Log-Free Concurrent Data Structures

Tudor David, ***IBM Research, Zurich;*** **Aleksandar Dragojevic,** ***MSR Cambridge;***
Rachid Guerraoui and Igor Zablotchi, ***EPFL***

https://www.usenix.org/conference/atc18/presentation/david

This paper is included in the Proceedings of the
2018 USENIX Annual Technical Conference (USENIX ATC '18).

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-939133-02-1

Open access to the Proceedings of the
2018 USENIX Annual Technical Conference
is sponsored by USENIX.

Log-Free Concurrent Data Structures*

Tudor David† *IBM Research, Zurich*
Aleksandar Dragojević *MSR Cambridge*
Rachid Guerraoui *EPFL*
Igor Zablotchi *EPFL*

Abstract

Non-volatile RAM (NVRAM) makes it possible for data structures to tolerate transient failures, assuming however that programmers have designed these structures such that their consistency is preserved upon recovery. Previous approaches are typically transactional and inherently make heavy use of logging, resulting in implementations that are significantly slower than their DRAM counterparts. In this paper, we introduce a set of techniques aimed at lock-free data structures that, in the large majority of cases, remove the need for logging (and costly durable store instructions) both in the data structure algorithm and 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 presenting a durable version of Memcached that maintains 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 [53], Phase Change Memory [31, 49], and 3D XPoint [39] being proposed. Nevertheless, these technologies are only now starting to become commercially available. Referred to as *non-volatile RAM (NVRAM)*, they 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 to work with NVRAM. 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 for NVRAM.

*Supported in part by ERC Grant 339539 (AOC).

†Work done while the author was at EPFL and MSR Cambridge.

In this paper, we focus on adapting to NVRAM the design and implementation of an essential component of modern software systems: concurrent data structures [10, 13, 38, 40, 41, 43]. Ideally, in the NVRAM environment, one would like concurrent data structures that (a) can be recovered in case of a transient failure with states that reflect all completed operations up to the failure, yet (b) whose performance and scalability resemble those of their counterparts designed for DRAM.

This task is challenging because neither data stored in registers, nor in caches, is 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. In order to enforce ordering, specific (and expensive) 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.

Previous approaches [2, 4, 6, 20, 25, 28, 30, 33, 37, 56] to implementing data structures for NVRAM rely mainly on transactions (either explicitly, or implicitly derived from critical sections). With transactions, some form of *logging* is necessary to cope with the possibility of a failure in the middle of a transaction. This log needs to be reliably written before the transaction is executed. This entails waiting 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. Logging is thus a major source of expensive sync operations.

We propose three techniques aimed at lock-free data structures that remove logging entirely from data structure operations and dramatically reduce logging in the memory reclamation scheme. We focus on lock-free algorithms, because they always keep the data structure in a consistent state and thus do not inherently require logging in order to maintain consistency across restarts.

The techniques we propose are: (1) the *link-and-persist* technique, which allows atomically changing and persisting a link in a data structure, (2) the *link cache*, which allows persisting entire batches of modified links, thus reducing the number of *sync* operations and (3) *NV-epochs*, a coarse-grained epoch-based memory reclamation scheme. We briefly describe these techniques below.

Link-and-persist applies the pointer marking technique [15] from concurrent programming to ensure atomicity in the face of crashes and recoveries. With link-and-persist, when a data structure link is modified, a mark is

added to the link to signify that the value of the link might not be durable yet. The link can then be persisted, and the mark removed, by the modifying operation or by any other operation (helping).

The *link cache* is 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 written to NVRAM, we add them to the link cache. Thus, we 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 links stored in our cache, which is significantly faster than waiting for writes to complete one at a time.

Finally, memory allocation and reclamation is also a central concern for concurrent data structures. When working with NVRAM, the traditional approach for avoiding persistent memory leaks or use-after-free problems is again some form of logging. To avoid this, we propose *NV-epochs*, a coarse-grained epoch-based memory reclamation scheme for durable and concurrent data structures. NV-epochs groups memory nodes into memory areas, and reliably and durably 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, we make the observation that most of the time, allocation and reclamation exhibit locality[1]. Therefore, logging can be sidestepped entirely in this case, because 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. 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 three techniques is of independent interest, and can be applied individually while maintaining its associated benefits. Together, these techniques can be used to produce what we call by abuse of language *log-free durable concurrent data structures*, namely, durable concurrent data structures that, in the large majority of cases described above, require no logging whatsoever. As we show in the paper, these data structures provide up to an order of magnitude faster updates than a traditional log-based approach, both in single-threaded and in concurrent environments. 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 completed before a crash are reflected after recovery.

We also highlight the practicality of our techniques by developing *NV-Memcached*, a durable version of Memcached [38] that is based on a lock-free, durable hash table. NV-Memcached performs similarly to the volatile memory version of Memcached.

Still, our approach is not a silver bullet. While it largely removes the cost of logging for all data structure sizes, this is especially beneficial for small and medium-sized data structures. Indeed, (1) these data structures exhibit high locality in memory allocation/deallocation, as we show in the paper, and (2) the relative cost of sync operations is higher for these data structures, as opposed to larger structures, where other costs, such as traversal, dominate.

To summarize, the contributions of this paper are:

1. *Link-and-persist*: a methodology for designing data structures with no logging in the main operations;
2. *Link cache*: a component that largely eliminates sync operations in durable data structures;
3. *NV-epochs*: a durable memory management scheme in which only a fraction of operations do any logging;
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 `go.epfl.ch/nvram`.

The rest of the paper is organized as follows. In § 2 we recall relevant background. We describe our link-and-persist technique in § 3. We discuss our link cache in § 4, and memory management in § 5. We show experimental results in § 6 and discuss related work in § 7.

2 Background

Traditionally, storage has either been fast, but volatile (i.e., data 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. NVRAM latencies are expected to be somewhat larger than those of DRAM, with writes being more expensive than reads. Table 1 compares expected PCM and Memristor latencies [51, 54, 59] to those of DRAM and caches.

As highlighted in the introduction, one of the main difficulties when working with NVRAM stems from the fact that, by default, we do not control the order in which cache lines to which we have performed stores are evicted from the caches, and actually written to NVRAM. However, there are current and upcoming instructions [21, 23] on Intel processors[2], which allow us to ensure that a cache line is indeed written to memory. In this paper, we consider

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

[2] In this work, we assume a TSO-like memory model; our work can be extended to more relaxed memory models as well.

	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).*

the *clwb* instruction, because it (a) writes-back a cache line without invalidating it—as opposed to *clflushopt*—and (b) it is only ordered with respect to fences (or to instructions that have an implied store fence, such as, for example, Compare-and-Swap)—as opposed to *clflush*. Property (b) is especially beneficial to performance, since it allows multiple cache-line write-backs to proceed in parallel [22]. We refer to one or more such instructions followed by a store fence as a *sync* operation.

A machine may fail at any point in time (e.g., due to a power failure), but can be expected to restart and resume normal operation (transient failure). We assume, as is commonly done 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.

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. 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 The Link-and-Persist Technique

In this section, we present our *link-and-persist* technique for designing concurrent and durable data structures. We first argue that such data structures should be lock-free, then we detail the technique itself and how it can be used 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 [10, 13, 38, 40, 41, 43]. Nevertheless, our techniques also apply to other data structures.

The Case for Lock-Free Algorithms. As noted by previous work [7, 26, 29, 46, 47], lock-free algorithms are a good fit for the NVRAM environment. This is because in lock-free algorithms, threads must ensure that the data structure is in a consistent state at all times, so that the failure of any number of threads does not prevent remaining threads from making progress. A beneficial consequence is that, when used with NVRAM, lock-free algorithms ensure that as long as threads' stores are persisted in the order in which they are issued (we show how this can be relaxed), regardless of when 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, as opposed to transactional approaches which inherently require logging.

Our Technique. 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 new 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 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 that all direct dependencies of an operation be durably written before the operation is performed. Otherwise, a scenario in which the user receives a return value, the system restarts, and the stored data no longer reflects the state observed by the user is possible.

In order to deal with this issue, the most straightforward approach is what we call the *link-and-persist* operation. Essentially, when performing a link update that changes the state of a node, the link is atomically updated normally, but contains a mark to signal that 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 will try to do these steps itself. This method involves no blocking, and introduces bounded overhead, thus being suitable for all concurrent algorithm classes, including lock-free and wait-free algorithms [17].

We illustrate our technique through the example of a lock-free linked list that uses the algorithm proposed by Tim Harris [15]. In the original (volatile) 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 flag 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. Figure 1 shows the extra steps taken when inserting a new node using link-and-persist.

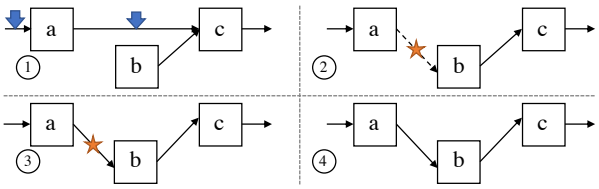

Figure 1: *Stages of inserting a node in a linked list using link-and-persist. 1.) The new node (b) is created and its predecessor's (a) adjacent links (indicated by downward arrows) are persisted. 2.) The predecessor node's next pointer is atomically made to point to the new node. A mark ($\star$) is added to the link to indicate it is not durable yet. 3.) The new node's incoming link is persisted. 4.) The mark is removed from the incoming link.*

Durable Implementations. We have used link-and-persist to implement several durable data structures: linked lists, hash tables, skip lists, and BSTs. These structures model the set abstraction, and have methods to insert, remove, and search for 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 [8]. Our linked list is based on Harris' algorithm [15], the hash table uses one Harris linked list per bucket, the skip list uses the lock-free algorithm by Herlihy et al. [19], while the BST uses the algorithm proposed by Natarajan and Mittal [45]. Other algorithms and data structures can be similarly modified.

Correctness. Our data structures are linearizable, since we start from linearizable algorithms and add only flushes, which do not impact linearization points. We also ensure two additional properties [12]. First, each update operation ensures that its changes are durable before returning. 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, 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) [8], 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.

4 Limiting Write-Backs: the Link Cache

We now introduce our second technique, aimed at further reducing the number of *sync* operations in durable data structures.

4.1 Link Cache Overview

As discussed in § 2, batching multiple cache line write-backs is significantly faster than persisting them one at a time. Therefore, we propose the following scheme: when doing an update, do not immediately persist links, but place them in a fast, *volatile* cache (the *link cache*), and write back all the links in this cache when an operation that directly depends on one of them occurs. Of course, this means that clients which have inserted links into the link cache can only consider the operation completed 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 (achievable 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, modified links 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 § 3). 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.

4.2 Link Cache Implementation

Our main aims for the link cache are small memory footprint, non-blocking operation, and fast insertions. With these requirements in mind, we chose to make insertions in the cache best effort. The cache is only useful if it can improve the time updates spend waiting. Therefore, if an

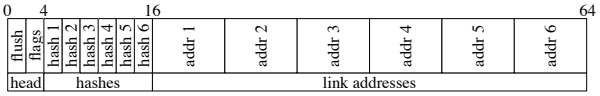

Figure 2: *A bucket in the link cache.*

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, link cache insertions have 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 2 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 have a hash space of size 2M. 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, the link cache is being flushed, 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 increments the corresponding flushing counter 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 thread then waits for the write-backs to complete by issuing a fence, decrements the flushing counter, and returns.

Scan. The scan operation is given a key and searches for any link pertaining to this key in the link cache. 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 link cache, 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.

Illustration: Link Cache Effectiveness. We illustrate the effectiveness of the approach using the same linked list example employed in the previous section. The schedule of operations in our example, as well as the way the link cache is constructed are presented in Figure 3. 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.

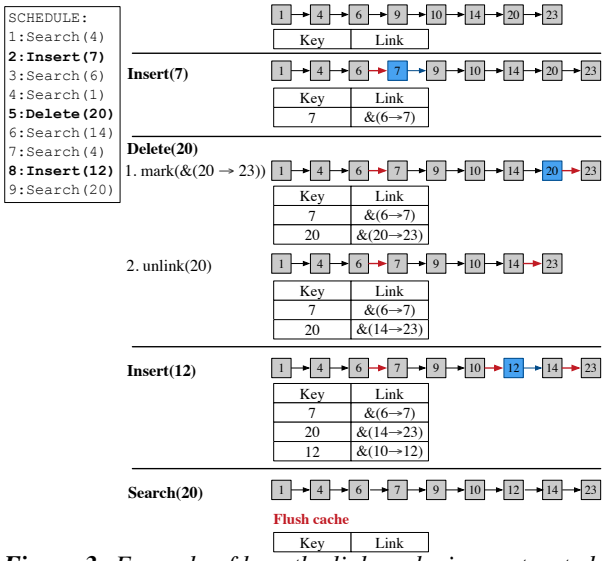

Figure 3: *Example of how the link cache is constructed.*

5 Memory Management with NV-Epochs

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

5.1 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 is first allocated and initialized, after which the node is 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 between these two main steps both in case of an insertion and a removal, a persistent memory leak occurs: we 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 coarser-grained active memory areas instead of keeping track of individual allocations/deallocations. Intuitively, when allocating, threads often reserve larger contiguous memory areas from which they serve user requests; thus, 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 to which no references are held. This reclamation step is only run periodically for performance considerations (typically, when a certain number of unlinked objects have been collected). 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: most of the time, the memory area will already be marked as active. We illustrate the potential benefits of the approach in Figure 4.

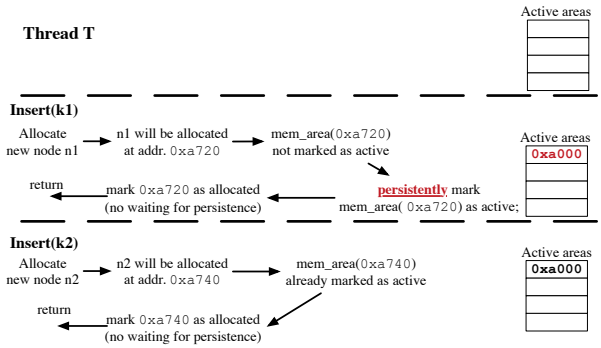

Figure 4: *Illustration: thread T performs two inserts. While both allocate memory, with our approach, only the first allocation performs a durable write.*

While allowing us 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 areas at the time of shutdown, 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 data structure nodes belong to memory areas which store no other type of data. 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 after a failure.

5.2 Epoch-Based Memory Reclamation

Epoch-based memory reclamation [11] 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 and in the middle of an operation. 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*.

5.3 Interface with NVRAM Allocators

Memory allocators usually reserve a large contiguous address space, which is then recursively split into smaller chunks. The chunk from which an object is allocated depends on the object's size. These chunks of contiguous memory are generally referred to as allocator *pages*. Since smaller pages from which 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 structures maintaining active pages, the allocator can recover its state in case of a restart. An existing persistent concurrent allocator can be used with our system, 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. We use a modified version of jemalloc [27], with write-backs inserted when updates to allocator metadata occur to model the run-time performance of persistent allocators.

5.4 Tracking Active Memory Areas

In our approach, each thread keeps a set of active memory pages in a structure called the *active page table (APT)*. 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 durably (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, 42, 51, 56]. In this instance, we used the allocator provided with `nvml` [24].

5.5 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 approaches to verifying this, both of which can be parallelized in order to decrease recovery time. The efficiency of each method 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 an address n is allocated, search the data structure for n's key. If (i) the search returns a result and (ii) the

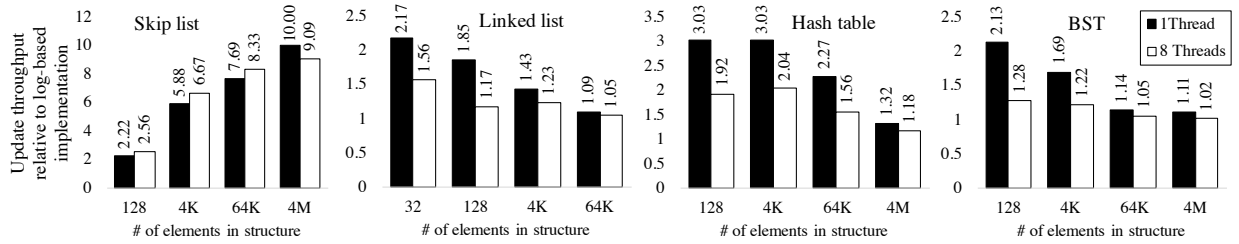

Figure 5: *Update throughput of data structures implemented using our techniques (1 and 8 threads). Values are normalized to throughput of redo log based implementations (1 and 8 threads, respectively).*

address of the returned node is the same as *n*, 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.

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.

6 Evaluation

We now study the impact our proposed techniques have in practice. We look at the overall performance improvements, as well as at the benefits of individual techniques.

6.1 Experimental Setup

We run experiments on an Intel Xeon machine having four E7-4830 v3 12-core processors operating at 2.1–2.7 GHz, with cache sizes of 32KB (L1), 256KB (L2), and 30MB (LLC, per die). We work with key-value pairs, both of which are 8B in size. Nodes are cache-aligned to 64B. Larger values can be accommodated by using indirection instead of directly storing values inside nodes. Shown 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 simulate `clwb` by writing data normally, and then pausing for an appropriate number of cycles, similar to previous work [3, 6, 33, 55, 56]. The number of cycles to pause is chosen so as to account for the increased latency of NVRAM (we assume an NVRAM write latency of 125ns, which is an average of the projected values). 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. In order to account for the benefit of flushing several cache lines at a time over flushing them one by one, we inject the artifical pause described above only once per batch of cache lines being written to NVRAM (e.g., only once per flush of the link cache).

6.2 Data Structure Performance

We 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 for 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 [16], a lock-based skip list by Herlihy et al. [18], bst-tk [8], and a hash table with a lazy linked list per bucket. We manually apply logging to each data structure, taking advantage of knowledge of the algorithms so as to minimize the number of *syncs* while maintaining correctness. We do this for fairness of comparison, as the alternative of using a generic transactional/logging framework would have likely resulted in more *syncs* and thus worse performance.

In Figure 5, 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

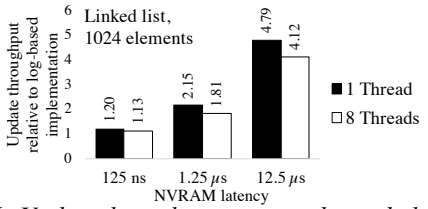

Figure 6: *Update throughput compared to redo log based implementation for various NVRAM write latencies.*

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 § 6.3. In addition, as the number of concurrent updating threads increases, the link cache becomes somewhat less efficient (also discussed in § 6.3). Thus, for high degrees of concurrency, we can turn the link cache off.

In our experiments, we use NVRAM latencies comparable to those of DRAM. Nevertheless, current technologies still have significantly larger latencies. We therefore also perform a simulation where we increase write latency (Figure 6). These measurements are representative of structures which are small enough for reads to be served from cache. As NVRAM write latency increases, our approach becomes much more effective: the ratio between our throughput and that of a log-based implementation becomes inversely proportional to the ratio between the number of *sync* operations in the two approaches.

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, these experiments show that our approach is beneficial for all the situations we have considered.

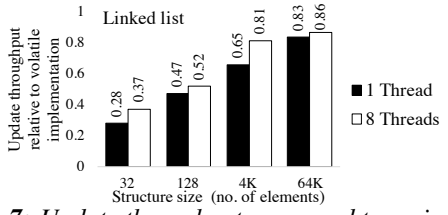

Figure 7: *Update throughput compared to an implementation oblivious of NVRAM.*

We also compare our approach to algorithms aimed at volatile memory, which do not concern themselves with data durability. Briefly, the per-operation overheads related to durability introduced by our approach are constant. While these might account for a non-negligible fraction of the operation latency for small structures, for larger structures, as total operation latency increases, these become less apparent. This is illustrated on a linked list in Figure 7. Thus, in terms of performance, our approach represents a middle ground between volatile data structures and log-based durable approaches.

6.3 Performance: a Closer Look

We now explore the impact each of our techniques has on overall performance.

Link-and-Persist and Link Cache. We evaluate the individual impact on performance of the link-and-persist technique and of the link cache. We measure the throughput of each data structure with the log-based implementation, with a log-free implementation that uses link-and-persist and with a log-free implementation that uses the link cache (all using identical memory management schemes). We then normalize the throughput of the log-free implementations with respect to the log-based implementation to determine the change in performance. We use an update-only workload, with 1024-element data structures. The link cache occupies 32 cache lines.

Figure 8 shows the results. As a result of removing logging, algorithms using link-and-persist outperform log-based alternatives for all structures, both in single-threaded and in concurrent scenarios. Moreover, in most cases, the link cache brings an additional increase in performance with respect to the link-and-persist implementation, due to its batching of write-backs.

NV-Epochs. We evaluate the efficiency of our active page table. The active page table is only efficient if it saves *sync* operations: i.e., if an important fraction of updates do not have to write active page table entries.

We consider 4KB memory pages, and we try to trim an active page table when it exceeds 16 elements. We measure the fraction of allocations and deallocations that

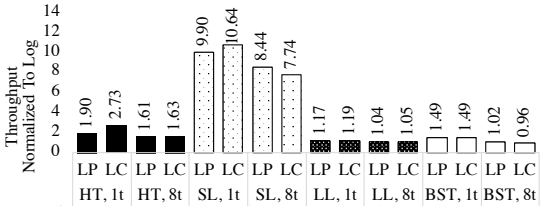

Figure 8: *Throughput of log-free implementations using link-and-persist (LP) and the link cache (LC), normalized to throughput of log-based implementations. Data structures are 1024-element hash table (HT), skip-list (SL), linked list (LL) and binary search tree (BST). Workloads are 100% updates, single-threaded (1t) or with 8 concurrent threads (8t).*

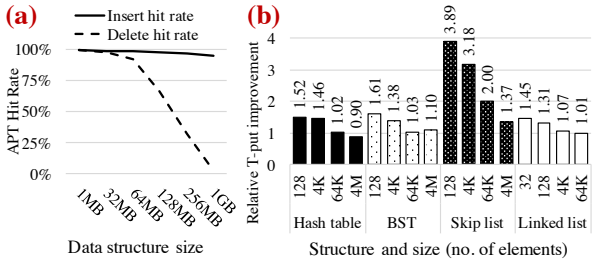

Figure 9: *Active page table hit rates and throughput improvements due to NV-epochs.*

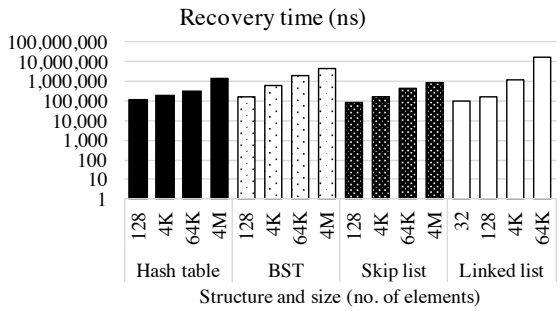

Figure 10: *Data structure recovery times.*

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 9.a. 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.

We note that the hit rate is close to 100% for allocations, regardless of structure size. In case of deallocations, the hit rate starts decreasing after the structure exceeds 64 MB (more than 1M nodes). 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, which have small access latencies. In such situations, our approach is effective for both types of operations.

This conclusion is reflected in the throughput observed when using NV-epochs (Figure 9.b): for small and medium-sized structures, NV-epochs can increase throughput several-fold. For large structures, when keeping track of memory at 4KB granularity, NV-epoch's effectiveness decreases. However, the granularity at which we keep track of active memory areas is adjustable. Larger memory areas result in higher hit rates and throughput improvements, at the expense of increased recovery time.

6.4 Recovery

We now measure the time it takes to recover a data structure. We simulate a crash by first stopping execution of the algorithm at an arbitrary point. Then, we ensure the structure's data is not in the write-back caches (by purging the caches). Next, we run the recovery process which first brings the data structure to a consistent state and then traverses its active pages to free allocated-but-not-reachable nodes[3]. We show recovery times for the various structures as a function of their size in Figure 10.

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 linked 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 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, search operations must traverse more pointers for larger structures, and data is less frequently present in the higher-level caches. We believe the observed recovery times are acceptable in case of a reboot.

[3] After recovery, new threads can spawn and resume execution at a "safe" point (a point in the instruction stream from which execution can continue regardless of when the crash occurred). Determining such safe points in general is outside the scope of this paper, but for our specific case, any point in-between two data structure operations is a safe point.

6.5 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 durable by replacing its core data structures—the hash table and the slab allocator—with durable versions. This transformation entails interesting technical challenges.

First, Memcached uses a lock-protected sequential hash table; thus replacing it with our durable 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 [8], and replacing CLHT with our log-free durable hash table.

The second challenge is related to the recovery of items. With a naive implementation of a durable 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 (§ 5): 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 and memcached-clht using

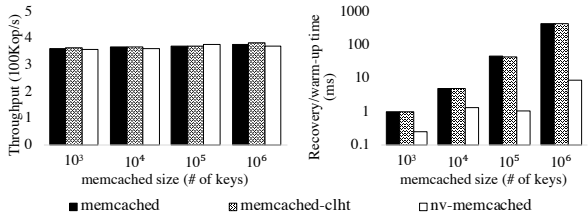

Figure 11: *Performance and warm-up time comparison of Memcached and NV-Memcached.*

memtier-benchmark [50]. 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. 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 the three systems for different key ranges, under a 1:4 `set` to `get` ratio. Figure 11 shows that there is no notable performance drop between Memcached, memcached-clht and *NV-Memcached*. Thus, 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 and memcached-clht (the time it takes to populate the cache with half of the key range) to the recovery time of *NV-Memcached* (the time it takes to recover after a restart). Figure 11 shows that populating a (volatile) Memcached or memcached-clht instance with items can take up to three 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 a means of interaction with NVRAM [2, 6, 9, 14, 25, 28, 30, 33, 37, 56]. The benefits of transaction-based approaches are generality and ease of use. Yet, their inherent and significant overhead has been recently highlighted (e.g., [52]), and several attempts to alleviate the problem have been proposed. Izraelevitz et al. [25] introduce an approach in which by reliably keeping track of the last executed store instruction at each thread, one is able to simply complete the execution of critical sections after a restart. While efficient if write-back caches are persistent, the approach otherwise requires a write to the log for each store in critical sections. Kolli et al. [30] focus on static transactions in lock-based applications, and attempt to minimize persist dependencies in order to limit waiting time. The authors also show how the commit stage of transactions can be performed while not holding any locks. Similarly, Kamino-Tx [37] 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 when locality is preserved, allows us to circumvent such logging altogether in the context of concurrent data structures.

A number of 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 when compared to our method, in particular due to logging.

Several proposals for indexing trees for NVRAM have been made [5, 32, 48, 54, 58]. 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 easily generalized to other data structures. Friedman et al. [12] introduce lock-free algorithms for durable queues, but do not go beyond this data structure, or consider memory management. Other work advocating lock-free algorithms either assumes durable caches [46, 47], or automatically generates durable algorithms that issue syncs after each update [26].

The problem of general memory allocation and reclamation for NVRAM has also received attention. Generic persistent memory frameworks [2, 6, 35, 56] handle allocation and reclamation as part of the transaction mechanisms they provide, and thus rely on logging. nvm_malloc [51] provides an interface to allocate and free persistent memory, but because of 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 at both run time and recovery time. Our approach in fact builds upon basic memory allocators, and handles concurrent memory reclamation as well. Thus, our scheme can take advantage of an efficient durable memory allocator at its core.

Other Memcached adaptations for NVRAM have been proposed, but they use transactions extensively [6, 36, 44] or they do not guarantee all completed requests are durable [57], whereas *NV-Memcached* ensures all completed requests are durable and limits transactions to the slab allocator code by using our log-free hash table.

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] Nachshon Cohen, Rachid Guerraoui, and Igor Zablotchi. The Inherent Cost of Remembering Consistently. SPAA 2018.

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

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

[10] Facebook. RocksDB. `http://rocksdb.org`.

[11] Keir Fraser. *Practical Lock-Freedom*. PhD thesis, University of Cambridge, 2004.

[12] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A Persistent Lock-Free Queue for Non-Volatile Memory. PPoPP 2018.

[13] Google. LevelDB. `http://leveldb.org`.

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

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

[16] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III, and Nir Shavit. A Lazy Concurrent List-Based Set Algorithm. OPODIS 2005.

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

[18] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A Simple Optimistic Skiplist Algorithm. SIROCCO 2007.

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

[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] jemalloc. `http://jemalloc.net/`.

[28] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. NVWAL: Exploiting NVRAM in Write-Ahead Logging. ASPLOS 2016.

[29] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. Language-Level Persistency. ISCA 2017.

[30] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M Chen, and Thomas F Wenisch. High-Performance Transactions for Persistent Memories. ASPLOS 2016.

[31] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting Phase Change Memory as a Scalable DRAM Alternative. ISCA 2009.

[32] 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.

[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, Achin Mishra, Amee Trivedi, Yihe Huang, Faisal Zaghloul, Sanidhya Kashyap, Margo Seltzer, Tim Harris, Steve Byan, Bill Bridge, and Dave Dice. Persistent Memory Transactions. *arXiv preprint arXiv:1804.00701*, 2018.

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

[37] 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.

[38] Memcached. `http://www.memcached.org`.

[39] Micron. 3D XPoint Technology. https://www.micron.com/about/our-innovation/3d-xpoint-technology.

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

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

[42] 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.

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

[44] 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.

[45] Aravind Natarajan and Neeraj Mittal. Fast Concurrent Lock-Free Binary Search Trees. PPoPP 2014.

[46] Faisal Nawab, Dhruva Chakrabarti, Terrence Kelly, and Charles B. Morrey. Zero-Overhead NVM Crash Resilience. NVMW 2015.

[47] Faisal Nawab, Dhruva R Chakrabarti, Terence Kelly, and Charles B Morrey III. Procrastination Beats Prevention: Timely Sufficient Persistence for Efficient Crash Resilience. Technical Report HPL-2014-70. Hewlett-Packard, 2014.

[48] 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.

[49] Moinuddin K Qureshi, Vijayalakshmi Srinivasan, and Jude A Rivers. Scalable High Performance Main Memory System Using Phase-Change Memory Technology.

[50] Redis Labs. NoSQL Redis and Memcache Traffic Generation and Benchmarking Tool. https://github.com/RedisLabs/memtier_benchmark.

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

[52] Seunghee Shin, James Tuck, and Yan Solihin. Hiding the Long Latency of Persist Barriers Using Speculative Execution. ISCA 2017.

[53] Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. The Missing Memristor Found. *Nature*, 453(7191):80, 2008.

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

[55] 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.

[56] Haris Volos, Andres Jaan Tack, and Michael M Swift. Mnemosyne: Lightweight Persistent Memory. ASPLOS 2011.

[57] 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.

[58] 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.

[59] Yiying Zhang and Steven Swanson. A Study of Application Performance with Non-Volatile Main Memory. MSSR 2015.