Log-Free Concurrent Data Structures

Tudor David*
IBM Research Zurich
udo@zurich.ibm.com

Aleksandar Dragojević
Microsoft Research, Cambridge
alekd@microsoft.com

Rachid Guerraoui
EPFL
rachid.guerraoui@epfl.ch

Igor Zablotchi
EPFL
igor.zablotchi@epfl.ch

Abstract

Non-volatile RAM (NVRAM) makes it possible for data structures to tolerate transient failures, assuming however programmers have designed these structures in a way to preserve their consistency upon recovery. Previous approaches, typically transactional, inherently used logging, resulting in implementations that are significantly slower than their DRAM counterparts. In this paper, we introduce a set of techniques that, in the common case, remove the need for logging (and costly durable store instructions) both in the data structure algorithm as well as in the associated memory management scheme. Together, these generic techniques enable us to design what we call *log-free concurrent data structures*, which, as we illustrate on linked lists, hash tables, skip lists, and BSTs, can provide several-fold performance improvements over previous, transaction-based implementations, with overheads of the order of milliseconds for recovery after a failure. We also highlight how our techniques can be integrated into practical systems, by introducing a durable version of Memcached that is able to maintain the performance of its volatile counterpart.

1 Introduction

Fast, non-volatile memory technologies have been intensively studied over the past years, with various alternatives such as Memristors [51], Phase Change Memory [29, 46], and 3D XPoint [38] being proposed. Nevertheless, they are only now starting to become commercially available. Such memories, referred to as *non-volatile RAM (NVRAM)*, promise byte-addressability and latencies that are comparable to DRAM, yet also non-volatility and higher density than DRAM.

From a programmer's perspective, NVRAM can be read and written using load and store instructions, identically to DRAM. However, a significant fraction of software needs to be redesigned. Unlike DRAM on the one hand, in order to take advantage of NVRAM's non-volatility, the stored data needs to be in a state that allows the resumption of execution after a transient failure (e.g., a power failure). Unlike block-based durable storage on the other hand, the granularity at which data is read and written is much finer, and the latencies much smaller. Thus, strategies that might have yielded the best performance in case of block-based storage might not be appropriate when working with NVRAM.

In this paper, we focus on the design and implementation of concurrent data structures for NVRAM. Such structures are central to many software systems [9, 12, 37, 39, 40, 42]. Ideally, in the NVRAM environment, one would like concurrent data structures which can be recovered in case of a transient failure, and whose state would reflect all completed operations up to the failure, yet whose performance and scalability resemble those of their counterparts designed for DRAM.

However, this task is complicated by the fact that neither data stored in registers, nor caches, are durable in the face of transient failures. Moreover, by default, the program does not control the order in which cache lines are evicted and written to NVRAM. Explicit instructions, which we refer to as *sync* operations, must be used to ensure that a store is written through to NVRAM at the desired point. Finally, if the user expresses their updates as transactions or critical sections containing several stores, some form of logging is necessary for the eventuality of a failure in the middle of a transaction. This log needs to be reliably written before the transaction is executed. In all these scenarios, one must wait for stores to be written to NVRAM before proceeding, which is particularly expensive: whereas when using DRAM, one would at most wait for data to be written to the L1 cache, now one has to wait for data to be written all the way to NVRAM.

Previous approaches [2, 4, 6, 20, 25, 27, 28, 33, 36, 54] to implementing data structures for NVRAM relied mainly on transactions (either explicitly, or implicitly derived from critical sections), and focused on minimizing the associated logging overhead. The motivation of our work is to remove the need for logging in the data structure altogether, without incurring additional *sync* operations. We achieve this (in the *common case*, when we can take advantage of locality in memory allocation and reclamation) by using three techniques: (1) we reduce *sync* operations in our algorithms by using a *link cache*, (2) we remove logging in the data structures by using *lock-free* algorithms, and (3) we avoid logging associated with memory management by focusing on *coarse-grained* memory tracking. We briefly discuss these three techniques below.

*Work done while the author was at EPFL.

The link cache is essentially an extremely fast, best-effort concurrent hash table stored in volatile memory, which contains data structure links that have not yet been durably written. When modifying the data structure, instead of ensuring updated links are durably written, we add them to the link cache. This enables us to avoid writing them to NVRAM one at a time. When the durable write of one of them is necessary for correctness, we batch the write-backs of all the links stored in our cache, which is significantly more efficient than waiting for writes to complete one at a time.

Logging in a concurrent data structure can be avoided using lock-free algorithms: intuitively, a lock-free algorithm must never bring a data structure in a state that prevents other threads from continuing normal execution. Thus, as long as the order of stores in the algorithm is reflected in NVRAM, no further logging is required. We also discuss how the store ordering requirement can be relaxed without jeopardizing correctness.

Finally, memory allocation and reclamation is also a central concern for concurrent data structures. Inserting and removing objects in a data structure consists of two main steps: (i) allocating (or deallocating) memory for the node, and (ii) adding (or removing) a pointer to the node in the data structure. When working with NVRAM, a crash between these two steps would result in a persistent memory leak or use-after-free problems. The traditional approach for avoiding such issues is again some form of logging. To avoid this, we propose *NV-epochs*, an epoch-based memory reclamation scheme for durable and concurrent data structures. NV-epochs groups memory nodes into memory areas, and reliably and persistently keeps track of the active (recently used) memory areas instead of individual allocations. This bookkeeping of active memory areas can be seen as the only form of logging in our approach. However, in the common case, due to locality in allocation and reclamation, the memory area an operation accesses will already be marked as active, and thus we do not have to wait for any additional store for memory leak prevention[1]. When recovering after a failure, we simply need to traverse the memory areas that were active at the moment of the crash and detect which objects belonging to these areas are still linked in the data structures. This is significantly faster than generic mark-and-sweep garbage collection for instance [1].

Each of these techniques is of independent interest, and can be used individually while maintaining its associated benefits. Together, these three techniques result in what we call *log-free durable concurrent data structures*, namely, durable concurrent data structures that, in the common case described above, require no form of logging. As we show in the paper, these data structures provide up to an order of magnitude faster updates when compared to a traditional, log-based approach, in a single-threaded, as well as in a concurrent environment. Moreover, we achieve these benefits while maintaining low recovery times in case of restarts: even for gigabyte-sized structures, the time required to recover the structure is of only a few milliseconds. In terms of correctness, our implementations guarantee durable linearizability [26]. Briefly, all the operations that were completed before a crash are reflected after recovery.

[1]For small and medium sized data structures, as we show, this covers more than 99% of memory operations.

	L1	L2	LLC	DRAM	PCM	Memristor
Read	2	6	15	50	50-70	100
Write	2	6	15	50	150	100

Table 1. Caches, DRAM, and NVRAM (projected) latencies (ns).

We also highlight the practicality of our techniques by developing *NV-Memcached*, a persistent version of Memcached [37] that is based on a lock-free, durable hash table. NV-Memcached performs similarly to the volatile memory version of Memcached (more details in Section 6.6).

Our approach is however not a silver bullet. While it is extremely efficient for small and medium-sized data structures, its benefits are less apparent for very large data structures. And, as we discuss in the paper, in a scenario with a large number of concurrent updaters, our link cache may limit scalability and might need to be turned off.

To summarize, the contributions of this paper are:

1. The *link cache*: a component that mostly eliminates sync operations in durable data structures;
2. *NV-epochs*: a coarse-grained durable memory management scheme that requires no logging in the common case;
3. A methodology for building *log-free* durable data structures starting from algorithms designed for DRAM;
4. *NV-Memcached*, a durable version of Memcached based on our techniques;
5. A library of log-free durable data structures, as well as the link cache, NV-epochs, and NV-Memcached implementations, all available at `anonymized_link`.

The rest of the paper is organized as follows. Section 2 recalls some background. We discuss our link cache in Section 3, and memory management in Section 4. We describe our log-free durable structures in Section 5. Experimental results are provided in Section 6, while related work is discussed in Section 7. Section 8 concludes this paper.

2 Background

Before presenting our techniques, we first discuss some background and basic assumptions regarding NVRAM.

Traditionally, storage has either been fast, but volatile (i.e., the data stored is lost in case of a power failure), as is the case with DRAM, or non-volatile, but slow, as is the case with flash storage for instance. However, more recently, a new class of storage that promises low latency, byte-addressability, and non-volatility is becoming available. Candidate technologies include Memristors [51], Phase Change Memory [29, 46], and 3D XPoint [38]. NVRAM latencies are expected to be somewhat larger than those of DRAM, with writes being more expensive than reads. Table 1 provides a comparison of expected PCM and Memristor latencies [49, 52, 57] with those of DRAM and caches.

As highlighted in the introduction, one of the main difficulties when working with such NVRAM stems from the fact that, by default, we do not control the order in which cache lines to which we have done stores are evicted from the caches, and actually written to NVRAM. On current Intel processors however, we can ensure that a cache line is indeed written to memory by using the `clflush` instruction. This instruction invalidates the cache line. In addition, such flushes are ordered with respect to one another: if we issue two flushes, the second one will only start executing once the first one has completed. On upcoming processors, Intel is introducing two new instructions targeted at NVRAM [21, 23]. The first one is `clflushopt`. This instruction is not strongly ordered. Therefore, we can issue a `clflushopt` before the previous one has finished; thus, multiple such instructions can proceed in parallel. `clflushopt` is only ordered with respect to store fences. The second new instruction is `clwb`, which only does a write-back to memory, without invalidating the cache line from the cache hierarchy. Like `clflushopt`, it is only ordered with respect to store fences (or to instructions that have an implied store fence, such as, for example, Compare-and-Swap).

In our work, when working with NVRAM, we consider the `clwb` instruction to ensure that cache lines are written to memory. An important observation is that since these instructions are unordered with respect to one another, when the relative order in which they are written to NVRAM is not important, batching several write-back instructions is beneficial for performance [22]. We refer to one or more write-back instructions followed by a store fence as a *sync* operation.

We assume that a machine may fail at any point in time (due to, for example, a power failure), but can be expected to be restarted and resume normal operation (transient failure). We require, as is commonly the case in practice, that only the data stored in durable main memory is still available after a crash. The data that was in a processor's registers or in the write-back caches at the moment of the crash is not available after a restart. Nevertheless, our approach would be highly beneficial and remove the need for logging on an architecture that maintains enough residual energy to flush the register and the caches in case of a power failure as well.

Moreover, similar to related work [1], we assume that a region of NVRAM can be mapped to the same region of virtual memory across restarts. Alternatively, if this is not the case, we can update persistent pointers at recovery time.

In the context of concurrent software, it is important to define correctness conditions in the face of restarts. In particular, we require a framework that allows us to reason about the state that is stored in NVRAM after a crash. For this purpose, we use the concept of *durable linearizability* introduced by Izraelevitz et al. [26]. Essentially, a durably linearizable implementation guarantees that the state of the data structure after a restart reflects a consistent operation subhistory that includes all the completed operations at the moment of the crash.

3 Avoiding frequent write-backs: the link cache

We now introduce our first technique, aimed at minimizing the number of *sync* operations in durable data structures.

3.1 Link cache overview

In linked data structures, a new node becomes visible when a link to it from an existing node is atomically inserted. Once this happens, other operations can see that the node is present. Furthermore, in many algorithms, a node becomes logically deleted when a mark is atomically inserted on a link to signal deletion. After this, all operations enquiring about the state of this node will consider the node as no longer in the structure. A node becomes unreachable when the last link to it from another node in the data structure is atomically removed.

All these operations change the fundamental state of the node, and determine the return value of other operations which depend on the particular node. In the context of NVRAM, in order to ensure durable linearizability, it is therefore essential for each operation to ensure that the data its return value directly depends on is durably written before the operation returns. Otherwise, a scenario in which the operation returns the outcome to the user, the system restarts, and the state no longer reflects the user's operation is possible.

In order to deal with this issue, the most straight-forward approach is what we call the *link-and-persist* operation. Essentially, when performing an operation that changes the state of a node, a link is atomically updated normally, but contains a mark to signal the there is no guarantee its state is persisted. The updating operation then persists the newly modified link, and once the link is guaranteed to be persisted it atomically removes the mark. If another operation whose result depends on the marked link occurs before the updating thread can persist it and remove the mark, the second operation can try to do these steps itself. This method involves no blocking, and is thus suitable for all concurrent algorithms classes, including lock-free and wait-free algorithms [16].

However, as previously discussed in Section 2, batching multiple cache line write-backs is significantly more efficient than persisting them one at a time. Therefore, we propose the following scheme: do not wait for links to be immediately

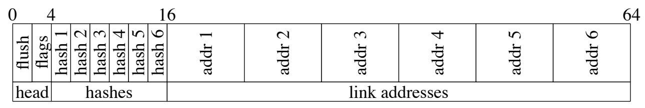


Figure 1. A bucket in the link cache.

persisted when doing an update, but place them in a fast, volatile memory cache, and write back all the links in this fast cache when an operation that directly depends on one of them occurs. We call this auxiliary structure a *link cache.* Of course, this means that clients which have inserted links into the link cache will only be notified of the successful completion of their operations once the link cache is flushed to NVRAM. The changes of a link and the insertion of a corresponding entry in the link cache must occur atomically (can be achieved in a non-blocking manner by using hardware transactional memory, or by marking the pointers to be inserted in the link cache while the operation is ongoing). If a restart happens, the modified link currently in the link cache might be lost. However, this is not problematic: the fact that these link addresses were in the cache at the moment of restart means that no operation that directly depends on them completed, and thus its outcome may or may not be visible. We thus maintain the durable linearizability property. In addition, an atomic update of an ongoing operation not being durably recorded does not leave the data structure in an incorrect state after a restart. Where ordering of durable updates is necessary, we enforce it in the data structure algorithm (see Section 5.2). The link cache is practical as long as inserting an entry in the cache is faster than waiting for a cache line to be written back to NVRAM.

3.2 Link cache implementation

We now go into details regarding the implementation of the link cache. It is important to note that the link cache does not have to reside in NVRAM, as we do not rely on its content after a restart. Our main aims were small memory footprint for the cache, non-blocking operation, and fast insertions.

With these requirements in mind, we chose to simplify the design, by making insertions to the cache best effort. The cache is only useful if it can improve the time updates spend waiting. Therefore, if an update attempts to insert an entry in the cache, but does not succeed on the first try, it gives up and persists the modified link itself instead of waiting. Thus, the link cache has constant worst case performance.

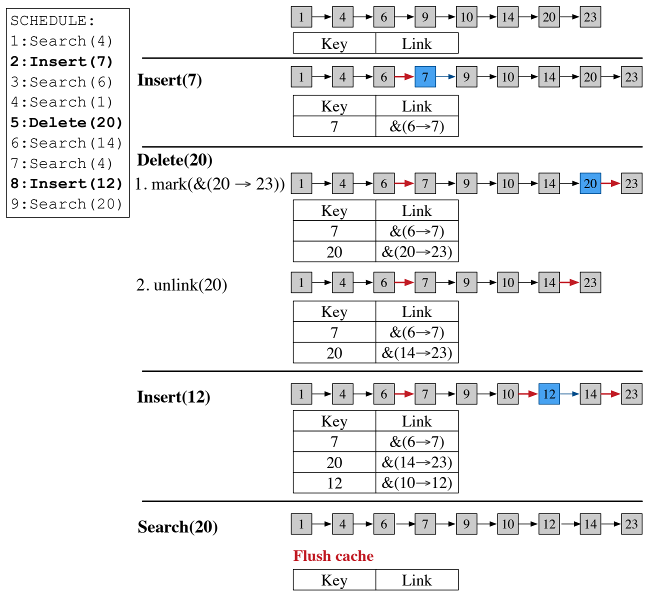
Our hash table has a configurable (but fixed throughout the execution) number of buckets. Each bucket spans exactly one cache line, and can store up to 6 links. Links concerning a particular key map to one and only one bucket. Figure 1 details the contents of a bucket. The first two bytes are used to signal whether the bucket is currently being flushed. The next two bytes are used to store the current state of each of the entries in the bucket. An entry can be free, pending or busy. We next store the 6 keys associated with the links in the bucket. In order to be able to fit 6 entries in a single cache line, instead of storing the entire key, we only store a 2-byte hash for each of the keys. While this might result in false collisions, they are extremely unlikely. With 32 buckets, we essentially have a hash space with 2M elements. Even if false collisions do occur, this is not problematic: we would simply be triggering a flush of the links in the cache when this might not have been strictly necessary. The hashes therefore require 12 bytes in each bucket. The remaining 48 bytes in the cache line are used to store the addresses of the 6 links.

The interface of the link cache has three operations, which we discuss in the following.

Try link and add. If there is space in the link cache, this operation atomically modifies the link in the data structure and inserts an entry in the link cache. The operation first tries to reserve an entry in the link cache. To this effect, it tries to atomically change the state of an entry from free to pending. If no free entry exists, or the attempt to reserve an entry is not successful, the caller is notified that the operation did not succeed and that it should persist the link itself. Once an entry is reserved, we set the corresponding key and link address in the link cache. Next, we try to update the link in the data structure. We insert the new link, but use a bit to mark the fact that for now, this link has been neither persisted, nor has its addition to the link cache been marked as completed. If the link update fails, we set the state of the link cache entry to free and return failure to the caller. We next set the state of the entry in the link cache to busy (to mark the fact that we have added the key and link address, and that the link address in fact contains the value that we want to persist). Finally, we remove the mark from the link in the data structure.

The fact that this operation is best effort, and the fact that we do several atomic updates (link marking, transitioning between multiple states) just in order to be able to handle concurrent readers make this operation an ideal candidate for the use of hardware transactional memory (HTM). In fact, we first try to execute a fast-path HTM-based operation before reverting to the code presented above. In the HTM path, we do not need to insert a marked link into the data structure, and we can avoid going through the pending state in the link cache.

Flush. This operation writes all the finalized entries in a bucket to NVRAM. The operation first atomically sets a flag to signal that it is in the process of flushing a bucket. The flush operation then issues write-backs for the link addresses in the busy entries in the bucket one by one (without waiting for the write-backs to complete) and sets the state of these entries to free. Next, the operation checks if any of the entries we have not written back have become busy (completed) in the meanwhile, and if yes, issues write-backs for them as well. This is repeated until no new busy entries appear. The



Key	Link
7	&(6→7)
20	&(14→23)
12	&(10→12)

Figure 2. Example of how the link cache is constructed.

thread then waits for the write-backs to complete by issuing a fence, resets the flushing flag, and returns.

Scan. The scan operation is given a key and searches if any link corresponding to the key is in the cache link. If such an entry is found in busy state (i.e., the insertion of the link was finalized), a flush is triggered. If an entry is found but is in pending state, the operation checks whether the new pointer has been inserted into the data structure. If this is the case, the current operation's linearization point should be after that of the operation currently inserting into the cache link, and therefore the current operation triggers a write-back of the new value of the link. Otherwise, the current operation's linearization point is before that of the update, and no further action needs to be taken. In order to guarantee durable linearizability, every data structure operation needs to call the scan method for its key, as well as for its predecessor in the structure in case of updates. However, this is as fast as reading two cache lines.

3.3 Illustration of the link cache's effectiveness

We illustrate the effectiveness of the approach through an example. We consider a lock-free linked list that uses the algorithm proposed by Tim Harris [14]. In this algorithm, in the case of inserts, once a node is properly allocated and initialized, we simply have to set the *next* pointer of its predecessor to point to it. In the case of a delete, we must first atomically mark the *next* pointer of the node to be deleted to signal logical deletion, after which the *next* pointer of its predecessor is set such that it bypasses the node to be deleted.

The schedule of operations in our example, as well as the way the link cache is constructed are presented in Figure 2. We assume an initially empty link cache, and we only depict the effects of operations that change the state of the data structure or the link cache. Normally, updates would have to wait for one link to be persisted in the case of the insert operations, and two links in the case of the delete operation (one for marking and one for deletion). However, in this example, by using the link cache, we have replaced writing back 4 cache lines one at a time by a single batch of 3 cache line write-backs.

4 Memory Management with NV-Epochs

We now address another issue that is unavoidable whenever inserting or removing nodes in a concurrent data structure: memory management.

4.1 Overview

Two separate steps need to be performed both when inserting and removing a node: in case of an insertion, memory for the new node first needs to be allocated and initialized, after which the node has to be linked into the data structure; in case of a deletion, the node is first unlinked from the data structure, and later, when we are sure no references to it exist, its memory is freed. If a restart occurs in between these two main steps both in case of an insertion and a removal, a persistent memory leak would occur: we would have allocated data that is not linked anywhere in our data structure.

The typical way of addressing the issue in the context of NVRAM is to use some form of logging: before allocating and linking, we log our intention, as we do before unlinking and freeing memory. Once the operation has (durably) completed, the log entry can be removed. However, this entails an extra write to NVRAM per update. In addition, this write needs to complete before we can proceed with the update, thus producing a non-negligible increase in the latency of updates.

In order to avoid waiting for the durable log to be written at each allocation or deallocation, we propose keeping track of active memory areas instead of keeping track of individual allocations/deallocations. Intuitively, when allocating, threads often reserve larger contiguous memory areas from which they allocate. Therefore, consecutive allocations tend to belong to the same memory area. In addition, memory reclamation schemes keep track of which objects have been unlinked, and periodically free those for which it is guaranteed that no references are held. This reclamation step is only run periodically (either at fixed time intervals, or, more commonly, when a certain number of unlinked objects have been collected) for performance considerations. Therefore, we tend to free multiple nodes at the same time. Out of these, it is usually the case that several of them map to the same memory area. Thus, there is a certain degree of locality in deallocation as well. Hence, if instead of logging every node we unlink from the data structure, we only keep track of the memory areas from which the unlinked nodes come from, we can expect significant savings in term of write-backs to NVRAM: in the common case, the memory area will already be marked as active.

While providing us with important time savings at run time, this method does defer some work for when we need to recover. In particular, we need to go over the allocated memory addresses in the active pages and check if they indeed represent nodes that are linked into the data structure. To be able to do this, we also make the assumption that we can dedicate memory pages to only store data structure nodes. To achieve this, we use an allocator specifically for such nodes.

We first briefly describe the principles of the memory reclamation scheme that we employ, after which we go into more details into how we keep track of the active memory page set, and how we recover in case a restart occurs.

4.2 Epoch-based memory reclamation

Epoch-based memory reclamation [10] is based on the following principle: if an object is unlinked, then no references to the object are held after the operations concurrent with the unlink have finished.

One method of using this principle in practice (and which we use in our reclamation scheme) is to provide each thread with a local counter, keeping track of the *epoch* the current thread finds itself in. The epoch of a thread is incremented when the thread starts an operation, and when it completes it. Thus, if the current epoch number of a thread is odd, the thread is currently active. We collect multiple objects, and free them when the vector formed by the current epochs of the threads is larger than the one when any of the objects were unlinked (only the epochs of threads that were active at the moment of unlinking need to be larger). We refer to the set of unlinked nodes which we attempt to free together as a *generation.*

4.3 Interface with NVRAM allocators

Memory allocators usually reserve a large contiguous memory address space, which they then recursively split into smaller chunks. The chunk from which an object is allocated then depends on the object's size. These chunks of contiguous memory are generally referred to as allocator *pages*. Since smaller pages from which objects such as data structure nodes are directly allocated are part of larger pages, we can configure the granularity of the pages which we keep track of. High-performance concurrent allocators usually partition the memory space for allocations among threads, such that there is minimal communication necessary between them: pages are assigned to individual threads.

Existing persistent allocators provide the capability of atomically allocating and linking (or unlinking and deallocating) objects, which, as discussed, is generally achieved through some form of logging. We do not require this capability: we only require that the persistent allocator is able to correctly maintain its durable metadata when allocating or deallocating. Moreover, in our case, the last write-back (which marks the memory as allocated or free in a thread's local allocator metadata, and is usually the only write-back the allocator issues) does not have to be completed before proceeding: in the case of an allocation, the data structure algorithm will have to wait for the write-backs to complete after the memory is initialized, while in the case of deallocations the memory reclamation scheme waits for all the deallocations it issues at once to be completed. Thus, in most cases, when the allocator only does one store to thread-local data, we do not have to issue a sync operation for the allocator metadata. Based on its metadata and our active page tables, the allocator can recover its state in case of a restart. An existing persistent allocator, such as for instance, `nvm_malloc` [49], can be used for our purposes, with the small changes we mentioned. We also require the allocator to provide a method that returns the next node address to be allocated. As allocators generally assign larger chunks of memory to individual threads, and threads do not "steal" memory from one another, adding this method is trivial.

4.4 Maintaining the set of active NVRAM memory areas

In our approach, each thread keeps a set of active memory pages. For each memory page, we also store some metadata determining when the page can be considered as no longer active and can thus be removed from the set. This metadata consists of (i) the largest epoch at which this thread has unlinked memory belonging to the page from the data structure, and (ii) the largest epoch at which this thread has allocated memory belonging to this page. The addresses of the memory pages need to be stored persistently (meaning that when we insert a new page, we have to wait for the write-back of the address of the page to complete before continuing), while the metadata is only needed for removing table entries, and is not needed in case of a restart.

We attempt to trim a thread's active page table when it exceeds a certain size. For this purpose, the metadata associated with each page is used as follows. A page from which unlinks have happened is active until the epoch-based memory reclamation scheme is guaranteed to have freed all unlinked nodes. This can be verified by having the reclamation scheme keep track of the epoch vector of the most recent generation of objects that were collected. A page from which allocations have happened is active until the insert operation has finished, i.e., while the current epoch of the thread is equal to the last epoch at which a node allocation from this page took place. When using a link cache, we also have to ensure that it contains no entries pertaining to the page under consideration. For this reason, the operation that attempts to trim the active page table issues a link cache flush as well. If all the unlinked nodes have been freed, and all the allocated nodes have been linked, the page can be removed from the table.

We use a separate persistent allocator for the active memory page table. Allocations for the table happen very infrequently (we preallocate a number of entries for each thread at start-up, and allocate multiple entries at once when more space is needed; in addition, tables usually do not grow beyond a certain size, and thus no allocations are needed from a certain point). We require that this second allocator provide the interface previous work on NVRAM memory allocators does [6, 24, 41, 49, 54]. In this instance, we used the allocator provided with `nvml` [24].

4.5 Recovery after transient failures

On recovery, we must make sure that there are no nodes that are not linked in the data structure but are allocated.

There are two ways of verifying this. The efficiency of each of these methods depends on the size of the data structure, the complexity of the search method, and the size of the memory space that needs to be verified. In both cases, we assume a well-formed data structure. That is, the recovery procedure should first ensure that the data structure is brought to a consistent state before attempting to remove memory leaks. This step is not necessary for any of the data structures we developed.

The first approach is to go over all the node addresses in the active memory pages at the moment of the crash, and, if they are allocated, perform a search in the data structure for the key the allocated address contains. If the search (i) returns a result and (ii) the address of the returned node is the same as the address we were considering, we leave the node as allocated. Otherwise, we free the node. Condition (ii) is necessary because we might have an allocated but uninitialized node. Therefore, a node with the key that we retrieve from that uninitialized memory might indeed exist in the data structure, but it might not be pointing to this uninitialized memory.

The second approach is to traverse the structure only once, and for each node check if its address belongs to the set of active pages. If this is the case, store the address of the node in a volatile memory buffer. Next, go over all the allocated node addresses in the active memory pages, and check if they are in the volatile memory buffer. If they are not, it means they are not linked in the data structure and can be deallocated.

Both of these approaches can be parallelized in order to decrease the time spent on recovery.

We note that in our implementation, it cannot be the case that a node is linked into the data structure, but not marked as allocated. This is because before linking a node in the data structure, we issue a store fence that ensures that the contents of the node, as well as the allocator metadata (for which we issue write-backs, but do not wait for last one to complete when calling the allocation method) are durably written.

5 Log-Free Durable Data Structures

In this section, we present our general approach to designing concurrent and durable data structures. We first argue that such data structures should be lock-free, and illustrate the steps we take in order to obtain correct lock-free data structure implementations for NVRAM. We focus on implementations of linked lists, skip lists, hash tables, and search trees, which are commonly used in practice [9, 12, 37, 39, 40, 42]. Nevertheless, our techniques apply to other data structures as well. Besides the optimizations presented in this section, the data structures use the previously introduced techniques, namely the link cache and NV-epochs.

5.1 The Case for Lock-free Algorithms

As discussed, previous work has taken the approach of using transactions and logging (normally either write-ahead logging or copy-on-write) to ensure the correctness of durable data structures. The log must be durably written before the updates it refers to, thus introducing frequent waiting of hundreds of cycles. Thus, when moving from volatile to durable data structures, one can expect a significant drop in performance. This is particularly problematic for small and medium-sized concurrent data structures, which would normally read and write most of the data from the write-back caches.

In order to mitigate the problem at the level of the data structure algorithms, we leverage lock-free algorithms. A concurrent algorithm is said to be *lock-free* if it can guarantee that at any point in time, at least one thread that is trying to take steps is able to make progress. As previous work has shown [7], lock-free algorithms tend to scale and perform extremely well in practice.

We argue that lock-free algorithms are even more appealing in a system using NVRAM. A corollary of the theoretical definition of lock-freedom is that in a system with n threads, if $n - 1$ threads stop executing at any point in their operation, the one remaining thread must be able to continue making progress. Otherwise, a thread stopping execution at a problematic stage could prevent the progress of all other threads, even when they are trying to make progress, thus breaking the lock-free property. In other words, no thread can at any point in its execution leave the data structure in a state that is inconsistent and from which other threads cannot continue their operation themselves. In particular, in the state-of-the-art lock-free algorithms, there is an atomic step (usually performed through an atomic compare-and-swap) which makes an update visible: enough information is introduced in the data structure through this step for any other thread to be able to complete the update. Once this atomic update is durably written, upon a restart, the update can be completed by some thread, and the data structure is thus in a consistent state. If the update is not persisted, it is as if the update had not occurred at all, and thus the data structure is of course in a consistent state. Thus, the moment when this essential update is durably written is the linearization point [19] of updates.

In the case of NVRAM, we can thus guarantee that as long as the stores of the threads are persisted in the order in which they are issued (we show how this can be relaxed), regardless of where a crash occurs, upon a restart the data structure is in a consistent state that allows the execution to resume. Therefore, we remove the need for logging for the data structure itself.

5.2 Durable Data Structure Implementations

We have implemented durable data structures for linked lists, hash tables, skip lists, and BSTs. These structures model the set abstraction, and have methods to insert, remove, and search of elements identified through a unique key. We consider one implementation per data structure type, starting from the concurrent algorithm that has been shown to provide the best performance and scalability [7]. Our linked list is based on Harris' algorithm [14], the hash table uses one Harris linked list per bucket, the skip list uses Herlihy and Shavit's lock-free algorithm [18], while the BST uses the algorithm proposed by Natarajan and Mittal [44]. Other algorithms and data structures can be similarly modified.

We illustrate our approach on the skip list, as most of the issues that appear in the context of other data structures appear in the case of the skip list as well.

Illustration: Log-free Durable Skip list. We start from a version of Herlihy and Shavit's algorithm that uses the optimizations proposed by Asynchronized Concurrency [7]. In a volatile memory environment, searches in this algorithm are wait-free and perform no stores. Insert operations link a new node in the data structure starting from the bottom level, and progressively link the node in its higher level lists. Remove operations first mark a node's next pointers to signal logical deletion (starting from the node's top level next pointer, and going down to the bottom level), after which the node is physically unlinked from the skip list one level at a time (again, starting from its top level). We note that nodes in a skip-list may span multiple cache lines.

One particularity of this algorithm is the very fact that it does not guarantee that the skip list is well formed. Due to concurrency, it might be the case that a node is present in some higher-level list, but not in (some of) the levels below. It is also possible that a node that was unlinked from all the skip-list levels might reappear (in a state that is marked for deletion) in one of the higher skip list levels. However, since the membership of a node in a skip list is determined by its presence or absence in the bottom-level list, this does not affect correctness. The higher-level lists are used simply for performance reasons.

In a sense, this makes this algorithm particularly appealing for NVRAM: the algorithm itself can work with a data structure that is not completely well-formed, so we can take advantage of this and not ensure that every update to the higher levels of the skip list is persisted, since we know that at recovery we can continue operation even if certain links are missing. Thus, for the higher-level links, we do not issue write-backs and wait for them to be persisted one at a time. We only issue these write-backs at the end of the operation, but we do not wait for them to be completed before proceeding.

In another sense, this causes difficulties at run-time: the fact that at the end of a delete operation we cannot guarantee the fact that a node is no longer reachable in the data structure makes memory reclamation problematic. To address this, we identify the scenarios under which a node may still be traversed even after its delete operation has completed. We modify the original algorithm such that once all the updates that were concurrent with a delete operation finish, the deleted node is guaranteed to no longer reachable in the data structure (even though at the end of the delete operation itself it may still be). This is sufficient for our purposes, as this is the same mechanism our memory reclamation scheme uses. We believe these optimizations may be useful when using the algorithm in a volatile memory setting as well.

Whenever we insert or remove a node from the bottom-level list, in order to ensure durable linearizability, we can either use the previously described *link-and-persist* technique, or our link cache. Before performing an update, in order to provide durable linearizability, we must ensure previous updates concerning the node's key, as well as nodes directly related to the update have been durably written. We discuss these aspects in the following.

Correctness. Our data structures are linearizable, since we start from linearizable algorithms and add only flushes or link cache operations, which do not impact linearization points. We also ensure two additional properties [11]. First, each update operation ensures that its changes are durable before returning (when using the link cache, this happens after a cache flush). Second, each operation O ensures that all operations O depends on (that involve some of the same nodes and were already linearized) are also durably linearized before O makes changes. Together, these two properties ensure durable linearizability, because they ensure that after a restart, the data structure reflects a consistent cut [26] of the history including all operations that completed before the crash and potentially some operations that were ongoing when the crash occurred.

The first property is easily ensured by durably writing any new edges or nodes introduced by an operation. Making sure an edge e is durably written just means checking if *e* is marked (or in the link cache), and issuing write-backs only if *e* is not yet durable. The second property is achieved because operations ensure that (1) before an edge is modified, the edge is durably written and (2) incoming and outgoing edges (*adjacent edges*) of nodes involved in the operation are durable before proceeding.

We detail point (2) on a linked list (similar considerations apply for the other linked data structures). For a successful search, we make sure adjacent edges to the returned node are durably written before returning. For a failed search, we make sure the node is durably unreachable before returning (e.g., in the case where a node is marked but not yet durably unlinked). For the parse phase of a modify operation (insert or delete), we take the same steps as for a search. For an insert, we also ensure that adjacent edges to the predecessor are durable before linking the new node. For a delete, we ensure that adjacent edges to the target node *T* and to *T*'s predecessor are durable before unlinking the target node. In all cases, if an edge *e* has changed between the time *e* is read and the time we try to durably write *e*, then the operation that changed *e* made sure *e* was durable.

6 Evaluation

We now study the impact our proposed techniques have on practical data structures. We look at the overall performance improvements, as well as at the behavior of components such as the link cache, and the active page tables.

6.1 Experimental setup

We run our experiments on an Intel Xeon machine that has four E7-4830 v3 12-core sockets. The cores operate at 2.1-2.7 GHz, while cache sizes are 32KB (L1), 256KB (L2), and 30MB (LLC, per die). We work with key-value pairs, both of which are 64 bytes in size. Larger values can be accommodated by using a pointer instead of the 64-byte value. Experimentally obtained values are the median of 5 repetitions.

As neither NVRAM with latencies comparable to DRAM, nor processors providing the `clwb` instruction are available yet, we write data to DRAM, simulate the `clwb` instruction, and inject software created delays, similar to previous work [3, 6, 33, 53, 54]. Intel reports issuing several flushes with `clflushopt` can be up to an order of magnitude faster than flushing them one at a time using `clflush` [22]. We assume similar performance characteristics for the `clwb` instruction. Moreover, we assume an NVRAM write latency of 125ns, which is an average of the projected values.

6.2 Data structure performance

We first look at the run time behavior of our data structures. We focus on updates, as it is these operations that must be durably recorded in NVRAM. We compare our implementations with alternatives that use lock-based critical sections (and thus use logging). We find that in the context of such data structures, an approach that uses redo logging provides good performance in addition to ensuring durable linearizability. We use the algorithms that we find perform best for each data structure: the lazy linked list [15], Herlihy's lock-based skip list [17], bst-tk [7], and a hash table with a lazy linked list per bucket.

In Figure 3, we show the increase in the number of updates per second obtained by using our structures relative to log-based implementations. We use a workload where 50% of the operations are inserts of random keys, while 50% are removes of random keys, and show results for 1 and 8 concurrent updating threads. We show relative improvements, as the precise latencies are dependent on the assumptions made about `clwb` instruction performance, as well as NVRAM store latencies.

Our method yields important benefits regardless of the data structure type. In particular, for the skip list, where in a log-based implementation a logarithmic number of locks are held while a logarithmic number of updates must be logged, our approach results in an order of magnitude increase in performance. We note that for small and medium sized data structures, we obtain significant improvements by applying our techniques. For large structures however, our improvements become less impressive. There are two main reasons for this. The first is that as the structure size increases, the latency of an update becomes dominated by the time needed to reach the point in the data structure where the modification needs to be made, both due to the need to traverse more pointers, and because when the structure does not fit in the caches anymore, reads become more expensive. In the case of the linked list in this experiment, it is in fact the only factor that is responsible for the decrease in latency improvement. The second reason has to do with a decrease in the efficiency of our active page tables for deallocations as the structures become large. We discuss why this is, as well as ways of alleviating the issue in Section 6.4. In addition, as the number of concurrent updating threads increases, the link cache becomes somewhat less efficient, as we discuss in the following. Thus, for high degrees of concurrency, we can turn the link cache off.

To summarize, it is important to note that while the precise magnitude of the improvements of our approach may depend on the characteristics of the NVRAM technology being used, this experiment has shown that our approach is beneficial for all the situations we have considered.

6.3 Link cache efficiency

The link cache technique is an optimization that can be turned on or off in an algorithm. In this experiment, we identify the scenarios under which it is beneficial.

The main potential barrier to link cache performance is scalability, given that it is a structure occupying a small number of cache lines, which is repeatedly accessed by all threads.

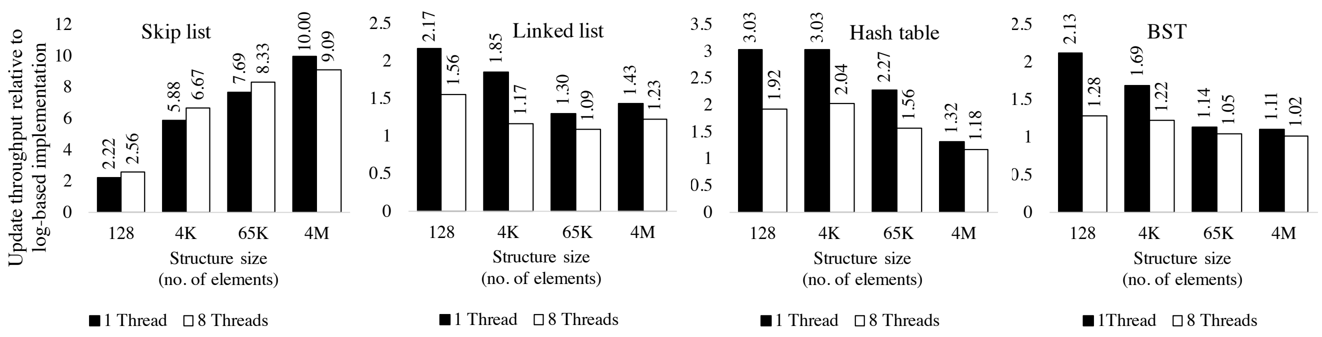


Figure 3. Update throughput improvements compared to redo log based implementations.

We evaluate an algorithm with and without the link cache, and measure its scalability. In this experiment, we use a hash table as a base data structure, due to the small latencies of updates, which allows us to stress the link cache. Each thread continuously issues insert and remove operations (thus, no read operations). The link cache occupies 32 cache lines.

Figure 4 presents the throughput of the hash table version using the link cache, normalized to the throughput of the hash table version not using the link cache. While with one thread, the link cache improves throughput by ~50%, as the contention increases, the link cache becomes less effective, until, with 12 concurrent threads, we observe no benefit.

Thus, this experiments highlights two main points (1) the link cache can be extremely beneficial up to moderate degrees of concurrency; however, (2) as its cache lines become contested, its benefits become less apparent.

We note that the link cache is the only component we add which requires inter-thread coordination, and is thus the only potential impediment to scalability.

6.4 Active page table efficiency

We now look at the efficiency of our active page table mechanism. The active page table is only efficient if it saves *sync* operations: that is, if an important fraction of updates do not have to write active page table entries.

In this experiment, we consider 4KB memory pages, and we try to trim an active page table when it exceeds 16 elements.

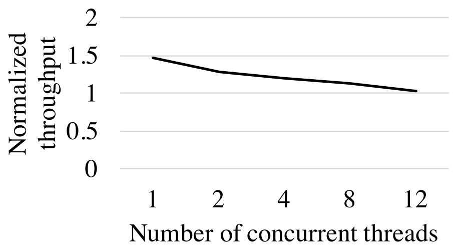


Figure 4. Throughput when using the link cache normalized to throughput when not using link cache, for a 1024-element hash table, with all operations being updates.

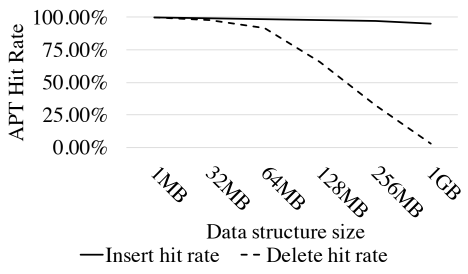


Figure 5. Active page table hit rates.

We run a data structure algorithm, and measure the fraction of allocations and deallocations that do not need to add an entry to the active page tables (that is, the fraction of hits in the active page table). Results are shown in Figure 5. In this experiment, we have used a skip list. Results are similar for other data structures, as the important factor is the data structure size, rather than its type.

We note that the hit rate is close to 100% for allocations, regardless of data structure size. In case of deallocations, the hit rate starts decreasing after the structure exceeds 64 MB (that is for data structures of more than 1M nodes; or more than 0.5M in the case of skip lists). This is because as the amount of used memory increases, there is less locality in memory reclamation steps. However, fast memory allocation and deallocation is particularly important for small data structures that fit in the write-back caches, and which have small access latencies. In such situations, our approach is effective for both types of operations.

The parameters of our system can be adjusted to deal with deallocations in larger data structures as well. The granularity at which we keep track of active memory areas is adjustable. By using larger size pages, we can improve the hit rate. This coarser granularity would however result in a somewhat larger recovery time after a restart. Additionally, we can also maintain larger active page tables (in terms of the number of entries). However, this would make active page table operations slightly slower at run time. Moreover, the number of nodes that our memory management scheme

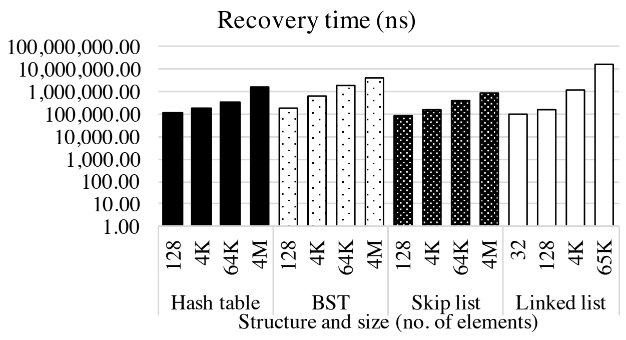


Figure 6. Data structure recovery times.

stores in a generation (and thus frees as the same time) can be increased.

To summarize, our memory management scheme is useful regardless of the size of the data structure, but is particularly efficient for small and medium-sized data structures.

6.5 Recovery

We now measure the time it takes to recover a data structure. We stop execution of the concurrent algorithm at a certain point, and then traverse its active pages and check for memory leaks. Prior to recovering, we ensure the structure's data is not in the write-back caches. We show recovery times for the various data structures as a function of their size in Figure 6.

For hash tables, BSTs, and skip-lists, which have fast search methods, recovery is extremely efficient: even in structures with 4M elements, we can recover in less than 5ms. Recovery time for such structures is two orders of magnitude lower than doing a full mark-and-sweep pass in this environment [1]. In the case of the link list, which has a linear search method, in order to avoid repeated passes over the entire structure, we employ a strategy similar to mark-and-sweep. Recovery time in this case is somewhat slower: a linked list with 64K elements can be recovered in 16ms. For all structures, recovery time increases with data structure size. Small structures tend to have a smaller number of active pages at any point in time. In addition, the search operations must traverse more pointers for larger structures, and data is less frequently present in the higher-level caches. We believe the recovery times we observed in this section are acceptable in case of a reboot.

6.6 NV-Memcached

We now show how our techniques can be applied in a larger context by developing an object caching system for durable memory: *NV-Memcached.* The main idea behind *NV-Memcached* is to make Memcached persistent by replacing its core data structures—the hash table and the slab allocator—with persistent versions. Straightforward as this idea may seem, it does entail interesting technical challenges.

First, Memcached uses a lock-protected sequential hash table; thus replacing it with our persistent non-blocking hash table would negate the latter's lock-freedom. We solve this challenge by basing *NV-Memcached* on *memcached-clht* [34], a version of Memcached that avoids protecting the hash table with locks by employing a concurrent hash table—CLHT [7], and replacing CLHT with our persistent hash table.

The second challenge is related to the recovery of items. With a naive implementation of a persistent slab allocator, it is possible for memory leaks to occur after a restart. An item can be allocated, but not yet linked in the hash table, or an item can be unlinked from the hash table but not yet marked as free in the allocator. We address this issue with a similar approach to our active page technique (Section 4): we keep track of active slabs. During recovery, we iterate over each thread's active slab table and free any memory which is marked as allocated but not yet or no longer reachable from the hash table.

We compare the performance of *NV-Memcached* to that of Memcached using `memtier-benchmark` [47]. The benchmark runs for a predetermined amount of time, issuing a mix of `get` and `set` operations using keys drawn uniformly at random from a given key range. The key range and the ratio of `get` to `set` operations are configurable parameters. Before each experiment, we warm up the cache by inserting items covering half of the key range. Both the server and the client are run with the default number of threads (4). The results are averaged over 5 runs for each configuration.

The first experiment compares the throughput of *NV-Memcached* and Memcached for three different key range sizes, under a 10:1 `set` to `get` ratio. As we can see in Figure 7, there is no notable performance drop between *NV-Memcached* and Memcached. While this is partly due to the fact that *NV-Memcached* uses a faster, more scalable concurrent hash table, the comparable performance also shows that our techniques remain practical when applied to real-world applications.

The second experiment compares, for three different key range sizes, the warm-up time of Memcached (the time it takes to populate the cache with half of the key range) to the

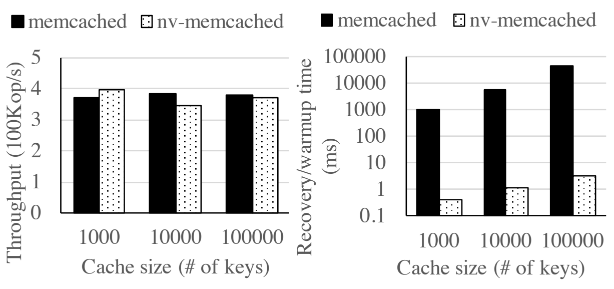


Figure 7. Performance and warm-up time comparison of Memcached and NV-Memcached.

recovery time of *NV-Memcached* (the time it takes to recover after a restart). Figure 7 shows that populating a (volatile) Memcached instance with items can take up to four orders of magnitude more time than recovering a *NV-Memcached* instance of the same size. This justifies the practicality of a non-volatile memory caching service—recovering such a service after a machine restart takes just a fraction of the time necessary for its volatile counterpart to get re-populated (and thus be useful again).

7 Related Work

Several approaches have used transactions as means of interacting with NVRAM [2, 6, 8, 13, 25, 27, 28, 33, 36, 54]. Yet, the significant overhead associated with transactional logging, inherent to such methods, has been recently highlighted (e.g., [50]), and several attempts to alleviate the problem have been proposed. Izraelevitz et al. [25] introduce an approach in which if failures occur within a critical section, upon recovery the critical section can simply be run to completion instead of reverting to previous state. To achieve this, one simply needs to reliably keep track of the last store instruction performed by each thread. While efficient in a scenario where write-back caches can be assumed to be persistent as well, the approach essentially requires a write to the log for each store instruction in a critical section. Kolli et al. [28] focus on static transactions in lock-based applications, and attempt to minimize persist dependencies in order limit waiting time. The authors also show how the commit stage of transactions can be performed while not holding any locks. In the same vein, Kamino-Tx [36] uses a copy-on-write technique, and avoids logging in critical sections. DudeTM [33] optimizes redo logging by first executing the transaction and obtaining a redo log in volatile memory, then atomically flushing the redo log to persistent memory, and only then modifying the original data.

In this paper, we go beyond optimizations to logging: we provide a method that in the common case allows us to circumvent such logging altogether in the context of concurrent data structures.

Several efforts [2, 4, 20] have been dedicated to the generation of correct durable applications for NVRAM from existing code. These approaches generally assume lock-based code. Due to their general-purpose nature, they incur additional overheads, in particular due to logging, when compared to our method, which is specialized to concurrent data structures.

Several proposals for indexing trees for NVRAM have been made [5, 30, 45, 52, 56]. However, they either require logging in some form, or do not address potential memory leaks during new node creation. In addition, the techniques cannot be broadly generalized to other data structures. Friedman et al. [11] introduce lock-free algorithms for durable queues, but do not go beyond this data structure, or consider memory management.

The problem of general memory allocation and reclamation for NVRAM has also received a lot of attention. Generic persistent memory frameworks [2, 6, 54] handle allocation and reclamation as part of the transaction mechanisms they provide, and thus use logging to ensure correctness. nvm_malloc [49] provides an interface to allocate and free persistent memory, but because of the fine-grained accounting, incurs significant overheads for each allocation and deallocation. Makalu [1] and NVthreads [20] also keep track of allocator metadata at a coarser-grain level. However, they incur higher costs at recovery time, as they require a garbage collection pass over the entire memory. Unlike all these approaches, we propose a method that is highly tuned to concurrent data structures. Thus, we are able to minimize overheads both at run-time (by efficiently keeping track of active memory areas and not requiring inter-thread coordination), as well as at recovery time (by avoiding a full mark-and-sweep pass). Additionally, our memory management scheme builds upon basic memory allocators and deals with the issue of memory reclamation as well. Thus, our scheme can take advantage of an efficient memory allocator at its core.

The design of the buckets in our link cache shares some similarities with the volatile metatdata hash table of RAMCloud [48], the software cache of Li et al. [31], CLHT [7] and the hash index of MICA [32]. Fundamentally, our cache is different from alternatives through its use of HTM, and in the sense that it is best-effort - we prioritize common case performance over the certainty of being able to insert a link. This however does not jeopardize the correctness of our approach.

Other Memcached adaptations for NVRAM have been proposed, but they use transactions extensively [6, 35, 43] or they do not guarantee all completed requests are durable [55], whereas *NV-Memcached* ensures all completed requests are durable and limits transactions to the slab allocator code by using our log-free hash table.

8 Concluding Remarks

In this paper, we introduced an approach yielding fast and durable concurrent data structures. By using lock-free algorithms, we avoid excessive writes to NVRAM. By using a link cache, we mostly eliminate waiting for the completion of write-backs, regardless of the data structure. By keeping track of memory allocations and deallocations at a coarser grained granularity, we avoid logging associated with these operations at run time, at the cost of modest increases in recovery time.

Yet, we do not claim our approach is a silver bullet. While it is extremely effective for small and medium-sized concurrent data structures, which are common in practical situations, our method is somewhat less effective for large data structures. Moreover, our approach is applicable to concurrent objects for which efficient lock-free algorithms exist. While this is the case for most common data structures, our approach is not as generic as a complete transactional system.

References

[1] Kumud Bhandari, Dhruva R Chakrabarti, and Hans-J Boehm. Makalu: Fast recoverable allocation of non-volatile memory. OOPSLA 2016.

[2] Dhruva R Chakrabarti, Hans-J Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. OOPSLA 2014.

[3] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D Viglas. Rewind: Recovery write-ahead system for in-memory non-volatile data-structures. VLDB 2015.

[4] Himanshu Chauhan, Irina Calciu, Vijay Chidambaram, Eric Schkufza, Onur Mutlu, and Pratap Subrahmanyam. NVMove: Helping Programmers Move to Byte-Based Persistence. INFLOW 2016.

[5] Shimin Chen and Qin Jin. Persistent b+-trees in non-volatile main memory. VLDB 2015.

[6] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. ASPLOS 2011.

[7] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. ASPLOS 2015.

[8] Joel Edward Denny, Seyong Lee, and Jeffrey S. Vetter. Language-Based Optimizations for Persistence on Nonvolatile Main Memory Systems. IPDPS 2017.

[9] Facebook. RocksDB. http://rocksdb.org.

[10] Keir Fraser. *Practical lock-freedom.* PhD thesis, University of Cambridge, 2004.

[11] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. Brief Announcement: A Persistent Lock-Free Queue for Non-Volatile Memory. DISC 2017 (to appear).

[12] Google. LevelDB. http://leveldb.org.

[13] Yonatan Gottesman, Joel Nider, Ronen Kat, Yaron Weinsberg, and Michael Factor. Using Storage Class Memory Efficiently for an In-memory Database. SYSTOR 2016.

[14] Timothy L Harris. A Pragmatic Implementation of Non-blocking Linked Lists. DISC 2001.

[15] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III, and Nir Shavit. A Lazy Concurrent List-Based Set Algorithm. In *Principles of Distributed Systems*, volume 3974. 2006.

[16] Maurice Herlihy. Wait-free Synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[17] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A simple optimistic skiplist algorithm. SIROCCO 2007.

[18] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[19] 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.

[20] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. NVthreads: Practical Persistence for Multi-threaded Applications. EuroSys 2017.

[21] Intel. Intel Architecture Instruction Set Extensions Programming Reference. https://software.intel.com/sites/default/files/managed/b4/3a/319433-024.pdf.

[22] Intel. Intel64 and IA-32 Architectures Optimization Reference Manual. https://www-ssl.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[23] Intel. Intel64 and IA-32 Architectures Software Developers Manuals Combined. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[24] Intel. NVM Library. http://pmem.io.

[25] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via JUSTDO logging. ASPLOS 2016.

[26] Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. Linearizability of persistent memory objects under a full-system-crash failure model. DISC 2016.

[27] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. NVWAL: exploiting nvram in write-ahead logging. ASPLOS 2016.

[28] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M Chen, and Thomas F Wenisch. High-performance transactions for persistent memories. ASPLOS 2016.

[29] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 2–13. ACM, 2009.

[30] Se Kwon Lee, K Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H Noh. Wort: Write optimal radix tree for persistent memory storage systems. FAST 2017.

[31] Pengcheng Li, Dhruva R Chakrabarti, Chen Ding, and Liang Yuan. Adaptive Software Caching for Efficient NVRAM Data Persistence. IPDPS 2017.

[32] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. MICA: a holistic approach to fast in-memory key-value storage. NSDI 2014.

[33] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. DUDETM: Building Durable Transactions with Decoupling for Persistent Memory. ASPLOS 2017.

[34] LPD-EPFL. memcached-clht. https://github.com/LPD-EPFL/memcached-clht.

[35] Virendra Marathe, Margo Seltzer, Steve Byan, and Tim Harris. Persistent Memcached: Bringing Legacy Code to Byte-Addressable Persistent Memory. HotStorage 2017.

[36] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic In-place Updates for Non-volatile Main Memories with Kamino-Tx. EuroSys 2017.

[37] Memcached. http://www.memcached.org.

[38] Micron. 3d xpoint technbology. https://www.micron.com/about/our-innovation/3d-xpoint-technology.

[39] MonetDB. http://www.monetdb.org.

[40] MongoDB. http://www.mongodb.org.

[41] Iulian Moraru, David G Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan Binkert. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. TRIOS 2013.

[42] MySQL. http://www.mysql.com.

[43] Sanketh Nalli, Swapnil Haria, Mark D Hill, Michael M Swift, Haris Volos, and Kimberly Keeton. An Analysis of Persistent Memory Use with WHISPER. ASPLOS 2017.

[44] Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. PPoPP 2014.

[45] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory. SIGMOD 2016.

[46] Moinuddin K Qureshi, Vijayalakshmi Srinivasan, and Jude A Rivers. Scalable high performance main memory system using phase-change memory technology. *ACM SIGARCH Computer Architecture News*, 37(3):24–33, 2009.

[47] Redis Labs. NoSQL Redis and Memcache traffic generation and benchmarking tool. https://github.com/RedisLabs/memtier_benchmark.

[48] Stephen Mathew Rumble. *Memory and object management in RAMCloud.* PhD thesis, Stanford University, 2014.

[49] David Schwalb, Tim Berning, Martin Faust, Markus Dreseler, and Hasso Plattner. nvm malloc: memory allocation for NVRAM. ADMS 2015.

[50] Seunghee Shin, James Tuck, and Yan Solihin. Hiding the long latency of persist barriers using speculative execution. ISCA 2017.

[51] Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. The missing memristor found. *Nature*, 453(7191):80, 2008.

[52] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, Roy H Campbell, et al. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. FAST 2011.

[53] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M Swift. Aerie: Flexible file-system interfaces to storage-class memory. EuroSys 2014.

[54] Haris Volos, Andres Jaan Tack, and Michael M Swift. Mnemosyne: Lightweight persistent memory. ASPLOS 2011.

[55] Xingbo Wu, Fan Ni, Li Zhang, Yandong Wang, Yufei Ren, Michel Hack, Zili Shao, and Song Jiang. NVMcached: An NVM-based Key-Value Cache. APSys 2016.

[56] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. Nv-tree: Reducing consistency cost for nvm-based single level systems. FAST 2015.

[57] Yiying Zhang and Steven Swanson. A study of application performance with non-volatile main memory. MSSR 2015.