal` (line 24) before t_s. Note that $t_r < t''$, otherwise T would have seen l as already acquired by op at line 26 and continued without attempting to acquire l.

Let $\langle c,v \rangle$ be the return value of the `readInternal` call by T at t_r; this means that l's value was c at some time before t_r. Since T successfully performs the CAS at line 31 at time t, the value of l must also be c immediately before t. However, the acquisition of l at time $t'' > t_r$ changes the value of l from c. Therefore, it must be the case that some thread changes the value of l back to c at some time t_c between t'' and t. Note that c must be a word descriptor (due to Lemma 10). Since word descriptors are unique, they uniquely identify their parent operations. Therefore, l must have been owned by some operation op'' before t_r and again at t_c; this means that op'' *re-acquired* l at time t_c, contradicting our choice of t as the earliest re-acquisition time.



▶ **Lemma 13.** *A location l cannot be acquired by operation op after op is finalized as successful.* This follows from Lemmas 8 and 12.

▶ **Lemma 14.** *If op_1 helps op_2, then either op_2 highest acquired location is higher than op_1's highest acquired location, or op_1 has not acquired any locations.* If op_1 is a read operation, the

statement is trivially true. Assume now that op_1 is an MCAS operation that helps op_2 and that op_1's highest acquired location is higher than op_2's highest acquired location ($\star$). Since op_1 helps op_2, op_1 has observed one of its target locations l to be already acquired by op_2. But since op_1 iterates over locations in increasing order, l must be higher than op_1's highest acquired location. This contradicts $\star$.

We define the *helping graph at time* t, $H(t)$, as follows. The vertices of $H(t)$ are the ongoing operations at time t. There is an edge from op_1 to op_2 if op_1 is helping op_2 at t. We define the *call depth* of an operation op at time t to be the length of the longest path starting from op in $H(t)$.

▶ **Lemma 15.** *For any operation op and any time t, the call depth of op at t is finite.* Assume the contrary. Since each thread can have at most one ongoing operation at t, $H(t)$ has a finite vertex set. Let op be an operation and t be a time such that op has an infinite call depth at t. Then, $H(t)$ must contain a cycle. This is a contradiction: if the cycle contains an operation op_0 that has no acquired locations, then op_0's predecessor in the cycle cannot be helping it; if the cycle does not contain such an operation, then by traversing the cycle we would find operations with strictly increasing highest acquired locations (Lemma 14).

Informally, Lemma 15 says that while in our algorithm it is possible for operations to recursively help one another, the recursion depth is finite at any time, due to the sorting of memory locations.

We define the following predicates (recall that n is the number of threads). Let $S(k)$: "If there are $0 < k \leq n$ concurrent operations and at least one thread is taking steps and no operations are created, at least one operation will eventually return". Let $P(k)$: "If there are $0 < k \leq n$ concurrent operations and at least one thread is taking steps, at least one operation will eventually return".

▶ **Lemma 16.** *$S(k)$ is true for all k, $0 < k \leq n$.* Assume the contrary. Pick an active thread T: T is taking steps infinitely often, but no operations ever return. By Lemma 15, the call depth of T is finite, thus T must be taking some backward branch infinitely often. If T is taking the branch at Line 9 infinitely often, then MCAS operations are being finalized infinitely often (Line 9 is only executed if some operation was active at Line 7, but that same operation must be finalized by Line 9 due to the preceding MCAS call which returns only after the operation is finalized). This is a contradiction because we started with a finite number of MCAS operations and no operations are being created. If T is taking the branch at Line 31 infinitely often, then locations either (a) become acquired infinitely often or (b) change owners infinitely often. Both possibilities lead to a contradiction: (a) because there are a finite number of target locations of ongoing MCAS operation and locations never become unacquired and (b) because locations change owners only after operations become finalized, which would imply that operations become finalized infinitely often.

▶ **Lemma 17.** *$P(k)$ is true for all k, $0 < k \leq n$.* Consider the case $k = n$. $P(k)$ is equivalent to $S(k)$ in this case (no operations can be created if there are already as many operations as threads), and thus true. Consider the case $k = n-1$. If some operation is eventually created, then eventually some operation will return, by $P(n)$. If no operation is ever created, then eventually some operation will return, by $S(n-1)$. We can continue in this manner with $k = n-2,\ldots,1$, each time using either $P(k+1)$ or $S(k)$.

▶ **Lemma 18.** *Our implementation is lock-free.* This follows immediately from Lemma 17.

▶ **Lemma 19.** *Linearization point of a failed* MCAS. By Lemma 9, if descriptor d is finalized as failed by thread T at time t_1, then at time $t_0 < t_1$, T has observed some target location l

to contain a different value than l's expected value in d. We can take t_0 as the linearization point of the MCAS.

▶ **Lemma 20.** *Linearization point of a successful* MCAS. By Lemma 8, if thread T changes the status of descriptor d to successful, then T previously observed all of d's target locations to be acquired by d. Thus, when changing the status of d to successful, T changes the logical values of all target locations, marking the linearization point.

▶ **Lemma 21.** *Linearization point of a read.* The linearization point of a read is the last executed dereference instruction at Line 3.

A.3 Persistent MCAS with $k+1$ CAS and 2 Persistent Fences

Listing 4 Our MCAS algorithm for persistent memory. Commands in *italic* are related to memory reclamation, and underlined commands are related to persistence.

```
1  read(void* address) {
2      epochStart();
3      <content, value> = readInternal(address, NULL);
4      epochEnd();
5      return value; }
6
7  MCAS(MCASDescriptor* desc) {
8      epochStart();
9      success = true;
10     for wordDesc in desc->words {
11 retry_word:
12         <content, value> = readInternal(wordDesc.address, desc);
13         // if this word already points to the right place, move on
14         if (content == &wordDesc) continue;
15         // if the expected value is different, the MCAS fails
16         if (value != wordDesc.old) { success = false; break;  }
17         if (desc->status != ACTIVE) break;
18         // try to
               ↪ install the pointer to my descriptor; if failed,    retry
19         if (!CAS(wordDesc.address, content, &wordDesc)) goto retry_word;
               ↪ }
20     for wordDesc in desc->words { PERSISTENT_FLUSH(wordDesc.address); }
21     PERSISTENT_FENCE();
22     newStatus = success ? SUCCESSFUL : FAILED;
23     if (CAS(&desc.status, ACTIVE, newStatus | DirtyFlag)){
24         // if I finalized this descriptor, mark it for reclamation
25         retireForCleanup(desc); }
26     PERSISTENT_FLUSH(&desc.status);
27     PERSISTENT_FENCE();
28     parent->status = parent->status & ~DirtyFlag;
29     returnValue = (desc.status == SUCCESSFUL);
30     epochEnd();
31     return returnValue; }
```

Listing 5 The `readInternal` auxiliary function, used by our MCAS algorithm for persistent memory. Underlined commands are related to persistence.

```
32 readInternal(void* addr, MCASDescriptor *self) {
33 retry_read:
34     val = *addr;
35     if (!isDescriptor(val)) then return <val,val>;
36     else { // found a descriptor
37         MCASDescriptor* parent = val->parent;
38         if (parent != self) && parent->status == ACTIVE) {
39             MCAS(parent);
40             goto retry_read;
41         } else if (parent->status & DirtyFlag) {
42             PERSISTENT_FLUSH(&parent->status);
43             PERSISTENT_FENCE();
44             parent->status = parent->status & ~DirtyFlag;
45             goto retry_read;
46         } else {
47             return parent->status == SUCCESSFUL ?
48                 <val,val->new> : <val,val->old>;
49 }  }  }
```

A.4 Memory Management

We describe the memory management in the context of the MCAS implementation presented in Section 4.

We use two thread-local lists for reclaiming operation descriptors: One list is for descriptors that have been finalized, but not detached yet (`finalizedDescList`), and another is for descriptors that have been detached but to which readers might still hold references (`detachedDescList`).

In general, our memory management scheme is similar to an RCU (read-copy-update) implementation [44, 45]. We start with a simple blocking scheme, extending it into a non-blocking one. Each thread maintains an epoch number, incremented by the thread upon the entry to and before the exit from the `read` and MCAS functions (see, e.g., Lines 14 and 16 in Listing 3). In `retireForCleanup` function (cf. Line 34 in Listing 3), a thread adds the given descriptor to `finalizedDescList`. Once the size of this list reaches a certain threshold, the thread invokes a function similar semantically to `synchronize_rcu()` [44]. That is, it runs through all thread epochs, and waits for every epoch with an odd value (indicating that a thread is inside the `read` or MCAS functions) to advance. Once all epochs are traversed, all descriptors currently in the `detachedDescList` list can be reclaimed (returned to the operating system or put into a list of available descriptors for reuse). At the same time, all descriptors currently in the `finalizedDescList` list can be moved to the `detachedDescList` list, after replacing pointers to those descriptors in corresponding memory locations with their actual values.

To elaborate on this last step, given an `MCASDescriptor` descriptor `d` that is about to be moved from `finalizedDescList` to `detachedDescList`, a thread runs through all the `WordDescriptor`s stored in `d`. For every such `WordDescriptor` `w`, the thread checks whether `w->address` is equal to `d` and if so, writes `w->old` or `w->new` into `w->address` according to the status of `d`. The check and the write are done atomically using CAS.

The scheme presented so far is blocking—if a thread does not advance its epoch number, any thread will be unable to complete the traversal of epochs. To avoid this issue, each thread may store a local copy of all thread epochs it has seen during the last traversal. On its next epoch traversal, it compares the current and the previously seen epochs for each thread t, and if those two are different, it infers that t has made progress. If progress is detected for all threads, any descriptor that was placed into `finalizedDescList` (`detachedDescList`) before

the previous epoch traversal can be detached (reclaimed, respectively).

Note that while this scheme is non-blocking, a failure of a thread might prevent reclamation of *any* memory associated with descriptors. This is a common issue with epoch-based reclamation schemes [20], which could be resolved either by enhancing the scheme (e.g., as in [12]) or by switching to a different scheme, e.g., one based on hazard pointers [20, 46].

A.5 Managing Persistent Memory

In this section, we show how the memory management mechanism described in Section 6 is extended to manage persistent memory. Upon recovery from a crash, any pending PMCAS operation is applied using the same algorithm as presented in Listing 3. Pending PMCAS operations can be found by scanning allocated descriptors (e.g., if descriptors are allocated from a pool, similar to David et al. [17]). Moreover, since we assume the recovery is done by a single thread, we can immediately detach and recycle any finalized descriptor (after writing back the actual values into corresponding memory locations). Therefore, when considering persistent memory, the only change required to support correct recycling of descriptors (in addition to using a persistent memory allocator) is flushing all writes while detaching descriptors and introducing a persistent fence right before reclaiming descriptors from `detachedDescList`. The fence is required to avoid a situation where a detached descriptor is recycled and a crash happens while the descriptor is being initialized with new values. In this case, and if a fence is not used, some memory locations may still point to the descriptor (since updates to those locations might have not been persisted before the crash), while the descriptor may already be updated with new content. Note, though, that the flushes and the fence take place off the critical path, therefore their impact on the performance of PMCAS is expected to be negligible.

A.6 Efficient Reads

Once a memory location has been modified by an MCAS operation, even if by a failed one, it would refer to an operation descriptor until that descriptor is detached. Until that happens, the latency of a read operation from that memory location would be increased as it would have to access an operation descriptor to determine the value that needs to be returned by the read. The memory management mechanism as described above, however, would detach the descriptor only as a part of an MCAS operation. This might cause degraded performance for read-dominated workloads in which MCAS operations are rare.

To this end, we propose the following optimization for eventual removal of references to an operation descriptor and storing the corresponding value directly in the memory location as part of the read operation. If a read operation finds a pointer to a finalized operation descriptor, it will generate a pseudo-random number[2]. With a small probability, it will run a simplified version of the memory reclamation scheme described above. Specifically, it will scan epochs of all other threads, and then change the contents of the memory location it attempts to read to the actual value (using CAS). (To avoid deadlock between two threads scanning epoch numbers, a thread may indicate that it is in the middle of an epoch scan so that any descriptor can be detached, but not recycled at that time.)

[2] Generating a local pseudo-random number is a relatively inexpensive operation that requires only a few processor cycles (see, for instance, the generator in ASCYLIB [7].)

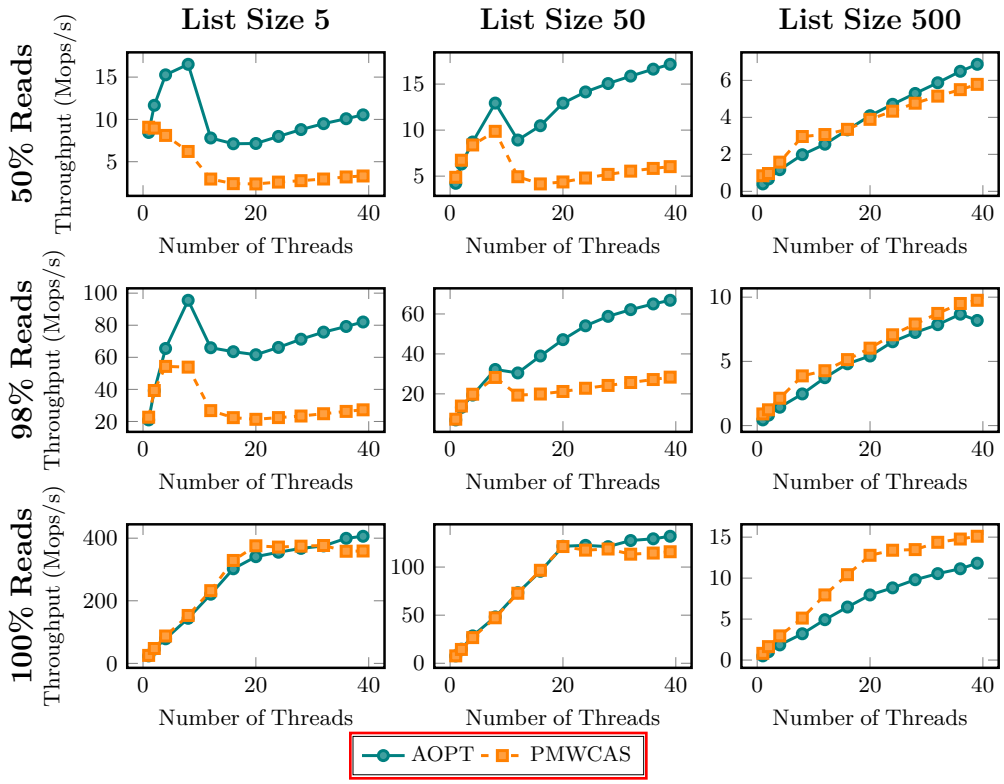


Figure 3 Doubly-linked list benchmark. Top row shows results for 50% reads workload; middle row shows results for 98% reads workload; bottom row shows results for 100% reads workload. Each column corresponds to a different initial list size (5, 50 and 500 elements, respectively).

A.7 Performance in read- and update-heavy workloads

Figures 3 and 4 show performance results for the 50%, 98%, and 100% reads workloads in the doubly-linked list and B+-tree benchmarks, respectively. All other settings are the same as in Section 7. These more extreme workloads largely magnify the performance effects demonstrated by the workloads included in the paper (see Section 7). Namely, as the write ratio increases, so does the gap by which AOPT outperforms PMwCAS in cases that involve contention between concurrent MCAS operations (i.e., small and moderate lists and trees). With 100% reads, our algorithm performs on par with or slightly trails behind PMwCAS at every contention level. Since the workload involves no update operations (and thus no MCASes), the lower complexity of our MCAS operations does not factor into the results, whereas the higher overhead of the extra-level of indirection in our read operations does factor in.

A.8 Prior Work on Non-blocking MCAS

Israeli and Rappoport [39] propose a lock-free and disjoint-access-parallel implementation based on LL/SC and show how LL/SC can be obtained from CAS. Their algorithm requires storing per-thread valid bits at each memory location, thus limiting the number of bits available for data. In the absence of contention, an a k-CAS requires $3k+2$ CAS instructions if using the LL/SC implementation from CAS provided in the paper. In their implementation, un-

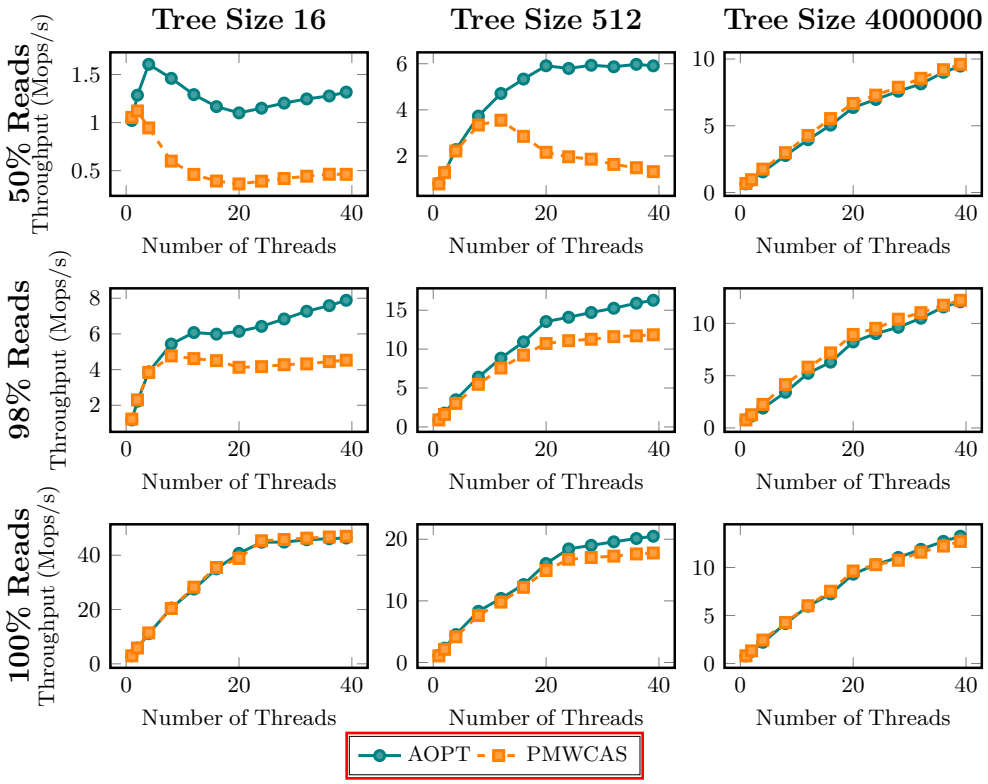


Figure 4 B+-tree benchmark. Top row shows results for 50% reads workload; middle row shows results for 98% reads workload; bottom row shows results for 100% reads workload. Each column corresponds to a different initial tree size (16, 512 and 4000000 elements, respectively).

contended reads (i.e., read operations that do not help concurrent MCAS operations) perform expensive atomic LL instructions, which can be emulated by writes to shared memory, thus limiting performance in common read-heavy workloads.

Anderson and Moir [4] propose a wait-free implementation also based on LL/SC. The strong progress guarantee comes with high space requirements: each memory word needs to be followed contiguously by an auxiliary word containing information needed to help complete an ongoing operation on the memory word.

Moir [47] simplifies [4] considerably by removing the requirement of wait-freedom. Instead, his algorithm is conditionally wait-free: it is lock-free and provides a means to communicate with an external helping mechanism which may cancel MCAS operations that are no longer required to complete.

Harris et al. [33] introduce a lock-free algorithm based on CAS operations. In order to avoid the ABA problem, the algorithm uses a double-compare-single-swap primitive (implemented using two CAS instructions, in the absence of contention) to make each target word point to a global MCAS descriptor while ensuring that the descriptor is still active. In order to distinguish between values and descriptors, the two least-significant bits are reserved in each word. In total, a k-word MCAS uses $3k+1$ CAS instructions in the uncontended case.

Ha and Tsigas [29, 30] provide lock-free algorithms which measure the amount of contention on MCAS target words and react by dynamically choosing the best helping policy.

Attiya and Hillel [8] give a lock-free implementation using CAS and DCAS that requires

$6k+2$ CAS instructions for a k-word MCAS in the uncontended case. To avoid the ABA problem, this algorithm stores a tag with each pointer which it atomically increments every time the pointer changes. The algorithm uses a conflict-resolution scheme in which contending operations decide whether to help or reset one another based on how many locations each operation acquired before the conflict was detected (preference is given to operations that own more locations). Their implementation does not provide a separate *read* operation.

Sundell [56] proposes a scheme that uses $2k+1$ CAS instructions for a k-word MCAS (in the absence of contention). An MCAS operation first uses CAS to acquire ownership of each target word, changes the status using a CAS and then uses CAS to write the final values back into the target word. The algorithm is wait-free under the assumption that there is a bound on the number of MCAS operations with equal old and new values.

Feldman et al. [22] propose an algorithm that is both wait-free and ABA-free. In their helping mechanism, a thread actively announces if it is blocked (i.e., if it fails to complete due to concurrent MCAS operations), relying on contending operations to help it to complete.

Restricted and extended multi-word operations. Previous work has explored other operations that atomically read and modify multiple words. These operations are either more general or more restricted than MCAS.

Luchangco, Moir and Shavit [42] present an obstruction-free implementation of a "k-compare-single-swap", which compares on k words but only modifies one word (more restricted than MCAS). Their algorithm is based on LL/SC, for which they give an obstruction-free implementation from CAS.

Brown et al. [11] introduce extensions to LL/SC called LLX/SCX, which are more general than k-compare-single-swap, but more restricted than MCAS. LLX/SCX primitives operate on sets of data records, each comprising several words. SCX allows modifying a single word of a data record, conditional on the fact that no data record in a specified set was modified since LLX was last performed on it. Furthermore, SCX allows finalizing a subset of the data records, preventing them from being modified again. While LLX/SCX and MCAS can be used to solve similar problems, MCAS is more generic, as it allows modifying k words atomically, whereas LLX/SCX only allow modifying a single word.

Timnat et al. [57] propose an extension of MCAS called MCMS (Multiple Compare Multiple Swap), which also allows addresses to be compared without being swapped (more general than MCAS). They provide implementations of MCMS based on HTM and on the algorithm by Harris et al. [33].