Persistent Memory and the Rise of Universal Constructions

Andreia Correia
University of Neuchatel
andreia.veiga@unine.ch

Pascal Felber
University of Neuchatel
pascal.felber@unine.ch

Pedro Ramalhete
Cisco Systems
pramalhe@gmail.com

Abstract

Non-Volatile Main Memory (NVMM) has brought forth the need for data structures that are not only concurrent but also resilient to non-corrupting failures. Until now, persistent transactional memory libraries (PTMs) have focused on providing correct recovery from non-corrupting failures without memory leaks. Most PTMs that provide concurrent access do so with blocking progress.

The main focus of this paper is to design practical PTMs with wait-free progress based on *universal constructions*. We first present CX-PUC, the first bounded wait-free persistent universal construction requiring no annotation of the underlying sequential data structure. CX-PUC is an adaptation to persistence of CX, a recently proposed universal construction. We next introduce CX-PTM, a PTM that achieves better throughput and supports transactions over multiple data structure instances, at the price of requiring annotation of the loads and stores in the data structure—as is commonplace in software transactional memory. Finally, we propose a new generic construction, Redo-PTM, based on a finite number of replicas and Herlihy's wait-free consensus, which uses physical instead of logical logging. By exploiting its capability of providing wait-free ACID transactions, we have used Redo-PTM to implement the world's first persistent key-value store with bounded wait-free progress.

ACM Reference Format:
Andreia Correia, Pascal Felber, and Pedro Ramalhete. 2020. Persistent Memory and the Rise of Universal Constructions. In *Fifteenth European Conference on Computer Systems (EuroSys '20), April 27–30, 2020, Heraklion, Greece.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3342195.3387515

EuroSys '20, April 27–30, 2020, Heraklion, Greece

ACM ISBN 978-1-4503-6882-7/20/04…$15.00
https://doi.org/10.1145/3342195.3387515

1 Introduction

Generic approaches to create concurrent data structures are very attractive because they are easy to deploy. At the same time, lock-free generic approaches are considered with some discredit, given that they serialize mutative operations. This view is changing as recent developments show encouraging advances in universal constructions (UC) and software transactional memory (STM) that can compete with custom solutions in terms of performance.

Concurrent data structures. A trivial solution for designing concurrent data structures is to create a universal construction that holds a mutual exclusion lock protecting all methods. Unfortunately, mutual exclusion locks are blocking, can cause deadlocks or livelocks, serialize all operations and at best provide *starvation-free* progress.

First proposed in 1990 by Herlihy [25], a wait-free universal construction (UC) takes the sequential implementation of a data structure and produces an equivalent concurrent data structure whose methods are linearizable and have wait-free progress. Thanks to lambdas and closures, available in most high-level languages (*e.g.*, C++, D, Java, Scala), using a UC becomes as simple as wrapping each method in a lambda and passing it to the UC for execution, typically via a call to `applyUpdate()` or `applyRead()` depending on whether the method is mutative or read-only. Similarly to locks and STMs, the code inside the lambda can be reasoned about in a sequential way. Unlike STMs, no annotation is required and the allocator calls remain unchanged. Although compiler instrumentation can reduce the need for user annotation when using STMs. A single constraint sets wait-free UCs apart from locks: the fact that the sequential code may be internally executed multiple times.

Two years after UCs were introduced, Herlihy and Moss proposed the concept of *transactional memory* [26] and, later, Shavit and Touitou presented a generic software-only TM (STM) [42]. Previous work has also shown that lock-free [18, 19] and even wait-free [40] STMs are possible. Depending on implementation details, language choice and compiler support, creating a concurrent data structure with an STM requires annotation of the types in the data structure, possibly annotation of the loads and stores executed in each method, as well as the replacement of allocation and deallocation calls with equivalent methods provided by the STM. The user can reason about the resulting code as if it were sequential.

Handmade lock-free data structures have been proposed in the literature since at least 1986 [45]. Design-wise, lock-free data structures represent an almost opposite approach to STMs and UCs: their implementation is optimized for a specific data structure, however their algorithms have a highly non-sequential reasoning. Significant expertise is required to design and implement a correct lock-free data structure, and even more when lock-free memory reclamation is deployed. Modifying a single line of code in lock-free data structures while maintaining correctness can be a challenging task, making their design and implementation inaccessible to most practitioners.

Interestingly, in scenarios where durability is a requirement, research work has focused on generic approaches (STMs [13, 16, 17, 40] and UCs [9]) and only recently the first persistent lock-free queue was shown [20]. This shift toward generic approaches can be attributed to two facts. First, applications that need to persist data are likely to have several persistent data structure instances and likely require consistent transactions between them. The second reason is related to memory allocation and deallocation, which can no longer rely on the system allocator. Allocations in persistent memory are restricted to the specified memory region and the operations of the allocator have to be resilient to failures [5]. The inherent requirement of resilience for data persistence is almost begging for transactions and this property can hardly be provided by specific (handmade) persistent data structures. It is therefore crucial to further research and develop *practical* universal constructions that can achieve high throughput for commonly used data structures.

Non-volatile memory. The introduction of byte-addressable Non-Volatile Main Memory (NVMM) technology has added a new layer to the design of concurrent data structures. In NVMM, concurrent data structures must also provide *failure resilience*, correctly recovering to a consistent state in the event of a non-corrupting failure [27]. We distinguish from corrupting and non-corrupting failures: corrupting failures cause damage to data required for recovery, which prevents restoring a consistent state, whereas non-corrupting failures allow for the recovery procedure to complete. To help researchers reason about these properties, recent work has proposed new consistency models similar to linearizability but with ACID properties, such as *durable linearizability* [28] where the effects of a transaction become both durable and visible to other threads. The same paper describes a *recipe* to transform concurrent algorithms designed for transient memory into failure-resilient algorithms for NVMM. Roughly speaking, this recipe implies the issuing of a persistence flush `pwb` and persistence fence `pfence` instruction for every store release or compare-and-swap (CAS) operation in the data structure's algorithm.

Recent x86 CPUs have introduced the `CLWB` instruction to implement `pwb`, with the `SFENCE` implementing `pfence`. These optimized `pwb` instructions have significantly lower

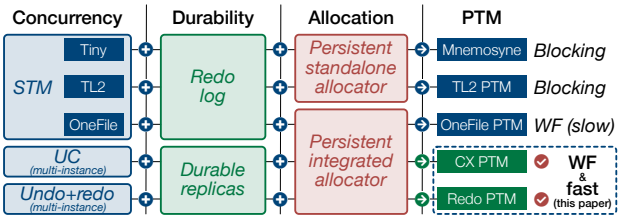

Figure 1. Illustration of the design space for PTMs. Most existing solutions (Mnemosyne [46], TL2 PTM [22, 35]) are blocking, while those with wait-free progress (OneFile [40]) are slow. We introduce in this paper two novel PTMs that are both efficient and wait-free (CX-PTM and Redo-PTM).

cost compared to a `pfence`. Handmade lock-free data structures typically rely on multiple CAS instructions and adapting them to NVMM using the recipe from [28] incurs an added cost of a `pwb` and a `pfence` for every CAS. This high number of `pfence` adds significant performance penalty and is a major bottleneck to achieving high throughput [29].

Generic techniques for NVMM have shown it is possible to execute fast transactions, using just four `pfence` instructions [10], or two [40, 46] and even a single [9] fence. In contrast, the highly optimized NVMM lock-free queue by Friedman et al. [20] requires 2 `pfence` instructions for enqueing and 4 for dequeueing, not accounting for the number of fences needed for allocation, deallocation or tracking of the queue's internal nodes. These recent developments have significantly raised the bar on what handmade data structures must be capable of for preserving their supremacy in production code and, for the first time, generic techniques are moving to a privileged position.

Generic constructions. Most wait-free UCs are based on one of two approaches, either copy-on-write (CoW), or adding the operations to a queue (or list). On CoW-based UCs [15], each mutation or group of mutations implies that a full copy of the data structure must be made. This is inefficient for large objects when converted to a persistent universal construction (PUC), due to the high number of `pwb` operations that must be executed for each cache line of the new object. On the other hand, queue-based UCs must keep a local replica (instance) of the data structure for each thread and, therefore, every mutative operation must be replicated on every instance, along with the respective `pwbs`, implying that the number of `pwbs` is $O(N_{threads} \times N_{stores})$. Previous work has shown that efficient (blocking) persistent transactional memory (PTM) [10] reduces the number of `pwbs` to at most $2 \times N_{stores}$.

Software transactional memory (STM) is another generic approach that commonly requires store and load interposing. Several existing PTMs rely on an STM to provide transactions with concurrent access (see Figure 1). For instance, the Mnemosyne [46] PTM is based on TinySTM [16, 17], though with a significant engineering effort added to it. TL2 [13] was also effectively transformed into a PTM [22].

Towards practical wait-free PTMs. In this paper we focus our research on PTMs with wait-free progress, as illustrated in Figure 1. Our work is based in a recently developed universal construction, called CX [12], that has shown significant performance improvement when compared with other universal construction and custom concurrent data structures. The PTMs we present in this paper are software-only solutions that work on commodity hardware. Our persistency model assumes both caches and registers are volatile, and that 64 bit writes to persistent memory are atomic.

In short, we make the following contributions. We first introduce CX-PUC, the first persistent universal construction with bounded wait-free progress. CX-PUC is a universal construction in the sense that it is capable of transforming any sequential implementation of an object into an equivalent concurrent durable linearizable object with wait-free methods, without requiring any annotation to the sequential implementation. We then propose CX-PTM, a PTM whose transactions are durable linearizable and have bounded wait-free progress. It requires, however, annotation of the sequential implementation. We finally present Redo-PTM, a new construction that is based on a finite number of replicas, like CX, but takes a different approach by relying on a volatile transaction log recording every store that took place during its execution. All three constructions provide correct wait-free failure-resilient memory (de)allocation, with a linear bound on memory usage. Furthermore, their operations or transactions execute with just two persistence fences (one `pfence` and one `psync` [27]). Both CX-PTM and Redo-PTM are capable of providing multi-step ACID transactions between several data structures or objects. We conducted in-depth evaluation of all three variants and compared them with other state-of-the-art techniques. For validation in realistic settings, we used Redo-PTM to develop an in-memory database with wait-free durable linearizable transactions that implements the LevelDB [21] and RocksDB [14] API. Using RocksDB benchmarks, we compared it with RedoDB, both deployed in NVMM, where RedoDB achieves superior performance in all tested workloads.

The rest of the paper is organized as follows. We discuss related work in §2 and briefly introduce in §3 the design of our algorithms. In §4, we present two algorithms, CX-PUC and CX-PTM, that extend the CX universal construction with support for persistence. We then present a new generic construction, Redo-PTM, in §5. We provide an in-depth evaluation of all approaches in §6 and finally conclude in §7.

2 Related Work

In this section, we give a short overview of some of the generic techniques for NVMM proposed in the literature, while focusing our attention on PTM libraries whose properties provide at least some kind of non-blocking progress and durable linearizability [28] consistency for their transactions.

Undo-log and redo-log are well known techniques for durable transactions, dating back to at least ARIES [37]. In *undo-log*, the original data is saved in the log and in the event of a non-corrupting failure, the modifications are reverted back to their original state. In the classic *redo-log* technique, 4 persistence fences are executed per transaction, regardless of the number of modified memory locations within the transaction. Modifications to each memory location during a transaction are added to a log, leaving the original data unchanged. In turn, every memory load requires a search for the last modification on the log. The number of entries in the write-set is kept in a volatile variable until commit time, and only made persistent at the end of the transaction. Recovery consists in (re-)applying the modifications stored in the persistent write-set, if its size is non-zero.

Mnemosyne [46] combines a persistent redo log with a lock based STM, TinySTM [16, 17]. In contrast, NV-Heaps [8] uses an undo log system and an object-based TM. Atlas [7] uses an undo log and transforms lock-critical sections into FASEs (failure-atomic-sections). This automatic transformation is not universally applicable because there are scenarios where lock-critical sections do not translate into FASEs that maintain program invariants. Operations in Atlas are *buffered durable linearizable* [28]. Blurred persistence (BPPM) [33] is similar to Mnemosyne [46] but does not provide durable linearizable transactions. DudeTM [32] uses shadow DRAM to provide efficient transactions, by decoupling the durability from the concurrent transaction, but its transactions are not durable linearizable either. Kamino-Tx [36] attempts to eliminate copying in the critical path by maintaining an additional replica (backup) of all the data and using that replica to recover from failures and aborts. All successful transactions are asynchronously applied to the backup once they are committed to the main replica. Copying of data to and from the backup is performed asynchronously, meaning that transactions are buffered durable linearizable. SoftWrAP [23] effectively decouples value communication for transaction execution from persistent memory logging for recovery. The record of updates is streamed to a log area in persistent memory asynchronously and concurrently in the form of log records. SoftWrAP also provides buffered durable linearizability. PMDK [44] is a PTM developed by Intel with undo log and an in-house efficient memory allocator. Concurrency in PMDK is left up to the user, typically with fine grained locking. All the aforementioned PTM libraries have *blocking transactions*.

Another class of techniques require hardware changes to current commodity servers: PHTM [2], DP2 [43], ThyNVM [41], JustDo [27], PHyTM [6], ATOM [31], FWB [38] and DTHM [30]. We refer to Joshi et al. [30, Table 1], and [10] for summarized comparisons of the various techniques on persistent memory.

Romulus [10] is a PTM whose dynamic transactions are durable linearizable. Romulus maintains two data replicas

in NVMM, guaranteeing that at least one of them is always consistent, and recovers from the consistent replica in the event of a failure. RomulusLR is a variant of Romulus with wait-free progress for read-only transactions and blocking but starvation-free progress for the update transactions.

ONLL [9] is a generic technique for NVMM where both read-only and update operations are lock-free and durable linearizable. In ONLL, the read-only transactions do not execute a persistence fence. The ONLL algorithm relies on two basic building blocks: a single-fence *persistent* log and a lock-free *volatile* queue. Each thread has its own *volatile* instance of the underlying object, implying that ONLL does not require interposing of stores nor loads. However, the input parameters passed to the update operations must be flushed, likely requiring some kind of annotation. So far, no programming language provides support for function code to be copied to persistent memory. Because of this limitation, ONLL does not provide dynamic transactions, any function called inside a transaction must have been previously encoded to a unique number. ONLL belongs to a class of algorithms that do persistent *logical* logging, where the log contains the operations, as opposed to *physical* logging, where the log describes the addresses in memory where modifications have occurred.

OneFile [40] is a wait-free PTM with durable linearizable dynamic transactions, accepting any deterministic code that can be re-executed multiple times always yielding the same outcome. OneFile uses a redo log for durability and a time-based concurrency control where mutative transactions are serialized but may run concurrently with disjoint read-only transactions. Every store in the sequential data structure requires one double-word CAS to PM.

The following table summarizes the properties of the main PTMs discussed in this paper. The log type can be persistent (p) or volatile (v). N represents the number of concurrently executing threads. R represents the number of modified contiguous ranges of data.

	Type of log	Progress	`pfence`	#Replicas
PMDK [44]	p - physical	blocking	2+2R	1
ONLL [9]	p - logical	lock-free	1	N
OneFile [40]	p - physical	wait-free	2	1
CX-PUC	v - logical	wait-free	2	2N
CX-PTM	v - logical	wait-free	2	2N
Redo-PTM	v - physical	wait-free	2	N+1

3 Design Overview

A persistent universal construct (PUC) can be obtained by making all components of a UC persistent. However, doing so can have a significantly negative impact on performance. Therefore, it is crucial to identify which components of the universal construct are required to be persistent and still be resilient to non-corrupting failures. As a rule of thumb, the more UC data is kept in volatile (transient) memory the higher the overall throughput.

We initially followed this approach by making the CX library [12] persistent, as described in §4. In this process, the most important design choice is to have the queue of operations in volatile memory. Unlike the ONLL PUC, whose list of operations must be persisted, CX does not need to persist the arguments of the functions nor the operations themselves. This allows CX to support dynamic transactions, thus making it easier to deploy and simpler to use. CX has a pointer to the latest up-to-date data replica, named `curComb`. The queue of operations can be volatile because all completed operations have their effects visible on the instance to which `curComb` points. Upon failure, only operations that are not yet completed can be lost. This behavior is durable linearizable because the operation is only considered completed when the transaction ends, if a crash occurs before the transaction ends, there is no guarantee the operation effects were actually persisted. CX and ONLL use a queue of operations (*logical logging*) where threads enqueue their operation. This queue of operations establishes the linearizable history, implying operations must be executed in that order and a thread may require to execute all operations preceding its operation in the queue. This can be problematic for long-lived transactions, such as inserting a node on a large linked list, as all threads will spend the majority of their time traversing the list repeatedly, even though the outcome of each operation is just a few modified memory locations.

We therefore developed Redo-PTM to address this problem, as described in §5. In Redo-PTM the first thread to execute the operation will save a *physical log* of the modified memory locations, allowing other threads attempting to execute the same operation to apply those modifications without having to execute the operation, thus avoiding the traversal of the list in the previous example. Redo-PTM shares little of the initial CX construction, apart from the usage of a fixed number of `Combined` instances.

4 CX-based PTMs

We introduce in this section two PTM variants that are based on the CX universal construction. We first summarize briefly the principle of CX before describing how we extend it to support persistence.

CX universal construction. In CX, a fixed number of replicas of the underlying data structure is required. If we consider a sequential history H that represents the most recent state of a data structure, then each replica is a data structure which contains all the effects of a sub-history of H. For CX to have wait-free progress, the necessary number of copies is 2 N, assuming each of the N threads can execute either read-only or update operations. At a given moment in time, each replica can have only one thread with write access, although read-only access can be shared by multiple threads. Concurrent access to each replica is managed by a

strong try reader-writer lock [11]. This lock has the uncommon properties of executing its methods in a finite number of steps and guaranteeing deadlock-freedom. These two properties are fundamental to providing wait-free progress for the universal construction.

The CX universal construction has two different methods for processing read and update operations. The read operations do not perform any mutation to the data structure. They only require access to a replica with the most recent state (called a "combined" instance in CX terminology as it aggregates several attributes associated with a replica), referenced by `curComb`. This is achieved by guaranteeing that an update operation only returns after `curComb` references a replica containing its effects and that this replica can be accessed in read-only mode. Concurrent threads performing update operations must agree by *consensus* on a global order of execution. The consensus mechanism establishes the sequential history H. For the CX-PUC and CX-PTM implementations, we chose the same consensus mechanism as the original CX algorithm, based on a *turn queue* [39]. The mutations queue establishes the order on which the update operations must be applied to each replica of the data structure.

The only objects in CX that need a memory reclamation technique, are the nodes of the queue. If a node is disposed of (or equivalently "self-linked") during memory reclamation, this can invalidate a replica and force a future updater thread to make a new copy from `curComb`.

The CX construction is composed of:

- `curComb`: a pointer to the most up-to-date `Combined` instance;
- `tail`: a pointer to the last enqueued node of the mutations queue; and
- `combs`: an array of `Combined` instances.

In turn, a `Combined` instance consists of:

- `head`: a pointer to a `Node` on the queue of mutations;
- `obj`: a replica of the data structure or object that is up to date until `head`, *i.e.*, any mutative operation that was enqueued after `head` has not yet been applied to `obj`; and
- `rwlock`: an instance of a reader-writer lock that protects the content of the `Combined` instance.

Finally, a `Node` holds:

- `mutation`: a function to be applied on the object;
- `result`: the value returned by the function, if any;
- `next`: a pointer to the next `Node` in the mutation queue;
- `ticket`: a sequence number to simplify the validation in case a thread's mutation is already applied to the object;
- `refcnt`: a reference counter for memory reclamation, as well as some other fields for internal use.

The CX algorithm has different code paths for read and update operations. An updater thread will call `applyUpdate()`, for which the main steps are:

1. Create a new `Node` with the desired mutation and insert it in the queue.
2. Acquire an exclusive lock on one of the `Combined` instances in the `combs[]` array, by attempting sequentially to acquire the lock from combs[0] to combs[$2N$].
3. Verify if there is a valid replica of the data structure in the `Combined` instance and make a copy if necessary.
4. Apply all mutations starting at `head` of the `Combined` instance until reaching the `Node` inserted in the first step, and update `head` to point to this node.
5. Downgrade the exclusive lock acquired in (2).
6. Compare-and-set (CAS) `curComb` from its current value to the just updated `Combined` instance. Upon failure, retry CAS until successful or until `head` of the current `curComb` instance is *after* the newly inserted `Node` in the queue.

A reader thread will call `applyRead()`, where it iterates at most *MAX_READ_TRIES* + N times. *MAX_READ_TRIES* is the maximum number of attempts by a reader thread, to acquire a shared lock on `curComb`, without enqueueing its read operation in the mutations queue. For each iteration the steps are:

1. Atomic load of `curComb` to a local variable `comb`.
2. At iteration *MAX_READ_TRIES*, create a `Node`, referred to as `myNode`, with the desired read operation and insert it in the queue.
3. Try to acquire a shared lock on `comb.rwLock`. If successful, execute the read operation on `comb.obj` and release the lock. Otherwise, execute another iteration starting at (1).

Once `myNode` is added to the queue, it is guaranteed that after N iterations the result will be available in `myNode.result`, where the read operation is executed by an updater thread. This guarantee follows from the CX algorithm and because the reader thread will only fail to acquire the shared lock if `curComb` transitions to a new `Combined`.

Consider Figure 2 to better understand how mutations are propagated to the available replicas, each replica being a sequential stack. Mutative operations in the wait-free queue are represented by rounded rectangles. The stack stores its elements in a linked list of nodes (circles), with dashed lines indicating the nodes that are not yet added to the specific instance of the data structure. Figure 2 presents a scenario where elements `a`, `b` and `c` were already added to the stack. `curComb` points to `combs[1]`, which holds an up-to-date stack (with all 3 elements inserted) protected by a shared lock (`RLOCK`). Three concurrent writers W_1, W_2 and W_3 are executing `push(d)` on `combs[2]`, `push(e)` on `combs[0]` and `pop()` on `combs[3]`, respectively. The order of the concurrent operations is determined by their position in the wait-free queue, i.e., `d` will be inserted first, after it will be removed from the stack and finally `e` is inserted. All writers will apply the operations on their `Combined` instances in this order. As can be observed, each writer updates his replica c, previously protected with an exclusive lock, up to its own operation,

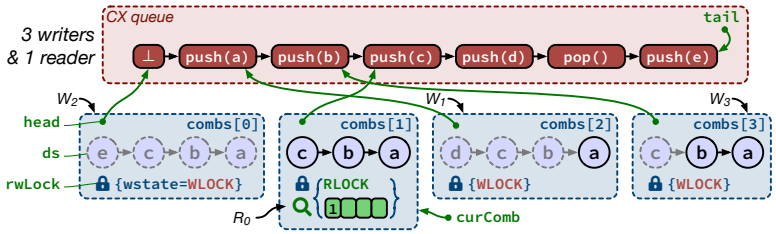

Figure 2. Illustration of CX's principle in a scenario with three writers and one reader are pushing and popping elements to a shared stack.

starting on the last applied operation `c.head`. Then each writer will try to transition `curComb` to its replica using a CAS. Meanwhile, a reader thread wants to execute `peek()` on `combs[1]`, the most up-to-date replica. The reader tries to acquire the shared lock by arriving at the read indicator, thus insuring `combs[1]` is protected for read-only execution.

Adding persistence to CX. Having described the main logic and components of the CX universal construction, we must now devise an approach for correctly persisting it. It is clear that, if all the components were made persistent following the recipe described by Izraelevitz et al. [27], then the resulting algorithm would be a correct persistent universal construction. However, the amount of persistent fences introduced by [27] would be detrimental to the overall performance.

To have a correct persistent algorithm that is durable linearizable, it is necessary to guarantee that all mutations that become visible to other threads are available and in a consistent state after recovering from a non-corrupting failure. The description of CX states that, an update operation only returns after `curComb` references a replica that contains its effects. In addition, a read operation always executes on the replica referenced by `curComb`. Therefore, if that replica is guaranteed to be persisted in a consistent state, then upon a non-corrupting failure, the most recent state of the data structure is available at restart for the user-application to continue execution. This implies that CX can be made durable linearizable if the variable `curComb` is persisted together with the replica that it references. All the other components, the queue of mutations and the array of `Combined` instances, can be kept in volatile memory, avoiding issues like persisting the mutation function.

On the other hand, because any of the $2N$ instances of `Combined` can be referenced by `curComb`, all `Combined.obj` may have to be persisted. This means that modifications to any of these instances must be flushed with corresponding calls to `pwb` before attempting to transition `curComb`. Regarding safe memory allocation and deallocation, the only objects dynamically allocated in volatile memory are the nodes of the wait-free queue, which are tracked with a wait-free implementation of hazard pointers with reference counting, similar to CX. As for allocations made in the user-code, we use a dedicated sequential persistent memory allocator to place objects within the appropriate NVMM region.

CX-PUC. Our first persistent implementation, CX-PUC (for "CX persistent universal construction"), is intended to be used with any sequential implementation of an object and requires no interposing of the data accesses, neither loads nor stores. However, to guarantee correct durability, methods of this object should not allocate objects in the transient heap using the system allocator and should instead use the persistent allocator provided by CX-PUC. Apart from this requirement, no other modification to the sequential implementation is necessary. This is useful for legacy code, where adding instrumentation can be a lengthy and complex task, specially when it comes to load interposition. Due to the lack of store interposing, there is no way to know what memory locations were modified and, therefore, determine which cache lines need to be flushed to persistence. The approach we followed was hence to flush the memory region in which the object has been allocated. As we will see later in §6, this approach is efficient enough for small objects.

CX-PTM. The persistent object annotation allows tracking of the memory locations modified during the transaction, issuing a persistence flush for each mutated cache line. This annotation requirement is valid for both the object and the persistent memory allocator. It represents a constraint for the user but can help improve the algorithm's performance.

In our second implementation, CX-PTM (for "CX persistent transactional memory"), all $2N$ regions contain pointers that refer to the first persistent region, named `MAIN` region. Furthermore, the algorithm uses a thread-local variable indicating the start of each region, `tl_start`, which is zero for the `MAIN` region.

When a replica is invalidated due to memory reclamation of the queue's nodes, a copy is made from the most up-to-date replica. The copy procedure is a byte per byte copy from the region referenced by `curComb`. This approach requires that all pointers reference the `MAIN` region, the same solution used by RomulusLR [10] but with multiple regions instead of two. CX-PTM requires pointer offset adjustment on load interposition. This is not the case for CX-PUC, which does in-place loads. CX-PTM and CX-PUC rely on logical logging, in section 5 we present PTMs that profit from physical logging.

5 Redo-PTM Construction

The CX universal construction was developed to serve as an abstraction for the user to manage wait-free concurrent access to an object. Such abstraction can be redesigned to take advantage of enforced object annotation. CX's original design [12] is based on a wait-free queue that establishes the partial subhistory of the update operations. These operations are stored in the queue as logical functions, which are applied to each replica in the sequential order defined by the queue. Re-execution of the same logical function for every replica results in a performance overhead for subsequent update operations. Future update operations that execute in a different replica do not take advantage of a previous execution of that logical function, representing a waste of resources as its re-execution will necessarily lead to the same mutative effects. In theory, it is more efficient to use a log that contains all mutative effects of the logical function, instead of logging the function itself. Indeed, the latter approach requires the function execution before it is published in the log. Reversing the serialization on the queue with the function execution, implies a significant change in the algorithm.

Overview of the algorithm. Redo-PTM relies on Herlihy's combining consensus [25] together with CX multi-instance based approach. The combining consensus is a wait-free consensus where all threads must announce its operation in a reserved position of a global array, and make a local copy of the *consensus object*. The consensus object contains an array of *applied* bits which indicates whether or not each thread's operation has been executed. After announcing its operation, a thread will traverse the array, gathering all the announced operations and updating its local consensus object to include those operations. Finally, it will try to make its local consensus object the new consensus using a compare-and-set (CAS). After at most two unsuccessful CAS attempts, the thread's announced operation is guaranteed to have been executed by another thread.

Redo-PTM pseudo-code is presented in Algorithm 1, 2 and 3.[1] Like P-Sim [15], which also uses Herlihy's consensus, Redo-PTM needs a `State` instance (the consensus object) in addition to the `Combined` instance. The `State` encapsulates all variable changes associated with a `curComb` transition, where `curComb` always references the latest state of the object. Unlike P-Sim, Redo-PTM requires a log of mutations so as to update each replica to its most recent state, since a copy of the object is only executed if the object is in invalid state. A replica reaches an invalidated state when it is no longer possible to apply the physical log in the sequential order established by the mutation queue, starting at `_c.head` until `curComb.head`, where `_c` represents the `Combined` instance of the replica (see Algorithm 3). A replica invalidation can occur when a `State` instance is reused. In the event of a replica invalidation, the thread will attempt to perform a copy from the replica referenced by `curComb` following an optimistic approach. To perform a successful copy, `curComb` must not change during the procedure. After two copy attempts, it is guaranteed by the combining consensus that the thread's operation was executed by a helper thread, thus ensuring wait-free progress. Compared with the CX universal construction, this optimistic approach allows a reduction on memory usage, requiring a maximum of $N + 1$ replicas. Redo-PTM lets each updater thread acquire the exclusive lock on one of the $N + 1$ instances of `Combined` and execute the requested published operations, following the combining consensus, on the object under protected access. It will then attempt to transition `curComb` to its `Combined` instance. If successful, this means that the sequential order of the operations was established and is visible to all threads. The `State` instance is associated with the `Combined` instance and contains the physical effects of the just committed transition. Afterwards, it is still necessary to append those physical effects to the mutations queue. This step will allow the eventual update of other replicas.

The main steps of `applyUpdate()` are as follows (see Algorithm 3, steps indicated as comments):

1. Publish mutation in the `req` and `announce` arrays.
2. Read `curComb` into a local variable `_curC`.
3. Get a new `State` object, referred as `_newSt`, and populate its attributes with the same data present on state `curComb.head`, where the array `applied` and `results` are kept.
4. Validate if tail of mutation queue is the same `State` referenced by `curComb.head` and, if different, help append it to the mutation queue.

Algorithm 1 Redo-PTM internal (volatile) data structures

```
1  class Combined {
2    atomic<SeqTidIdx> head;                    // 64 bit type
3    uint8_t* root;                  // Address where region starts
4    StrongTryRWRI rwLock;
5  }

7  class State {
8    atomic<SeqTidIdx> ticket;          // Sequence|thread Id|Index
9    atomic<bool> applied[N];
10   atomic<uint64_t> results[N];
11   WriteSetNode logHead;
12   WriteSetNode* logTail;
13   atomic<uint64_t> logSize;
14 }

16 class WriteSetNode {
17   WriteSetEntry log[MAXLOGSIZE];
18   WriteSetNode* next;
19   WriteSetNode* prev;
20 };

22 class WriteSetEntry {
23   uint8_t* addr;
24   uint64_t oldval;                          // Previous value
25   uint64_t val;                  // Desired value to change to
26 };
```

[1] Note that, for conciseness and clarity, we denote local variables by prefixing them with an underscore symbol (_).

5. Acquire an exclusive lock on one of the Combined instances in the combs[] array, referred as _c.
6. Apply the physical log to _c.0, starting at _c.head until the queue tail, where _c.0 represents the replica.
7. Simulate in _c.0 all announced mutations.
8. Downgrade exclusive lock acquired in (4).
9. CAS curComb from _curC to the just updated Combined instance _c. If successful, append _c.head to the tail of the mutation queue. Otherwise apply the undo log reverting the simulated mutations in (7) by setting oldVal for each address of _newSt, and re-execute from (2) one extra time.

Concerning a read-only operation, it will instead use the applyRead() procedure, which attempts MAX_READ_TRIES to acquire a shared lock in the Combined instance referenced by curComb before publishing the read operation in req. After the read operation is published in the req array, it will be executed by an updater thread in at most two attempts. The bounded limit of MAX_READ_TRIES+2 is guaranteed because the reader thread only fails to acquire the shared lock if curComb advances. Also, Herlihy's combining consensus guarantees that after an operation is published, it will be processed and their effects visible if curComb advances twice.

The main steps for each of the MAX_READ_TRIES+2 iterations of applyRead() for Redo-PTM are (Algorithm 2):

1. At iteration MAX_READ_TRIES, publish read operation in the req and announce arrays.
2. At iteration MAX_READ_TRIES+2, return the result available in the State referenced by curComb.
3. Read curComb into a local variable _curC.
4. Try to acquire a shared lock on _curC. If successful and the Combined instance is up to date, execute the read operation on _curC.obj and release the shared lock. Otherwise, execute another iteration starting at (1).

Illustration. Figure 3 illustrates the principle of the new Redo-PTM synchronization mechanism. In the chosen scenario two concurrent writers W_1 and W_2 respectively wish to execute operation push(d) and push(e) on a stack. At the start of their execution the stack already contains elements *a*, *b* and *c*.

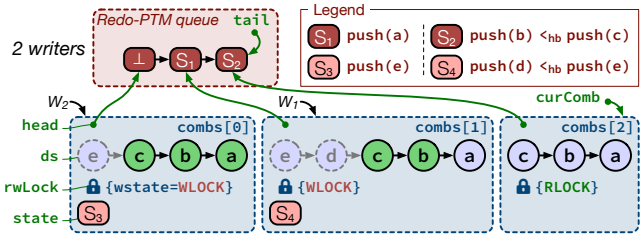

Figure 3. Illustration of Redo-PTM's principle with two writers pushing elements a through e in a shared stack.

A first transition of curComb resulted in the insertion of element *a* and the corresponding S_1 state appended to Redo-PTM queue. The second transition of curComb added both

Algorithm 2 Redo-PTM applyRead() pseudo-code

```
1  function APPLYREAD(readFunc, tid):
2      for _i ← 0, MAX_READ_TRIES + 1 do
3          if _i = MAX_READ_TRIES then                          {1}
4              req[tid] ← readFunc
5              announce[tid] ← ¬ announce[tid]
6          _curC ← curComb                                      {3}
7          _comb ← combs[_curC.idx]
8          if _comb.rwlock.sharedTryLock(tid) then              {4}
9              _tail ← _comb.head
10             if _curC = curComb then
11                 _ret ← readFunc()                            ▷ Function call
12                 _comb.rwlock.sharedUnlock(tid)
13                 _ringtail ← ring[_tail.seq%RSIZE]
14                 if _ringtail.seq < _tail.seq then
15                     pwb(curComb)
16                     psync()
17                 return _ret
18     end for
19     _curC ← curComb                                          {2}
20     pwb(curComb)
21     psync()
22     _comb ← combs[_curC.idx]
23     _tail ← _comb.head
24     if _tail.seq ≠ _comb.seq then
25         _tail = ring[_comb.seq%RSIZE]
26     _tailState ← stMatrix[_tail.tid].states[_tail.idx]
27     return _tailState.results[tid]
28 end function
```

b and *c* elements to the stack and a second state S_2 was appended to the queue, where S_2 contains a physical log of all the modifications associated with executing push(b) followed by push(c). As a consequence curComb transitioned from combs[0] to combs[1] and finally to combs[2]. The *ds* stack present in combs[0] is empty, with combs[0].head referencing the starting sentinel node of the queue. After the first transition, element *a* was added to the stack present in combs[1]. Finally in combs[2] all elements currently added to the stack are present and this Combined instance is referenced by curComb, which always references the most up to date replica. Each concurrent writer requires a Combined instance protected by an exclusive lock to execute in isolation the batch of operations published in the req array. Because both Combined instances, combs[0] and combs[1], are not up to date, each thread will apply the physical log recorded on the states after head. From Figure 3 we can conclude that the writer W_2 has traversed the announce array before the writer W_1 published its operation, because its state and simulated object only contains the effects of executing push(e). On the other hand, when writer W_1 traversed the announce array, it found two operations published and its simulation generated state S_4, which contains the effects of executing push(d) followed by push(e). At this point W_1 and W_2 are competing to transition curComb to its Combined instance, depending on which of them is successful it will result in a different history of events. If W_1 is successful, S_4 will be added to the Redo-PTM queue and operation push(d) occurred before operation push(e). In this scenario writer W_2 will apply the undo log stored at S_3, reverting all effects of its simulation, and will return the result of the execution done

Algorithm 3 Redo-PTM applyUpdate() pseudo-code

```
1  function APPLYUPDATE(updFunc, tid):
2      req[tid] ← updFunc                                          {1}
3      announce[tid] ← ¬announce[tid]
4      _poolSts ← stMatrix[tid]
5      _newSt ← _poolSts.states[_poolSts.lastIdx]
6      _c ← null
7      for _iter ← 0, 1 do
8          _curC ← curComb                                         {2}
9          _comb ← combs[_curC.idx]
10         _tail ← _comb.head
11         _tkt ← {_tail.seq + 1, tid, _poolSts.lastIdx}
12         cpySt(_tkt, stMatrix[_tail.tid].states[_tail.idx])      {3}
13         if _curC ≠ curComb then continue
14         _ringtail ← ring[_tail.seq%RSIZE]
15         if _tail ≠ _ringtail then                               {4}
16             if _ringtail.seq > _tail.seq then continue
17             ring[ltail.seq%RSIZE].cas(_ringtail, _tail)
18         if _c = null then _c, _idx ← exclusiveTryLock()         {5}
19         if ¬applyRedologs(_c, _tail, tid) then break           {6}
20         for _i ← 0, N - 1 do                                    {7}
21             _applied ← _newSt.applied[_i]
22             if _applied = announce[_i] then continue
23             _newSt.results[_i] ← req[_i]()          ▷ Func call
24             _newSt.applied[_i] ← ¬applied
25         end for
26         flushDeferredPWBs()
27         pwb(curComb)
28         _c.head ← _tkt
29         _c.rwLock.downgrade()                                   {8}
30         if curComb.cas(_curC, {_tkt.seq, tid, _idx}) then       {9}
31             _comb.rwLock.downgradeUnlock()
32             _ringTail ← ring[_tkt.seq%RSIZE]
33             if _ringTail.seq < _tkt.seq then
34                 pwb(curComb)
35                 ring[_tkt.seq%RSIZE].cas(_ringTail, _tkt)
36             _poolSts.lastIdx++
37             return _newSt.result[tid]
38         applyUndolog(_newSt)
39         reset(_newSt,_c)
40     end for
41     if _c ≠ null then
42         _c.rwLock.exclusiveUnlock()
43     _curC ← curComb
44     pwb(curComb)
45     psync()
46     _comb ← combs[_curC.idx]
47     _tail ← _comb.head
48     if _tail.seq ≠ _comb.seq then
49         _tail = ring[_comb.seq%RSIZE]
50     _tailState ← stMatrix[_tail.tid].states[_tail.idx]
51     return _tailState.results[tid]
52 end function
```

by W_1 stored in the `results` array. In case W_2 succeeds in transitioning `curComb` to reference `combs[0]`, then S_3 will be appended to the queue. The writer W_1 will have to roll-back all the effects of its simulation on `combs[1]`, leaving its operation `push(d)` to still be executed. Herlihy's combining consensus guarantees that, in the next simulation, `push(d)` will be executed by one of the concurrent threads.

Memory bounded wait-free queue. For simplicity, the queue of mutations has been until now represented as a linked-list based queue. In practice, our implementation of a *memory bounded wait-free queue* is array based, where the array is referred as `ring` with length RSIZE. A ring buffer implementation is suitable because a replica can be invalidated if an object `State` is re-used. All `State` objects are pre-allocated and each thread has RSIZE dedicated objects. The states matrix, `stMatrix`, is of size $N\times$ RSIZE.

In Algorithm 1, we present both the `Combined` and `State` class. These classes use the type `SeqTidIdx`, which corresponds to a 64-bit number that concatenates a 44-bit sequence, an 8-bit thread identifier that indicates the thread that last modified the region starting at the `root` address, and the index of a pre-allocated `State` belonging to that thread. As with the CX universal construction, Redo-PTM uses a strong reader-writer try-lock `rwlock` to protected each `Combined` instance and consequently the memory region associated. The strong try-lock guarantees wait-free progress using at most $N + 1$ replicas. The class `State` contains the relevant information regarding a transition of `curComb`. The `applied` array indicates which of the announced requests where executed. The `results` array contains the result of the applied operations. The `logHead`, `logTail` and `logSize` variables refer to the *physical log* where the redo and undo log are kept.

Like CX-PTM, the persistent memory layout is composed of a persistent header where the variable `curComb` is located, followed by an array of references to the dynamically allocated persistent objects. The rest of the persistent memory is partitioned in $N + 1$ regions where each of the N threads can execute in parallel. All other components of Redo-PTM are located in transient memory.

An interesting fact associated with this implementation is that no allocation is required during program execution, with the exception of allocations executed by the user code itself. Any mutative function calls Redo-PTM's persistent allocator, which is a sequential implementation because an exclusive lock was previously acquired for that region. We can therefore conclude that Redo-PTM is not only a wait-free universal construction, but it also provides wait-free allocation and deallocation. Algorithm 3 and Algorithm 2 present Redo-PTM's implementation using a ring of states, where the states are preallocated in `stMatrix`.

Timed variant. We developed a second version of Redo-PTM, which we named RedoTimed-PTM. On every update transaction, the RedoTimed-PTM variant limits the execution of the update transaction to the first two `Combined` instances for a limited period of time. This means that at the start of each update transaction, it can only make progress if one of the two instances are available, otherwise it will wait for another thread to execute it in its place. In our implementation, this limited period of time corresponds to four times the amount of time it takes to make a copy of the object. Still, RedoTimed-PTM has wait-free progress, like Redo-PTM, because the update transaction will no longer be restricted to those instances after the initial time period and will acquire an exclusive lock on one of the $N + 1$ instances. The goal of this variant is to increase the probability that the first 2 `Combined` instances are kept up to date, reducing the chances that any of the first 2 replicas become stale. RedoTimed-PTM

also implements a backoff mechanism [1] for all updater threads that are not able to acquire the exclusive lock on those first 2 instances.

Additional optimizations. To further explore all the potential improvements that a physical logging approach allows, we added several optimizations to RedoTimed-PTM, resulting in RedoOpt-PTM.

Store aggregation. The combining consensus incorporated in Redo-PTM allows a thread to aggregate transactions from multiple threads within a single transaction. This *combining* behavior, increases the probability of a memory location being updated several times, which allows for store aggregation, by writing a single time the final value. In our implementation we use a specially designed hash set that enables fast aggregation of the stores on each transaction and efficient reset and re-usage of the `State` instance.

Flush aggregation. With a similar rationale as *store aggregation*, by combining multiple transactions into one, we expected several modifications to occur in the same cache line, thus yielding high gains in performance by avoiding unnecessary flushes to PM. This technique has an even greater potential to aggregate `pwbs` because modifications on the allocator's metadata (header of a block) tend to occur on the same cache line as the user data modifications. In addition, when a transaction executes multiple allocations, the blocks are often placed contiguously by the allocator, in adjacent or shared cache lines, further enhancing this effect.

Copy using `ntstore`. Based on a recent study [29], using non-temporal stores can yield better results versus using regular stores followed by the `clwb` instruction. For scenarios where a large object needs to be copied, keeping it in the cache will be inefficient if the size of the cache can not hold the entire object. Because of this, during the copy of an object in Algorithm 3 we use `movntq`, a non-temporal move instruction that bypasses the CPU cache hierarchy to perform direct writes to PM.

Postpone issuing `pwbs`. The actual persistence to NVMM with the associated flush instruction is an expensive operation and it is therefore wise to postpone the issue of the `pwb` instruction as much as possible. The modifications done by a thread executing on a `Combined` instance, must be persisted only *before* the thread attempts to transition `curComb`. Until then, the addresses of the cache lines to be flushed can be kept in a volatile list associated to the `Combined` instance, and later issued by the thread that attempts to transition `curComb` to this `Combined` instance. If the cache line list grows larger than 1/10 the size of the object, then the `Combined` instance is marked to do a complete flush of the object before attempting the `curComb` transition.

Recovery. Implementing a correct recovery mechanism can be a complex task when designing a PTM or a database management system (DBMS). Lock-free algorithms have implicit resilience to faults, *i.e.*, their recovery method is usually simple. Moreover, for a subset of lock-free algorithms the recovery method is non-existent, also known as *null recovery* [28]. Our three constructions have null recovery.

Upon restarting after a failure, a lock-free algorithm with null recovery may spend some time completing operations that were on-going during the failure. In our constructions, upon initialization after a failure, the `curComb` is pointing to the latest persisted and consistent object instance, allowing new operations to start executing immediately.

6 Evaluation

In this section we present a detailed evaluation of our algorithms, using both synthetic benchmarks and a real-world application. These benchmarks were executed on a two-socket server, each socket holding a 2.5 GHz Intel Xeon Gold 5215 with 128GB of Intel Optane DC Persistent Memory, with a total of 20 cores (40 HW threads). The Operating System is Ubuntu LTS, using `gcc` 8.3 with the `-O3` flag. We use the `clwb` flush instruction as `pwb` and the `sfence` instruction as `pfence` and `psync`. Each data point is the median of 5 runs with each run executing for 20 seconds.

We compare our implementations with OneFile [40], a recently published technique that supports dynamic transactions on NVMM with wait-free progress, and with `libpmemobj` which is part of the PMDK [44] library. CX-PUC results are only presented for small sized data structures because the no-interposition requirement forces a complete flush of the data structure for every `curComb` transition, resulting in a very slow execution for medium to large data structures.

Synthetic benchmarks. We devised several benchmarks with persistent data structures in real PM. The first benchmark is a persistent variant of the SPS benchmark [8, 10, 24, 33, 34]. This benchmark allows us to understand the performance profile of the PTMs with transactions of varying sizes. Each swap exchanges the values of two randomly selected entries of an array of 10^6 64-bit integers, hence modifying two memory words in PM. In this benchmark there is no allocation being performed inside each transaction. As shown in Figure 4, RedoOpt-PTM is the best of the wait-free techniques for most of the transaction sizes, except for the experiments with 1 swap per transaction. OneFile is able to perform better for very short transactions, because this scenario has fewer `pwbs` to aggregate and RedoOpt-PTM nonetheless spends time in that attempt. In general, we observe that performance decreases as the number of threads increase. The SPS benchmark is characterized by a highly disjoint write intensive workload, with less potential for optimizations. Nevertheless, RedoOpt-PTM is able to sustain its throughput by avoiding flushes when modifications occur in the same cache line.

Persistent data structure. Seen as existing work in lock-free data structures for persistent memory focus in queue implementations, we have implemented a similar

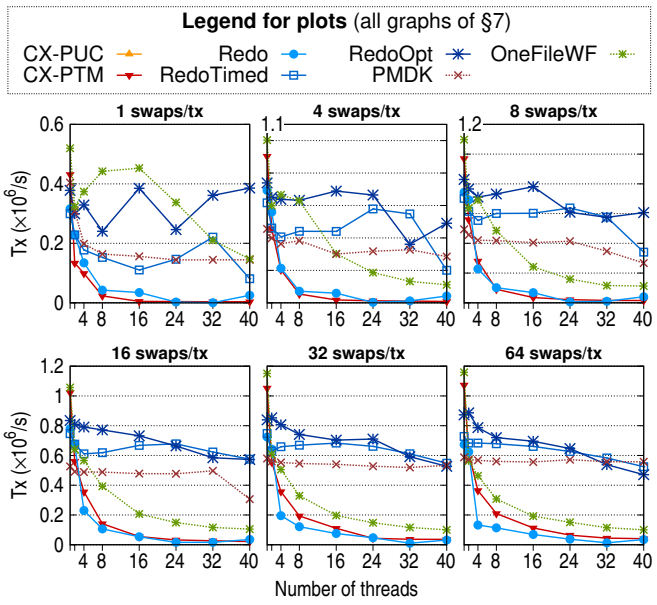

Figure 4. Persistent SPS integer microbenchmark.

benchmark [3, 20, 40] where the queue is pre-filled with 1,000 elements. In this benchmark, every thread executes a transaction with an enqueue operation followed by another transaction with a dequeue operation, maintaining on average the queue filled with 1,000 elements. In Figure 5 we compare the throughput of the same queue implementation when using RedoOpt-PTM, PMDK and OneFile, each using a persistent memory allocator. In addition, we also evaluate the handmade queues FHMP [20] and NormOpt [3], both using a volatile memory allocator, `libvmmalloc`, like in their original papers. This implies that all metadata of the volatile allocator will be lost in case of a crash, leaving those two queues inconsistent and unusable.

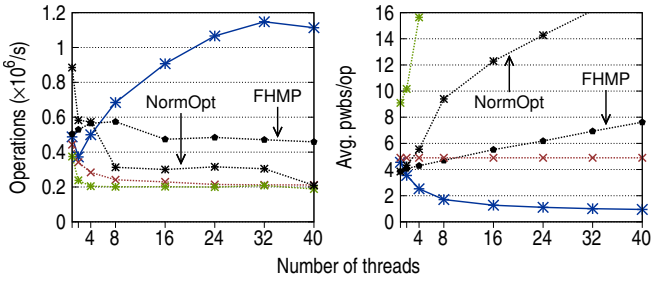

Figure 5. Persistent linked-list based queue pre-filled with 1,000 elements (FHMP and NormOpt use `libvmmalloc`), legend as in Figure 4.

As mentioned in prior work [29], on Intel Optane DC PM the number of modified cache lines and corresponding pwbs are a good indicator of the overall throughput. Generally speaking, the lower the number of pwbs an algorithm executes per transaction, the higher the throughput. RedoOpt-PTM is capable of combining operations from multiple threads under a single transaction, thus executing a

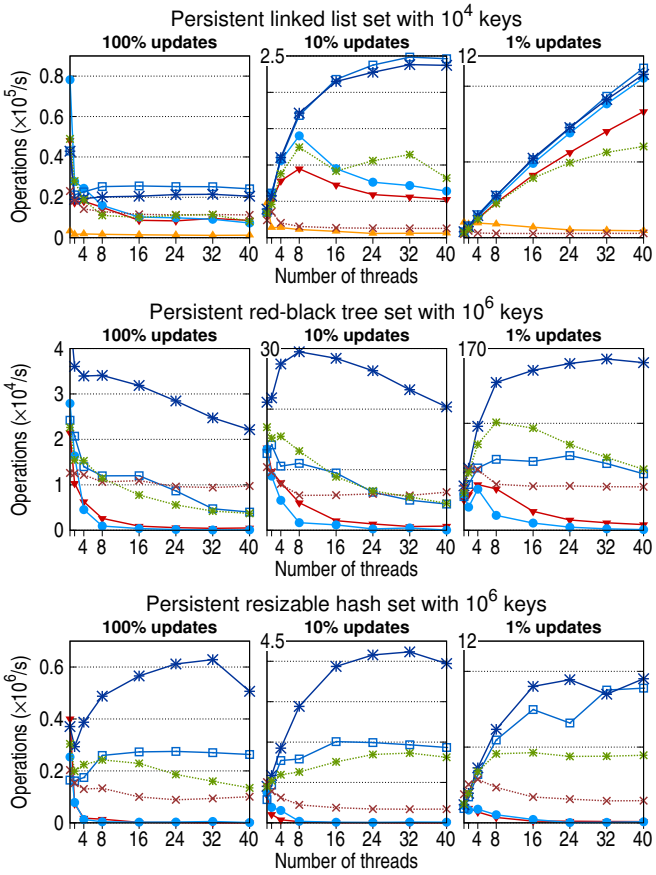

Figure 6. Throughput for an ordered linked list set of 10^4 keys (top), tree set of 10^6 keys (center) and hash set of 10^6 keys (bottom), legend as Figure 4.

single pwb per modified cache line, for all operations. This is a common occurrence on the queue, given that all operations will touch either its `head` or its `tail` and all operations will modify allocator metadata when allocating/de-allocating nodes of the queue. The queue benchmark benefits RedoOpt-PTM because it eliminates flushes that are done on the same cache line, and this is especially true as the number of threads in the system increases. The average number of pwbs decreases as the number of threads increases, explaining the performance gains for RedoOpt-PTM. The weight of the cache line flushes is clearly demonstrated as the two plots in Figure 5 display inverted trends.

The remaining evaluation shows multiple workloads for different data structures. All the data structures are *sets* and the micro-benchmarks described next have the same procedure. A set is filled with 1 million keys, or 10 thousand keys in case of the linked list implementation, and we randomly select keys doing either a lookup or an update, with a probability that depends on the percentage of updates for each particular workload (100%, 10% or 1%). For a *lookup*, we randomly select two keys from the existing keys in the set and call `contains(key)` for each of them; for an *update*, we randomly select one key from the existing keys in the set

and call `remove(key)`, and if the removal is successful we re-insert the same key with a call to `add(key)`, thus maintaining the total number of keys in the set.

For all tested data structures, RedoOpt-PTM is either among the best or the best of the PTMs. On the linked list set with 10^4 keys (Figure 6, top) RedoOpt-PTM can double the performance of state of the art PTMs, like OneFile or PMDK, for all tested scenarios. The results of Redo-PTM with no added optimization can already perform better than OneFile, however RedoTimed-PTM can double the performance because transaction execution is constrained to the first two instances at the start, minimizing the number of copies. The reduction of the number of flushes done by RedoOpt-PTM is not relevant for the linked list benchmark, because the traversal of the list during an update operation is the dominant cost. Even if our initial motivation was to improve performance for linked list implementation, and we have accomplished that objective, we can see even greater differences for tree and hash based sets. This happens because OneFile also does not re-execute the insert or delete function, but instead helper threads copy and apply the redo log published by the winning thread and the redo log only contains the store instructions.

Figure 6 shows execution results for a sequential implementation of a balanced red-black tree (center) and a resizable hash set (bottom) with 10^6 keys. Both implementations are able to triple the performance of the OneFile wait-free implementation and PMDK. For the resizable hash set, RedoOpt-PTM scales until 32 threads, even for write intensive scenarios, because of the optimizations that aggregate flushes and stores. Most of the modifications performed by an insert or a delete are done by the memory allocator, and there is an enormous potential to aggregate modifications. As the number of threads increases, more operations can be grouped by the combining consensus. There is more scalability until 24 threads because the machine has 20 cores. From this point on, there is a marginal increase and with 40 threads starts to decrease where the system is running in oversubscription. On the other hand, the red-black tree implementation with 100% updates shows negative scalability. This is due to the update operation, on a strictly balanced tree, being a large transaction with many modifications that can not be aggregated, thus adding a high cost to the next transaction that requires a new `Combined` instance to be updated to the most recent state.

Breakdown of the costs. To give more insight on where time is spent during an update operation for the different PTMs, Table 1 shows the time spent on each task for workloads consisting of 100% update transactions.

The measurements were done on the hash map and red-black tree pre-filled with 1 million keys, running the same benchmark described previously. The benchmark executed for 60 seconds with 4 and 16 threads. The `updateTX` column indicates the average number of μs spent for the whole operation, as well as the slowdown of the PTMs relative to RedoOpt-PTM (first row of each workload). The other columns indicate the time, in percentage of the whole operation duration, for respectively applying the redo log, issuing flushes, copying the data structure, executing the user code (lambda) and sleeping.

		updateTX	apply	flush	copy	lambda	sleep
Hash 4T	RedoOpt	**10.32**	0.4%	0.0%	14.2%	6.2%	63.4%
	Redo	**313** (30.3×)	0.4%	0.2%	95.0%	2.0%	0.0%
	RedoTimed	**23** (2.2×)	2.1%	0.8%	46.3%	12.0%	18.9%
	Onefile	**18** (1.7×)	30.8%	1.2%	0.0%	25.7%	0.0%
Hash 16T	RedoOpt	**28.28**	0.1%	0.0%	4.2%	1.4%	88.3%
	Redo	**5882** (208×)	0.0%	0.0%	99.7%	0.1%	0.0%
	RedoTimed	**59** (2.1×)	0.8%	0.6%	12.0%	3.9%	72.9%
	Onefile	**70** (2.5×)	27.9%	6.4%	0.0%	26.7%	0.0%
Tree 4T	RedoOpt	**117.78**	3.2%	0.0%	0.5%	17.2%	71.6%
	Redo	**881** (7.5×)	1.8%	12.1%	74.5%	10.4%	0.0%
	RedoTimed	**286** (2.4×)	18.6%	8.9%	0.0%	48.3%	24.2%
	Onefile	**262** (2.2×)	38.5%	17.3%	0.0%	33.2%	0.0%
Tree 16T	RedoOpt	**502.02**	0.7%	0.0%	0.2%	4.6%	92.7%
	Redo	**41667** (83×)	0.3%	2.8%	92.9%	3.1%	0.0%
	RedoTimed	**1334** (2.7×)	5.1%	3.0%	0.0%	11.8%	80.5%
	Onefile	**2066** (4.1×)	25.4%	50.3%	0.0%	18.1%	0.0%

Table 1. Breakdown of average time spent per update transaction for different PTMs operating on a hash set and a red-black tree with 4 and 16 threads.

Each PTM has different profiles of task execution. OneFile spends most of its time applying the redo log and flushing both the modifications and the redo log. The execution time of Redo-PTM is dominated by the data structure copy, while RedoTimed-PTM and RedoOpt-PTM do mostly backoff waiting for their operation to be executed by another thread. Comparing the profile of the hash map with the red-black tree, the time spent in flushing data to persistent memory increases for OneFile and Redo-PTM. This is not the case for RedoTimed-PTM and even less so for RedoOpt-PTM, clearly reflecting the optimizations described in section 5.

Wait-Free in-memory database. A natural use for a PTM is to serve as a database engine. LevelDB [21] and RocksDB [14] are well known in-memory databases that persist data to disk. We have implemented RedoDB, the first wait-free in-memory key-value store database, using a resizable hash map annotated with the transactional semantics provided by RedoOpt-PTM. This hash map was extended with iterator capabilities. RedoDB provides the necessary functionality to run LevelDB/RocksDB benchmarks, and its implementation relies on RedoOpt-PTM to guarantee durable linearizable transactions to persistent memory, *i.e.*, transactions in RedoDB have *serializable isolation* [4].

Figure 7 shows three different benchmarks that are part of the RocksDB (version 6.5) `db_bench` suite when executed on a database with 10^6 (top) and 10^7 (bottom) keys, where the keys are 16 bytes and the values are 100 bytes. Similarly to previous work [29] the RocksDB benchmarks were executed using the `–sync` flag to enable *serializable isolation* on every

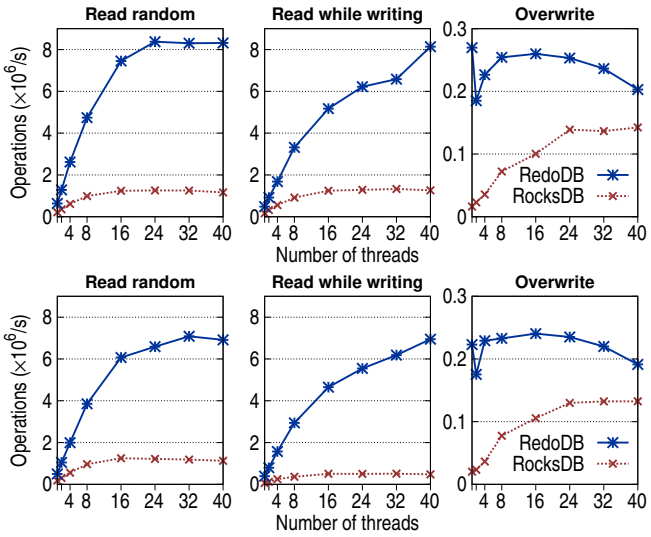

Figure 7. RocksDB vs. RedoDB with 1M (top) and 10M (bottom) keys.

operation and the database is stored in a PM device formatted with `ext4` with journalling enabled.

The *read random* benchmark executes random lookups on the database. Read-only operations in RedoOpt-PTM have their own snapshot of the data and, after acquiring the lock on the `Combined`, execute without any further synchronization, giving RedoDB high throughput for such operations. In the *reading while writing* benchmark, we show the number of lookups per second while an extra thread continuously overwrites a random key/value in the database. RedoDB excels in this workload because it allows read-only and mutative operations to concurrently execute, each on its own snapshot of the data, enabling it to surpass RocksDB by up to a factor of 14×. In the rightmost plot, *overwrite*, the database is pre-filled and for randomly chosen pairs the value is overwritten.

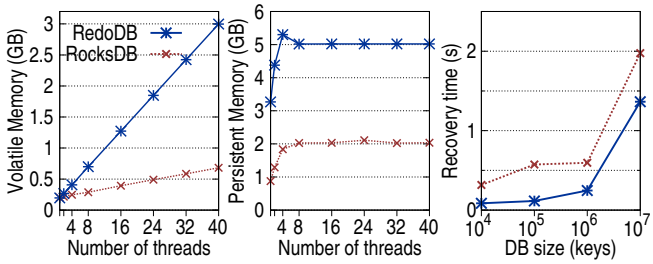

Figure 8. Volatile and Non-Volatile Memory usage of `fillrandom`, and recovery time after a crash, with 10M keys.

Figure 8 shows the volatile memory usage (left) and persistent memory usage (middle) for RocksDB and RedoDB. RedoDB utilizes more volatile memory, to store the `States` containing the physical logs of the operations. The number of `States` is proportional to the number of active threads, causing a linear memory growth. Regarding non-volatile memory usage, because the backoff policy limits execution on the first two `Combined` instances for a period of time, only

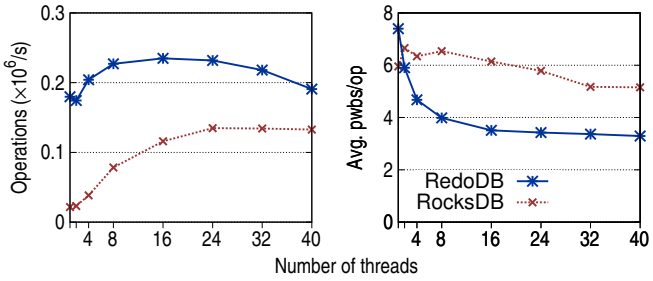

Figure 9. Throughput of `fillrandom` (left), pwbs of `fillrandom` (right), with 10M keys.

those two instances are used in practice during the benchmark. However, RedoDB has a non-volatile storage usage which is larger than 2x that of RocksDB, due to our persistent allocator reserving blocks whose sizes are powers of 2, which wastes extra space in PM. In the rightmost plot we measure the time it takes to recover and execute the first `fillrandom` transaction, after a simulated failure. RedoDB takes slightly less time to recover from a failure than RocksDB, and its recovery time increases with database size because a copy of the data structure is required on the first update transaction.

The left plot of Figure 9 shows the number of operations for the `fillrandom` benchmark, which randomly fills up the database with 10^7 key/value pairs. The right plot displays the number of `clwb` flush instructions for the `fillrandom` benchmark. On write-intensive workloads, RedoDB's higher throughput stems from executing fewer flushes to NVMM. The same rationale applies to the `overwrite` benchmark.

RedoDB improves over the state of the art in-memory databases, with added progress guarantees and a fully transactional system, showing that both performance *and* resilience are attainable goals. The code of the PTMs and RedoDB is available as open source https://github.com/pramalhe/RedoDB.

7 Conclusion

While the PTMs described in this paper have a higher cost in memory usage, they provide wait-free progress for NVMM and do so in an efficient manner. Our generic constructions allow programmers to quickly develop any data structure, with wait-free progress, resilience to failures and integrated memory reclamation, making them an invaluable tool for developing applications targeting persistent memory.

Non-volatile memory is expected to be a game-changing technology, particularly in the development of DBMS. Due to the wait-free progress and ease-of-use of our PTMs, they can be used as DBMS engines. We have used RedoOpt-PTM to build RedoDB, an in-memory DBMS which provides API compatibility with RocksDB and higher relative performance when deployed in PM with the same durability guarantees. RedoDB is the first key-value store to offer null recovery and wait-free durable linearizable dynamic transactions.

References

[1] Thomas E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[2] Hillel Avni, Eliezer Levy, and Avi Mendelson. Hardware transactions in nonvolatile memory. In *International Symposium on Distributed Computing*, pages 617–630. Springer, 2015.

[3] Naama Ben-David, Guy E. Blelloch, Michal Friedman, and Yuanhao Wei. Delay-Free Concurrency on Faulty Persistent Memory. In *The 31st ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2019, Phoenix, AZ, USA, June 22-24, 2019.*, pages 253–264, 2019.

[4] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. A critique of ANSI SQL isolation levels. *CoRR*, abs/cs/0701157, 2007.

[5] Kumud Bhandari, Dhruva R Chakrabarti, and Hans-J Boehm. Makalu: Fast recoverable allocation of non-volatile memory. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 677–694. ACM, 2016.

[6] Trevor Brown and Hillel Avni. PHyTM: Persistent Hybrid Transactional Memory. *PVLDB*, 10(4):409–420, 2016.

[7] Dhruva R Chakrabarti, Hans-J Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. *ACM SIGPLAN Notices*, 49(10):433–452, 2014.

[8] 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. *ACM Sigplan Notices*, 46(3):105–118, 2011.

[9] Nachshon Cohen, Rachid Guerraoui, and Igor Zablotchi. The Inherent Cost of Remembering Consistently. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures, SPAA 2018, Vienna, Austria, July 16-18, 2018*, pages 259–269, 2018.

[10] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient Algorithms for Persistent Transactional Memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 271–282. ACM, 2018.

[11] Andreia Correia and Pedro Ramalhete. Strong trylocks for reader-writer locks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 387–388. ACM, 2018.

[12] Andreia Correia, Pedro Ramalhete, and Pascal Felber. A wait-free universal construction for large objects. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '20, page 102–116, New York, NY, USA, 2020. Association for Computing Machinery.

[13] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *International Symposium on Distributed Computing*, pages 194–208. Springer, 2006.

[14] Facebook. RocksDB. https://rocksdb.org, 2019.

[15] Panagiota Fatourou and Nikolaos D Kallimanis. A highly-efficient wait-free universal construction. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 325–334. ACM, 2011.

[16] Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. Time-Based Software Transactional Memory. *IEEE Trans. Parallel Distrib. Syst.*, 21(12):1793–1807, 2010.

[17] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2008, Salt Lake City, UT, USA, February 20-23, 2008*, pages 237–246, 2008.

[18] Sergio Miguel Fernandes and João P. Cachopo. Lock-free and scalable multi-version software transactional memory. In *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011*, pages 179–188, 2011.

[19] Keir Fraser. Practical lock-freedom. Technical report, University of Cambridge, Computer Laboratory, 2004.

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

[21] Sanjay Ghemawat and Jeff Dean. LevelDB. *URL: https://github. com/-google/leveldb,% 20http://leveldb. org*, 2011.

[22] Ellis Giles, Kshitij Doshi, and Peter J. Varman. Continuous checkpointing of HTM transactions in NVM. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management, ISMM 2017, Barcelona, Spain, June 18, 2017*, pages 70–81, 2017.

[23] Ellis R Giles, Kshitij Doshi, and Peter Varman. SoftWrAP: A lightweight framework for transactional support of storage class memory. In *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*, pages 1–14. IEEE, 2015.

[24] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. Persistency for synchronization-free regions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 46–61, 2018.

[25] Maurice Herlihy. A Methodology for Implementing Highly Concurrent Data Structures. In *Proceedings of the Second ACM SIGPLAN Symposium on Princiles & Practice of Parallel Programming (PPOPP), Seattle, Washington, USA, March 14-16, 1990*, pages 197–206, 1990.

[26] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture, San Diego, CA, USA, May 1993*, pages 289–300, 1993.

[27] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via JUSTDO logging. *ACM SIGARCH Computer Architecture News*, 44(2):427–442, 2016.

[28] Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *International Symposium on Distributed Computing*, pages 313–327. Springer, 2016.

[29] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. Basic performance measurements of the intel optane DC persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[30] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. DHTM: Durable Hardware Transactional Memory. In *Proceedings of the International Symposium on Computer Architecture*, 2018.

[31] Arpit Joshi, Vijay Nagarajan, Stratis Viglas, and Marcelo Cintra. ATOM: Atomic durability in non-volatile memory through hardware logging. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pages 361–372. IEEE, 2017.

[32] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. DudeTM: Building durable transactions with decoupling for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 329–343. ACM, 2017.

[33] Youyou Lu, Jiwu Shu, and Long Sun. Blurred persistence in transactional persistent memory. In *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*, pages 1–13. IEEE, 2015.

[34] Youyou Lu, Jiwu Shu, Long Sun, and Onur Mutlu. Loose-ordering consistency for persistent memory. In *Computer Design (ICCD), 2014 32nd IEEE International Conference on*, pages 216–223. IEEE, 2014.

[35] Virendra J. 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. *CoRR*, abs/1804.00701, 2018.

[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. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 499–512. ACM, 2017.

[37] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, March 1992.

[38] Matheus Almeida Ogleari, Ethan L Miller, and Jishen Zhao. Steal but no force: Efficient hardware undo+ redo logging for persistent memory systems. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*, pages 336–349. IEEE, 2018.

[39] Pedro Ramalhete and Andreia Correia. POSTER: A Wait-Free Queue with Wait-Free Memory Reclamation. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 453–454. ACM, 2017.

[40] Pedro Ramalhete, Andreia Correia, Pascal Felber, and Nachshon Cohen. OneFile: A Wait-Free Persistent Transactional Memory. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 151–163. IEEE, 2019.

[41] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. ThyNVM: Enabling software-transparent crash consistency in persistent memory systems. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 672–685. ACM, 2015.

[42] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[43] Long Sun, Youyou Lu, and Jiwu Shu. DP 2: reducing transaction overhead with differential and dual persistency in persistent memory. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, page 24. ACM, 2015.

[44] PMDK team. Persistent Memory Development Kit. https://pmem.io/pmdk/, 2018.

[45] R Kent Treiber. *Systems programming: Coping with parallelism.* International Business Machines Incorporated, Thomas J. Watson Research Center New York, 1986.

[46] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*, pages 91–104, 2011.