OneFile: A Wait-free Persistent Transactional Memory

Pedro Ramalhete
Cisco Systems
pramalhe@gmail.com

Andreia Correia, Pascal Felber
University of Neuchâtel
andreia.veiga@unine.ch, pascal.felber@unine.ch

Nachshon Cohen
EPFL
nachshoncohen@epfl.ch

***Abstract*—A persistent transactional memory (PTM) library provides an easy-to-use interface to programmers for using byte-addressable non-volatile memory (NVM). Previously proposed PTMs have, so far, been blocking. We present OneFile, the first *wait-free PTM* with integrated wait-free memory reclamation. We have designed and implemented two variants of the OneFile, one with lock-free progress and the other with bounded wait-free progress. We additionally present software transactional memory (STM) implementations of the lock-free and wait-free algorithms targeting volatile memory. Each of our PTMs and STMs is implemented as a single C++ file with ∼1,000 lines of code, making them versatile to use. Equipped with these PTMs and STMs, non-expert developers can design and implement their own lock-free and wait-free data structures on NVM, thus making lock-free programming accessible to common software developers.**

I. Introduction

Modern computer architectures are based on shared memory systems, where multiple threads or processes can simultaneously access the same data. Sharing data in such a way leads to what is called *the concurrency problem*. The simplest solution to this problem is to ensure mutual exclusion using *locks*. As far back as 1963, Dijkstra was the first to show a mutual exclusion algorithm, originally made by Dekker [1]. Locks are by their very nature *blocking* and it took more than 20 years for Treiber to publish, in 1986, the first non-blocking data structure, a lock-free stack [2]. Since then, a multitude of non-blocking data structures have emerged over the years. Several of these data structures contained minor errors, a few had fatal design flaws, serving as a testament to the difficulty of designing and implementing *correct* non-blocking data structures.

Researchers quickly realized that reasoning about lock-free code was a difficult task and started to propose alternatives. In 1991, Herlihy designed the first universal construct (UC) with wait-free progress [3]. UCs with wait-free progress can wrap sequential implementations of an object or data structure, so as to provide wait-free progress when accessing the methods and data of the underlying object. Other UCs with lock-free and wait-free progress have been proposed since then, though up until now their performance has been too low to be of practical usage, remaining mostly the subject of theoretical work.

By their very nature, UCs imply no change to the sequential implementation of the underlying object. This is an attractive property from a theoretical standpoint, but in practice it can be reasonable to demand from the user some sort of minor annotation on the sequential implementation. Herlihy saw this opportunity in the design space and, two years later, together with Moss, they proposed another approach for non-blocking data structures [4], coining the term *transactional memory* (TM). The intent was for CPU manufacturers to implement TM as a set of hardware instructions (HTM). This dream only came to fruition many years later and, even now, HTMs still do not provide non-blocking progress guarantees.

In 1997, Shavit and Touitou presented the first software-only TM capable of running on commodity hardware [5], creating the first functional software transactional memory (STM). Their proposal went mostly unnoticed, likely because the research community was busy developing "hand-made" custom non-blocking data structures. However, by the turn of the century, STMs started receiving more attention, peaking in the decade of 2000-2010. Since then, many STMs have been proposed though rarely claiming lock-free progress and none with lock-free memory reclamation. Emerging out of this gold rush, lock-based STMs have been the winners when it comes to performance.

Until today, the role of STMs has been delegated to providing a user-friendly interface for working with lock-based concurrency. Making life easier for software engineers working with locks is an important undertaking, yet, it is far below the goal of what Herlihy originally thought UCs and TMs should be: a way of letting non-expert developers create their own lock-free data structures, safely and correctly.

The introduction of byte-addressable non-volatile memory (NVM) has added a new dimension to this old problem. Due to its need for resilience to failures, application development for NVM is particularly well suited to transactions, and constructing an STM with efficient and durable (ACID) transactions has become an important problem to solve [6]. Several persistent software transactional memory (PTM) exist in the literature with none claiming lock-free progress, at the exception of RomulusLR [7] which provides wait-free progress but solely for its read-only transactions.

In this paper we present OneFile, a novel PTM/STM with wait-free progress. OneFile is specifically designed to enable non-experts to conceive and implement their own lock-free and wait-free data structures. OneFile comes with

integrated wait-free memory reclamation. With OneFile, transforming a sequential implementation of a data structure into a wait-free implementation is as easy as annotating the types used internally in the data structure, replacing the allocation and deallocation methods with the ones provided by OneFile, and wrapping all methods in calls to `updateTx` or `readTx`.

In short, with OneFile we make the following contributions. We introduce a novel transactional memory algorithm that relies on a new multi-word CAS technique, with two different PTMs, one with lock-free progress and the other with bounded wait-free progress, which provide dynamic transactions for NVM with durable linearizability, *i.e.*, ACID transactions. We present two STMs based on the same design. Each of these four implementations is a single C++ header with ∼1,000 lines of code, simplifying the task of integrating with any sequential implementation of a data structure. Our STMs and PTMs have efficient lock-free and wait-free memory reclamation. OneFile's performance is capable of rivaling that of hand-made lock-free data structures, and it provides linearizable consistency for any method, something notoriously difficult to achieve for hand-made lock-free data structures.

The rest of the paper is organized as follows. We first discuss related work in §II. We then introduce the lock-free version of the OneFile algorithm in §III, followed by the wait-free algorithm. We present details on our solutions for providing efficient wait-free memory reclamation on NVM and volatile memory in §IV. We provide an in-depth evaluation of OneFile in §V and finally conclude in §VI.

II. Related Work

Our goal is to provide a way for non-experts to design *durable linearizable* [8] concurrent data structures that preserve their state after a failure. As such, we focus our attention on PTMs whose transactions guarantee this property.

PTMs: Most of the existing PTM implementations rely on one of two logging techniques: write-ahead logging (WAL) with *undo* [9]–[11] or *redo* [6], [10] log. A persistent log approach adds complexity to the implementation given that the log used to revert to a consistent state must itself be allocated in persistent memory and be reverted in case of failure. Two notable algorithms based on undo log are Atlas [12] and PMDK [13].

Atlas [12] requires an entry in the undo log for every store to persistent memory. Each log entry has four words: the destination address of the store, the original value at the address, a pointer to the next node, and the size of the store combined with the log type. This implies that a persistent store in user code will cause a total of 5 stores to NVM. To minimize cache line flushes, Atlas uses a helper thread to aggregate memory locations and to guarantee that a consistent state is persisted to memory. As with any undo log approach, the algorithm has to guarantee that the log entry is made persistent before any in-place modification. Therefore, modifications are written-back to NVM using a persistent write-back (`pwb`) instruction and a persistent fence (`pfence`) [8] is used to stall until `pwb`s finish. Transactions in Atlas are only *buffered durable linearizable* [8] since some finished transactions may not be included in a post-failure state.

PMDK [13] is a more recent undo log implementation that reduces the number of persistent fences by aggregating all modifications done on each object inside a transaction. The persistent memory allocator is highly optimized to allow a significant reduction on the number of `pwb` instructions.

Mnemosyne [6] was the first PTM. It uses a redo log and is built on top of the lock-based TinySTM [14], [15].

Romulus [7] is a recent development that does not rely in a persistent log but instead uses two replicas of the data to guarantee consistent recovery from a non-corrupting failure. The user code is executed directly in-place in the first replica and at the end of the transaction the modifications propagate to the second replica. To improve performance on the copy procedure, Romulus uses a volatile log to record the memory locations that were modified. Two implementations are publicly available, one with a scalable reader-writer lock, named RomulusLog, and the other using a universal construct supporting wait-free read-only transactions, named RomulusLR. Both variants use flat-combining [16] for their update transactions. RomulusLR was the first PTM to provide concurrent read transactions with wait-free progress.

STMs: A single lock-free STM has been identified in the literature, by [17], who proposed a modification of JVSTM [18] with lock-free operations. Their implementation is for the Java virtual machine (JVM) and, therefore, uses the JVM's garbage collector for memory reclamation, which is itself not lock-free.

[19] present a detailed review of many STMs. Among those, we specifically focus on TinySTM [14], [15] and ESTM [20] in our comparative evaluation. TinySTM uses eager locking and it deploys an array of locks to access shared memory. Similarly to TL2 [21], it relies on a shared counter as a "clock" to protect memory regions from conflicting accesses and order updates. TinySTM maintains a write-set as well as a read-set that is validated after the locks of the write-set are acquired, at commit time. ESTM [20] provides a variant of *elastic* transactions that support efficient implementations of search data structures. A key feature of ESTM is that, during its execution, an elastic transaction can be cut into multiple normal transactions, hence reducing the risk of conflicts and improving performance.

Multi-Word CAS (MCAS) Algorithms: At the core of the OneFile lies a technique that provides multi-word compare-and-set operations. MCAS is a programming abstraction that allows a thread to update a series of memory addresses in a single step [22]. This update is successful only when the values at these addresses did not change between

the reading of those values and the call to MCAS.

The first practical lock-free MCAS was proposed in [23]. A CAS operation is used to replace the expected value at an address with a pointer to a descriptor object. Two bits of the word are reserved to distinguish between values and pointers to descriptor objects. In order to prevent the "ABA" synchronization problem, the authors designed a variant with a *double-compare single-swap*.

Recently, Pavlovic et al. [24] have described an MCAS technique for persistent memory. This technique is blocking, requires HTM support, executes 4 persistence fences per operation and steals one bit from each modified word.

Wang et al. [25] have shown an MCAS approach with lock-free transactions and blocking memory reclamation, which steals two bits from each word.

Other techniques have been proposed [23], [26]–[28] that can work well for specific scenarios where MCAS are typically deployed, however, to use it as a part of an STM, a more versatile algorithm is needed, one that is ABA-free and that uses no bit of the word. In §III we describe such an algorithm and how to integrate it into an STM.

III. Persistent Transactional Memory

We first describe in this section the basic OneFile lock-free algorithm. We will then introduce a variant of the basic design that provides bounded wait-free progress.

A. Principle and Architecture

OneFile is a redo-log, word-based PTM, which does not maintain a read-set. Each thread uses and exposes a write-set to the other threads, so that they are able to help apply the current ongoing transaction. All mutative transactions are effectively serialized (ordered) on a single variable named `curTx` which is composed of a monotonically increasing sequence number and an index. The sequence number `#curTx` is unique and it allows read operations to have fast and consistent operations, using a technique similar to TL2 [21] and TinySTM [14]. The index indicates to which write-set (*i.e.*, thread identifier `tid`) the current transaction pertains to. We name the 64 bit-wide combination of the index and sequence the *transaction identifier*.

Each thread has its own write-set, however, other threads may read from this write-set during the *applying* phase when helping another transaction. To prevent ABA issues during the re-usage of the log and as an optimization to quickly identify when a transaction has been applied, each write-set contains a `request` variable. The write-set can be correctly re-used because if a thread A attempts to help another thread B to complete B's transaction, it first reads the `request` of B, then makes a local copy of B's write-set and after a load-ordering fence, re-checks that B's `request` has not changed and matches the sequence of `curTx`. When the value of `request` is the same as `curTx`, it indicates that the write-set is to be applied, *i.e.*, at least some of its entries need to be executed with a *double-word compare-and-set* (DCAS) instruction (`CMPXCHG16B`). If the `request` is different from `curTx`, then the current transaction has been applied and a new mutative transaction can now start. The basic data type is denoted `TMType`. It is composed of two 64-bit adjacent words, the first word containing the actual value, `val`, and the second word containing a numerical sequence, `seq`. A DCAS instruction acts simultaneously on the two adjacent 64-bit words of `TMType`.

A mutative transaction in the OneFile algorithm consists of three phases: *transform*, *commit* and *apply*.

During the *transform* phase, the user code for that update transaction is invoked, without the stores to `TMType` objects being executed. Instead, a redo log (write-set) is created with one entry for every store on a unique memory location, indicating the address of the `TMType` and its corresponding new value. The user code can be passed to the STM's `updateTx` method as a function pointer, a `std::function`, or a lambda expression (closure). This step effectively transforms the function containing the user code into a *write-set* [4]. A store on a memory location already present in the write-set will replace the previous value in the write-set with the new value. The write-set is implemented as an array with an intrusive hash-set, where short-sized transactions (less than 40 stores) do a linear lookup in the array, while larger transactions do a lookup on the hash-set. Unlike other STMs, there is no *read-set* on OneFile.

After the user function returns from its invocation, the transaction enters the *commit* phase. The `updateTx` method will attempt to change the shared variable `curTx` with a CAS, from the current transaction identifier to the new transaction identifier, where the sequence advances by one and the index becomes the current thread's identifier. If the CAS is successful, the transaction has effectively been *committed*, otherwise this transaction has failed.

On the third phase, the currently committed transaction is *applied*, by executing one DCAS for every entry in the write-set of the last committed transaction, if it has not already been applied. This write-set may belong to the current thread, or to another thread which successfully executed the CAS on `curTx` in its own commit phase. Regardless of the case, one thread has made progress, thus guaranteeing lock-free progress. To apply the write-set, we execute a DCAS on each double-word location (`TMType`), from the current value and sequence to the new value with the current transaction's sequence, as indicated in the current transaction's write-set. As explained in §III-C, this MCAS technique guarantees there are no ABA issues even if one thread lags behind on an older transaction's write-set during the apply phase. At the end of this phase, the write-set's `request`'s sequence is advanced to indicate to other threads that the corresponding apply phase is complete and that a new transaction can now start.

In OneFile, read-only transactions begin by reading the

Algorithm 1: Write-set and TMType with interposition

```
1  struct WriteSetEntry {
2    TMType<uint64_t>* addr;                    // address of TMType to modify
3    uint64_t val;                              // desired value to change to
4  };
6  struct WriteSet {
7    WriteSetEntry writeSet[TX_MAX_STORES];
8    uint64_t numStores {0};
9    void apply(uint64_t seq, const int tid) {
10     for (uint64_t i = 0; i < numStores; i++) {
11       WriteSetEntry& e = log[(tid*8 + i) % numStores];
12       uint64_t lval = e.addr->val.load(memory_order_acquire);
13       uint64_t lseq = e.addr->seq.load(memory_order_acquire);
14       if (lseq < seq) DCAS(e.addr, lval, lseq, e.val, seq);
15     }
16   }
17 };
19 template<typename T> struct TMType {
20   std::atomic<uint64_t> val;
21   std::atomic<uint64_t> seq;
22   inline T load() {                                  // load interposition
23     T lval = (T)val.load(memory_order_acquire);
24     uint64_t lseq = seq.load(memory_order_acquire);
25     if (lseq > seq(tl_oldTx)) throw AbortedTxException;
26     return (T)gOF.writeSets[tl_tid].lookup(this, lval);
27   }
28   inline void store(T newVal) {                      // store interposition
29     gOF.writeSets[tl_tid].addOrReplace(this, newVal);
30   }
31 };
```

current value of `curTx` and helping in the *apply* phase if the current transaction is not yet applied, so as to have a globally consistent view. Then, the user function that was passed to `readTx` will be invoked, with every load on a `TMType` object allocated by the STM interposed, and a check done of whether the sequence on each read word is no higher than the one at the start of the function's invocation. This algorithm is similar to read-only transactions in TL2 [21]. If one of the loads fails the previous checks, the read-only transaction is restarted via the exception mechanism, causing the re-reading of the current value of `curTx` and restarting of the user-provided read-only function.

B. Lock-Free PTM

We first introduce the general operation of the lock-free PTM. A persistent transaction executes the following steps:

1) Read `curTx` and store it locally in `oldTx`.
2) If there is an ongoing transaction, help apply it (see steps 8 to 10) and go to 1.
3) Execute user code, with loads and stores interposed.
4) Commit if write-set is empty (read-only transaction).
5) Open `request` by assigning it a new transaction identifier `newTx`, with sequence `#oldTx+1` and current `tid`.
6) Flush write-set to persistent memory, executing one `pwb` for every cache line.
7) Commit transaction by CASing `curTx` from `oldTx` to `newTx`, and flush it with a `pwb`.
8) Apply transaction, executing one DCAS for every entry in write-set.
9) Flush modified words using one `pwb` for each address.
10) Close `request` by CASing it to `newTx+1`.

The algorithm for the STM is similar, minus the `pwb`s.

C. MCAS Algorithm

We now detail the operation of our new MCAS technique, shown in Alg. 1. At the beginning of the *apply* phase, the write-set has been created with one `WriteSetEntry` per modified word, with a total of `numStores` entries. Applying the write-set consists of attempting each DCAS until all entries have been fully processed (lines 9–16).

Each operation is uniquely identified by a sequence, with all modified words of that operation having this unique `seq`. Seen as all update transactions are serialized in the *commit* phase, if the `seq` on a `TMType` is equal (or higher) than the `seq` initially read of the `curTx`, then some other thread has already applied that DCAS. This MCAS is ABA-free because any delayed thread will fail in the execution of its DCAS due to the `seq` no longer matching.

Unlike other MCAS algorithms, in our technique no bit is *stolen* [22] from the word containing the value or pointer in `TMType`, which gives it maximum flexibility and allows for easy deployment. On the other hand, it does require hardware support for a DCAS or equivalent instruction, which is supported for x86 although not implemented by all CPU vendors (likely because it was initially not considered useful for designing synchronization mechanisms [29]).

An alternative to using DCAS exists with a single word, without the need to *steal* bits or to use a sequence, if the store on the value can be made conditional with the load on `curTx`. Although is it possible to implement this operation as a short hardware transaction, *e.g.*, with Intel's TSX, as it stands today, such approach does not provide the progress guarantees necessary in the apply phase of OneFile for maintaining lock-free progress. The lock-free progress will hold as long as the hardware transaction implementation guarantees, in case of conflict, that at least one thread will eventually execute its transaction successfully. In our implementation we opted for `CMPXCHG16B` instead of TSX.

Pseudo code for the load and store interposing methods are also shown in Alg. 1 (lines 22 and 28).

D. Persistent Log and Recovery

Each thread contains a pre-allocated write-set in persistent memory. Before the *commit* phase, the write-set is flushed to persistence by executing one `pwb` for every four entries, where each entry of the log occupies two words, `addr` and `val`. On commit, the successful CAS acts as a `pfence` on x86, guaranteeing ordering with the prior `pwb`s and implying that if the new value of `curTx` is visible to other threads, then the write-set of the thread that won the CAS is *durable*.

For every 64 bit word of modified data by the user written to persistent memory, 3 other words are written: two words for the write-set log and one more when executing the DCAS, for the `seq` in the corresponding `TMType`. The *write amplification* [7] in OneFile-PTM is of 300%.

Our PTMs need no recovery method, a characteristic referred to as *null recovery* [8]. Either following a failure

or during normal execution, other threads will identify the write-set of the last committed transaction and whether that transaction is already closed. If the request is still open, they will attempt to apply the missing DCAS and close the corresponding request.

Null recovery is possible only if the DCAS instruction guarantees that both words are written to NVM atomically, which is the case for x86 because both words are on the same cache line [30], [31]. In architectures that do not provide such guarantee, the words must be written in order (`val` first, then `seq`), or a recovery method executed upon restart. Such a recovery method is trivially simple and consists of applying the values in the write-set using stores, each followed by a `pwb`, with a single final `pfence`.

E. Wait-Free PTM

We will now describe the modifications to the lock-free PTM required to create a similar PTM with bounded wait-free progress. Compared to the lock-free OneFile-PTM, the wait-free implementation has two extra member variables: an array of pointers to `std::function` where threads publish their operation and an array of results where the result of the operation is stored.

During the *transform* phase, an update transaction starts by encapsulating the code pertaining to its transaction in a `std::function` object and publishing a pointer to it in its thread's entry of the operations array. If there is an ongoing transaction that was committed but not yet applied, it will help apply that transaction, otherwise, it will aggregate all the functions in the operations array and invoke them one by one, including its own, producing a single write-set with all these operations. It will then attempt a *commit*, similar to the lock-free algorithm. The transform step will be repeated at most two times, because by the second iteration, its own operation is guaranteed to have been executed. This approach is inspired by Herlihy's wait-free universal construction [32], [33]. Unlike previous universal constructions, our technique is capable of executing dynamic transactions.

In our algorithm, the entries of the operations and results arrays are `TMTypes`, implying that each has its own sequence number. When the sequence number of a function pointer is equal to the sequence number of the corresponding result entry, this signifies that the function was not yet committed. This is the state when operations are published. When the transaction executes, the result of the operation is written to the result entry using a standard transactional write. Therefore, when the result is produced during the *apply* phase, the sequence number of the result will be higher than the sequence number of the function pointer. This indicates that the operation has been committed.

As for read-only transactions, they are similar to the lock-free algorithm. If the read-only operation fails more than n attempts (4 in our implementation) then the operation is

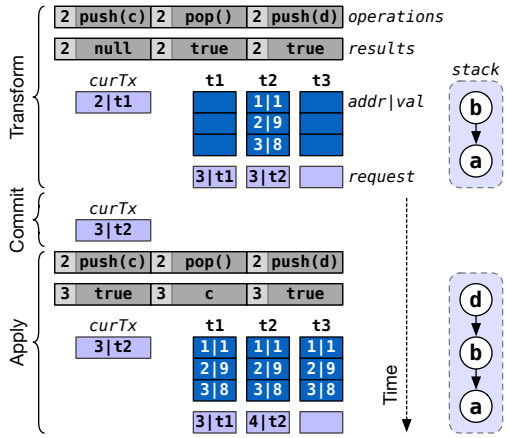

Figure 1: Main components of OneFile wait-free and its 3 phases.

placed on the array of operations as if it was an update, and then attempt to execute two more times. The algorithm guarantees that after $n+2$ failed iterations, one of the threads executing an update transaction has executed the read-only transaction and its result is available in the results array, thus ensuring bounded wait-free progress.

In the OneFile lock-free algorithm threads share their write-set with other threads, while in the wait-free algorithm threads share their transactions by encapsulating them in a `std::function`. This matches the result by [34] which shows that wait-freedom for a TM is not possible unless the code of each transaction is accessible to the other threads. Their result does not apply to lock-freedom.

To allow a better understanding of the algorithm, Fig. 1 represents the main components when a transaction in OneFile Wait-Free is used to provide concurrent access to a sequential implementation of a *stack*. The stack has two methods, `push()` and `pop()`, that are published by each thread in the `operations` array. In this example there was already an execution of a previous transaction that pushed element a and b to the stack. For simplicity we show only 3 threads and each has a `writeSet` that records the modifications required by the update transaction. In this scenario, all three published operations (`push(c)`, `pop()`, and `push(d)`) are grouped inside a transaction simulated by thread $t2$. At the start of the transform phase, the last committed transaction was simulated by thread $t1$ with sequence 2, represented by `curTx`. Once the simulation of the transaction is finished, the `request` field is set to the next sequence concatenated with $t2$ identifier.

The next phase is the commit phase, which will allow the transaction simulated by $t2$ to take effect.

Finally, the apply phase starts the moment the modifications recorded on the write-set are effectively written at the `addr` memory locations. It is then that the element d is added to the stack. At the end of the apply phase, `request` will transition to sequence 4 concatenated with $t2$ identifier, announcing the end of this phase and the end of the transaction whose sequence is 3.

IV. Memory Management

Allocating, de-allocating, tracking and reclaiming objects in dynamic memory are challenging problems, particularly on NVM. Sequential code de-allocates an object immediately after removing it from a data structure. In general, lock-free reclamation requires tracking accesses of other threads to ensure that an object is safe to de-allocate. Many schemes have been proposed over the course of the years [35]–[47]. However, most of these are applicable only to some specific data structures, require the data structure's algorithm to be adapted or rely on blocking code.

Supporting dynamic memory allocation on NVM has to be carefully considered because a PTM must handle non-corrupting failures (crashes). If an object is allocated but not yet inserted into a data structure, it might end up marked as allocated but unreachable by the application, leading to a *permanent* leak in the NVM heap. A similar leak happens if a crash occurs after a object is removed from a data structure but before it is de-allocated [48]. Although blocking techniques exist for reclaiming objects in NVM [7], [48], [49], no lock-free memory reclamation scheme for NVM has been presented in the literature.

A. Allocation and de-allocation in NVM

To support a wait-free, fault-tolerant, simple memory reclamation scheme for OneFile, we use a combination of two ideas: allocating and de-allocating memory inside a transaction and optimistic access to reclaimed objects. All allocations and de-allocations are assumed to be part of a transaction that inserts or removes a node (or object) from a data structure. The destructor is also invoked as part of the transaction. In the OneFile PTMs we use our own sequential implementation of an allocator whose metadata types have been annotated with `TMType`. Other sequential allocator implementations can be used. This design ensures that memory is never leaked during a crash. However, the concurrency problem remains: while one thread de-allocates an object and destructs it, other threads may access the object inappropriately.

To solve the concurrency issue, we use an optimistic technique. The main observation is that when a thread accesses a removed object, its current transaction conflicts with the removal transaction and will abort. It remains to show that the thread aborts its transaction correctly, despite accessing a removed object. Unlike standard virtual memory, NVM memory is not returned to the OS after de-allocation, being instead managed by a PTM-specific user-level allocator. Thus, accessing a de-allocated object does not trigger a page fault. Furthermore, the NVM memory is accessed only through the PTM interface, preserving the `TMType` structure, including the (ever increasing) sequence numbers. Next, we show that either reading from or writing to a de-allocated object results in aborting the transaction, without unexpected side effects.

Proposition 1: *If a thread reads from a `TMType` of a de-allocated memory block, it either reads the value before de-allocation or the transaction aborts without returning the read value.*

Sketch of proof: Let T be a transaction that accessed a field F on a de-allocated memory block and let R be the transaction that de-allocated it. Clearly T started before R committed, since otherwise T would not observe the de-allocated block. According to the read-interposition algorithm (Alg. 1, line 22), T first reads F, then read F's sequence number and aborts if it is higher than the current transaction number. If F was modified during or after de-allocation, then F's sequence number must be higher than T's transaction number (recall that T started before R committed), and T would abort without using the content of F. Notice that the members of the allocated objects are `TMType`s, thus preserving sequence numbers after de-allocation. On the other hand, if F was not modified during or after de-allocation, then the content of F observed by T is unaffected by de-allocation. Thus, the returned content of F represents a snapshot of F before the de-allocation. □

Proposition 2: *A thread never successfully writes to a TMType if this write pertains to a transaction that has already been applied.*

Sketch of proof: The contents of a `TMType` can only be modified by a thread calling the `apply()` method, when executing a DCAS and only if the sequence number of the transaction to which the modification corresponds is higher than the sequence currently in that TMType (Alg. 1, line 14). Consider a transaction T_i that was already applied, the modifications of the `TMType`s pertaining to T_i now have a sequence number that is equal to the sequence of T_i, or higher in the case of `TMType`s modified by subsequent transactions. Any (delayed) thread T attempting to apply one of the modifications pertaining to T_i will fail the DCAS, because the sequence number currently in the `TMType` will be equal to or higher than the sequence of T_i. □

Our memory reclamation scheme relies on internal management of the NVM memory. Despite being de-allocated, a chunk of NVM memory continues to be managed by the NVM allocator and does not lose its sequence numbers. In our technique, the entire NVM region consists of 16-byte aligned `TMType` entries, including all allocator metadata used by our system. In other words, if we were to number each of the 64-bit words in the memory region starting at zero, all even-numbered words would be a *value* and all odd-numbered words would be a *sequence*. The only annotation provided to the user is the `TMType`, which means that all data types must have this annotation for safe concurrent usage and all such object instances must therefore reside in NVM. This constraint in memory allocation ensures correctness when a memory block is re-allocated, *i.e.*, the allocator reuses a memory region previously occupied.

Proposition 3: *A thread never successfully writes to a de-allocated object.*

Sketch of proof: Let T be a transaction that writes to a `TMType` field F of a de-allocated object and let R be the transaction that de-allocated the object. If T is attempting a write to the field F, then T must be applying the modifications from a transaction Q that committed before R. Otherwise, T cannot observe the de-allocated object. Since the object was de-allocated, clearly R was committed successfully. Therefore, Q must have already been applied, and by proposition 2 the DCASes executed by T in Alg. 1 will fail because the sequence number in F was already increased when Q was applied. □

B. Closure Reclamation

On the wait-free algorithm, each thread publishes its transaction as a function that other threads can execute, a `std::function`. The function is stored as a stream of executable bytes and might not be anymore executable after rebooting the machine. Furthermore, the executable byte stream cannot be stored in `TMTypes`, so this function must be allocated in the transient memory. But this object must also be de-allocated, despite being accessed by (possibly many) concurrent threads.

To manage transient memory without foiling the wait-freedom guarantee of our algorithm and without introducing a high overhead, we use the *hazard eras* (HE) [41] algorithm. With HE, each thread publishes an *era*. All objects that were alive during this era cannot be reclaimed by concurrent threads since they might be accessed by the publisher thread. An object can only be reclaimed if the period of time during which it was alive does not overlap with the currently published era of a thread.

The HE scheme inter-operates well with the OneFile algorithm, utilizing the transaction number (the sequence number of `curTx`) as the era number. The thread publishes the transaction number it is using at the beginning of the transaction. According to the HE scheme, every object that is still accessible during this era (*i.e.*, still accessible during this transaction) must not be reclaimed. Objects that are not accessible in the current era (*i.e.*, by the current transaction) of any thread are reclaimed by the HE scheme. When a thread reads a new pointer to an object, it first checks whether it was installed by a newer era, in which case it is unsafe to access since the object might have been reclaimed. But reading a pointer published in a newer era (*i.e.*, by a newer transaction) also breaks the isolation property of the transaction. Thus, the load-interposition algorithm used by OneFile is also sufficient to ensure that no reclaimed memory is accessed.

The HE scheme guarantees wait-freedom for the reclamation procedure, but only lock-freedom for reading a pointer from the heap. After a new pointer is read, the HE scheme checks if the era was changed and, if so, re-reads the pointer. Thus, a thread may starve reading a pointer while other threads progress to new eras. Still, the helping mechanism we designed for OneFile guarantees wait-freedom also for the HE scheme. A thread is unable to progress only if a new era was announced. However, after two attempts, other threads must have executed the current operation on behalf of the current thread, thus ensuring progress for any thread in the system. To our knowledge, this is the first wait-free algorithm supporting full memory reclamation.

V. Evaluation

We now present a detailed evaluation of OneFile, divided in two subsections, volatile (transient) and non-volatile memory. Except for the latency plots in Fig. 7, all other microbenchmarks were executed on an AWS c5.9xlarge instance with 36 virtual cores, running Ubuntu LTS and using `gcc` 7.3 with the `-O3` optimization flag. This instance is hosted on a 3 GHz Intel Xeon (Skylake-SP) Platinum 8124M and it supports the new `CLWB` instruction as `pwb`.

A. Evaluation of Volatile Memory techniques

The only other known lock-free STM with dynamic transactions has been implemented in the JVM [17], for which there is no DCAS operation, making unfeasible a direct comparison with OneFile. Moreover, memory reclamation on the JVM is not lock-free. As such, we compare our STM implementations with lock-based (blocking) STMs, choosing two well known, fast, and easy to use STM implementations: ESTM [20] and TinySTM [14], [15].

To act as a *baseline*, we also evaluate lock-free data structures where appropriate. For the linked list based queues, we compare with Michael and Scott's lock-free queue [50], with two wait-free queues SimQueue [33] and Turn Queue [51]. For the array based queues, we compare with Morrison and Afek's lock-free queue [52] based on DCAS named LCRQ, and with the FAA lock-free queue based on single-word CAS by Correia and Ramalhete [53]. For the linked list set, we compare with Michael's lock-free linked list set [54] based on Harris [55]. For trees, we compare with the lock-free tree by Natarajan and Mittal [56] shown as NataHE, a relaxed tree. We chose this relaxed (non-balanced) tree because despite the existence of balanced lock-free trees in the literature [57], no implementation with hazard pointers or other lock-free memory reclamation schemes has ever been shown. For fair comparison with the STMs, all these hand-made data structures have integrated lock-free memory reclamation, with SimQueue and Turn queue having wait-free memory reclamation. We chose the fastest memory reclamation for each data structure, using Hazard Pointers [36] for the queues, and Hazard Eras [41] for all other data structures.

We start our evaluation with the SPS benchmark [11], [58]–[60] which consists of an array of 64bit integers with 1,000 entries, where random *swaps* of integer values are

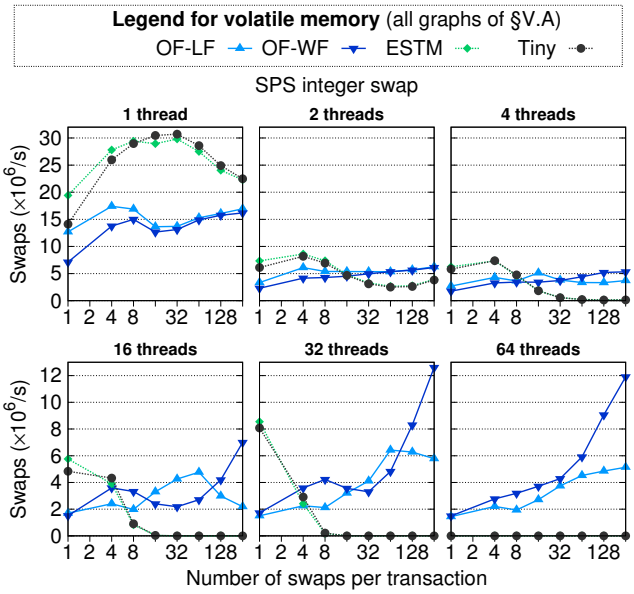

Figure 2: SPS integer microbenchmark for multiple STMs.

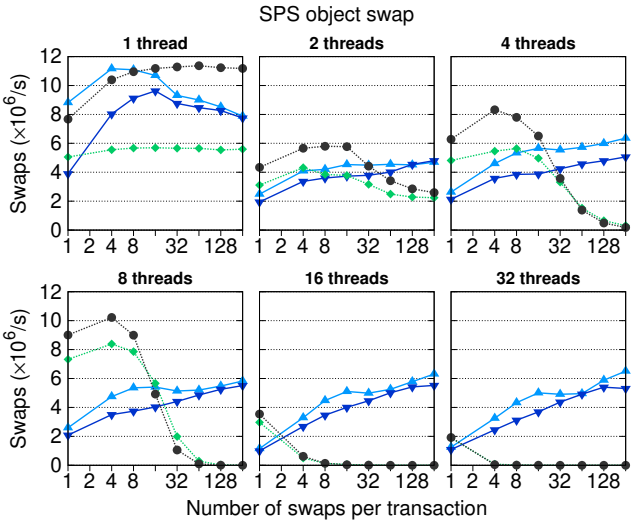

Figure 3: SPS object microbenchmark for multiple STMs.

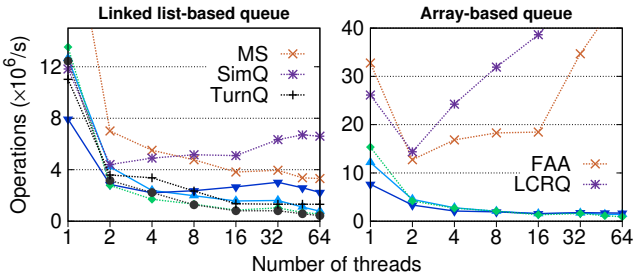

Figure 4: Linked list- and array-based queues.

executed. On Fig. 2 we plot the number of swaps per second as a function of how many swaps are done per transaction. As the number of threads and the transaction size increases, there will be more conflicts between transactions. This benchmark simulates a workload with no memory allocation being done during its execution. When the transaction size is small, the workload is mostly disjoint although as the transaction size increases, the probability of having conflicting

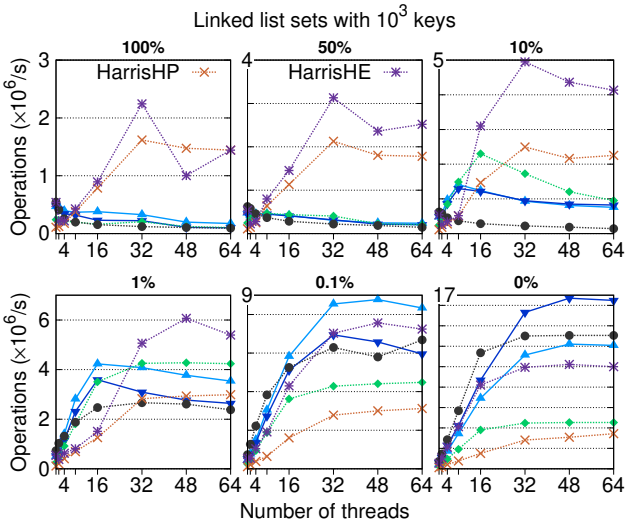

Figure 5: Linked list-based sets with 10^3 keys.

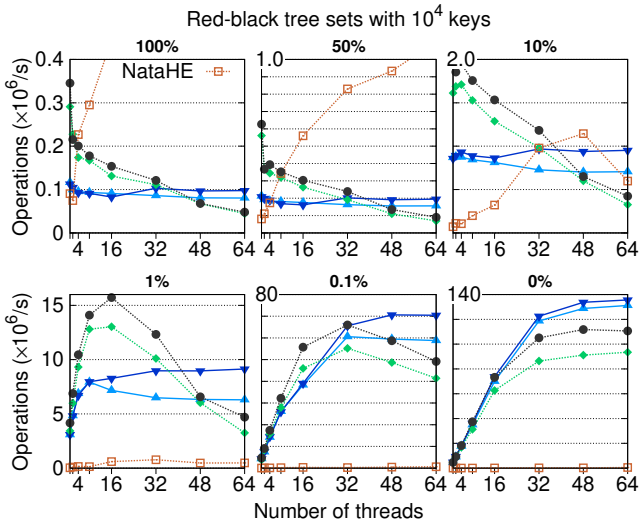

Figure 6: Red-black tree sets with 10^4 keys.

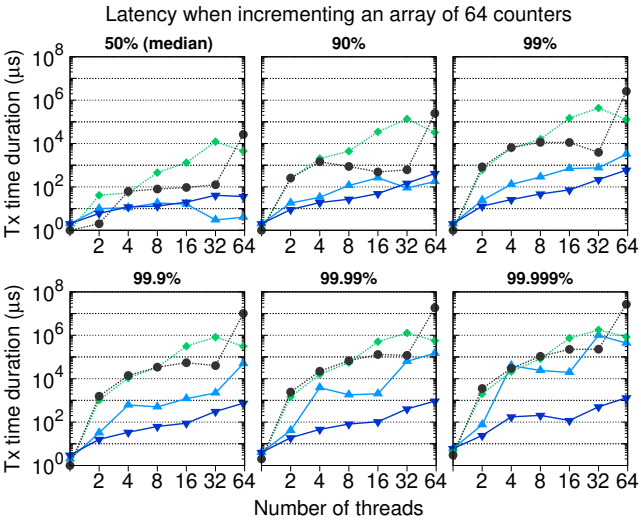

Figure 7: Latency distribution percentiles (lower is better).

transactions increases. ESTM and TinySTM perform well when contention is very low, i.e., a single threaded execution or a few swaps per transaction. However, for large transactions, as contention increases, their performance drops while our OneFile performs well, particularly OneFile wait-free because it aggregates multiple operations, thus reducing the average number of writes (DCAS) per transaction.

We also include a variant of the SPS microbenchmark which allocates memory. Every entry in the array points to a small object containing two `TMType<int64_t>`. A swap of two entries in this array allocates a new object, installs the new pointer, and de-allocates the previous object. Results are shown in Fig. 3. The OneFile algorithms perform well when memory allocation is involved. In contrast, performance of both TinySTM and ESTM reduces, making OneFile favorable even when contention is lower than the SPS microbenchmark without allocations.

In Fig. 4 we depict results for the queue data structure: linked-list based (left side) and array-based (right hand side). We executed 10^8 pairs of enqueues and dequeues for each data point. OneFile performs better than both ESTM and TinySTM for the linked list based, but slower than ad-hoc algorithms. Still, OneFile allows for linearizable traversals of the queue, is significantly simpler to design, maintain, customize, and use, making it a reasonable choice in many practical cases.

The following plots compare different *set* data structures, with the update ratio being shown at the top of each graph. An update operation is composed of two consecutive transactions, a removal followed by an insertion, whereas a read operation is composed of two consecutive read-only transactions, each executes a search for an existing random key.

Fig. 5 shows linked list sets under six different update ratio workloads. In the 100% update workload (top left-most plot), the OneFile beat the other STMs and the Harris lock-free implementations for the 2 threads and 4 threads scenarios. As the number of threads increases, the Harris lists are capable of doing more concurrent insertion and removal operations, allowing them to scale, giving them advantage over all the others. As the ratio of read-only operations increases, the advantage of OneFile extends up to a larger number of threads. Due to not having a read-set, the OneFile STMs excel in the read-mostly workloads where they can match or surpass the hand-made lock-free linked lists [54].

With a tree with 10^4 keys (Fig. 6), OneFile has several scenarios where it surpasses the lock-free tree [56], namely with low thread count or in over-subscription.

Fig. 7 shows six different percentiles of the latency distribution, when running on a high-core count CPU, namely, a dual-socket 2.10 GHz Intel Xeon E5-2683 ("Broadwell") with a total of 32 hyper-threaded cores (64 HW threads). The plots are in log-log scale, with the vertical axis representing the time in microseconds it takes to complete a transaction. In this microbenchmark there is an array of 64 counters where each transaction increments all of the counters, alternating between incrementing the counters starting from left to right, and on the next transaction incrementing right to left. This workload implies a strong serialization of the transactions and causes most STMs to have starvation effects. For example, on the 90% percentile plot, for 2 threads, with the wait-free OneFile 90% of the transactions take 9 microseconds or less to complete, while with ESTM take 243 microseconds or less to complete, and with Tiny STM take 848 microseconds or less to complete. The wait-free OneFile STM has a significant advantage at the tail of the distribution, having at least 100× lower (better) latency from the 99.9% percentile onwards, as soon two or more threads attempts to execute update transactions, reaching a peak improvement of 1,000× over ESTM and 10,000× over TinySTM. These results highlight the importance of wait-freedom for tail latency, a relevant characteristic for network operating systems and other soft real-time applications.

B. Evaluation of NVM techniques

For fairness and practicality reasons, we did not evaluate PTMs that require specialized hardware [10], nor did we consider PTMs that do not provide durable linearizable transactions [61], [62]. Comparisons were made with PMDK (libpmemobj++), RomulusLog and RomulusLR, described in §II. NVM was emulated by allocating a file in the `/dev/shm` directory that is mounted on DRAM.

We executed the persistent variant of the SPS [11], [58]–[60] benchmark mentioned previously. This benchmark allows us to understand the performance profile of the PTMs under different transaction sizes. Each swap exchanges the values of two randomly selected entries of an array of one million 64-bit integers, hence modifying two memory words in PM. As shown in Fig. 8, the lock-free OneFile-PTM surpasses the other PTMs for large transactions with over-subscription.

We implemented three persistent sets using the above PTMs, namely, a singly linked list set, a red-black tree set, and a hash set, shown in Fig. 9, 10 and 11 respectively.

The throughput of the OneFile PTMs is penalized due to the load interposition. On a singly linked list data structure, most of the time is spent traversing the list, imposing a call to the `load()` interposed method for every traversed node. On RomulusLog and PMDK this is implemented as a regular load, while on RomulusLR it contains a check for which memory region to traverse and adjust the regular load accordingly, both approaches being extremely fast. For the OneFile PTMs the load interposition is significantly more complex, requiring two acquire-loads and possibly a lookup on the write-set, affecting the overall throughput. Despite this disadvantage, OneFile is capable of beating RomulusLog and RomulusLR for the majority of the runs

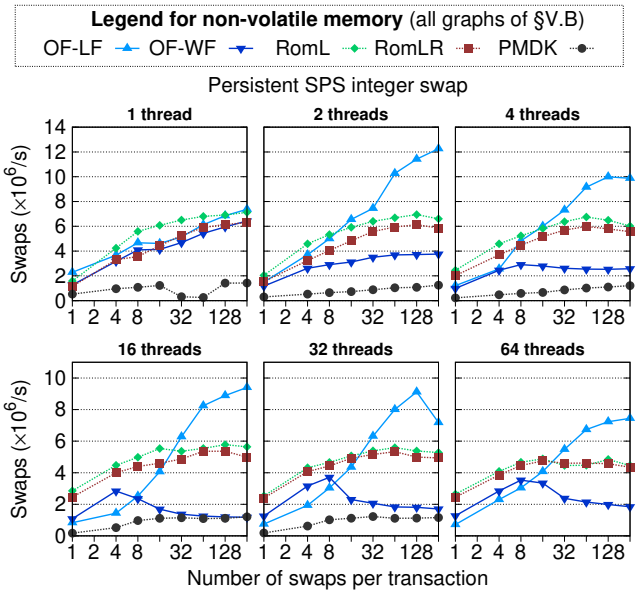

Figure 8: Persistent SPS integer microbenchmark.

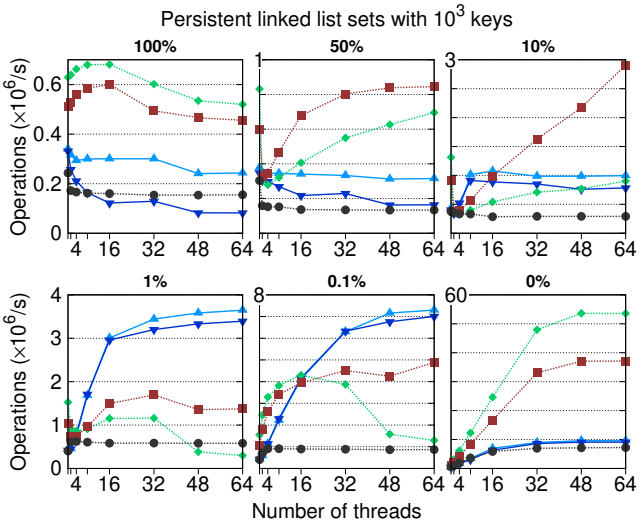

Figure 9: Persistent linked list-based sets with 10^3 keys.

in the 1% and 0.1% workloads, due to the non-blocking progress. The OneFile PTMs overtake PMDK on nearly all workloads.

On update transactions, due to the overhead on redo-log techniques added by the lookup in the log, traversing less nodes implies a smaller overhead. On a balanced tree filled sequentially with one million keys, seen in Fig. 10, the number of traversed nodes is ∼20, with the OneFile PTMs having high enough throughput to overtake PMDK in all scenarios. The OneFile PTMs match and slightly exceed the Romulus variants for the 0% update workload, and surpass Romulus in over-subscription for the 0.1% workload. This happens because in the Romulus variants, in over-subscription, a thread holding the lock may be preempted, blocking progress for all other threads, which does not happen in OneFile due to the lock-free progress.

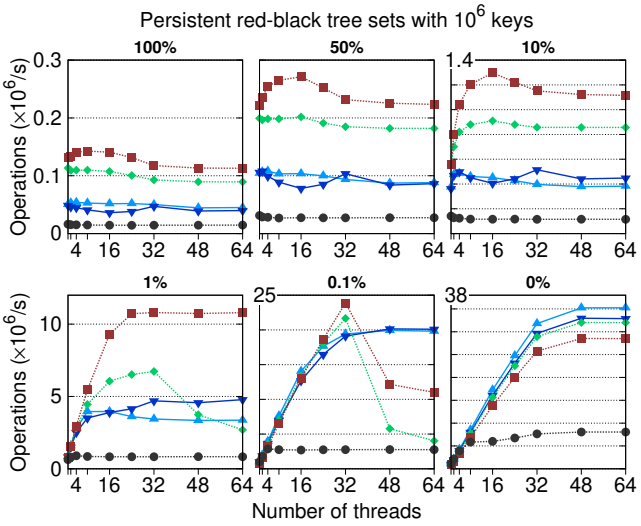

Figure 10: Persistent red-black tree sets with 10^6 keys.

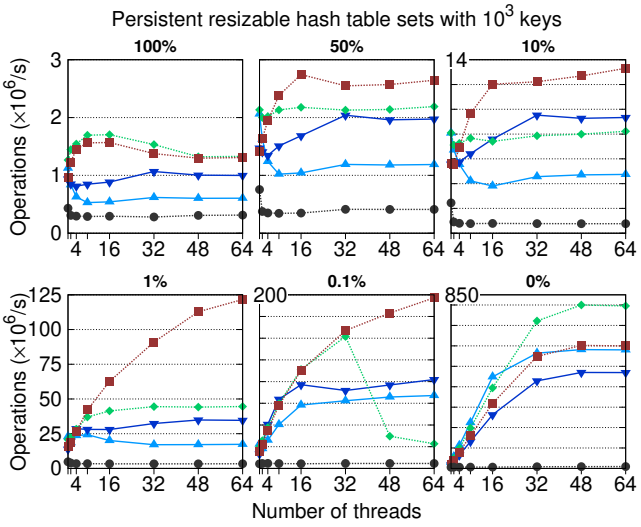

Figure 11: Persistent hash table sets with 10^3 keys.

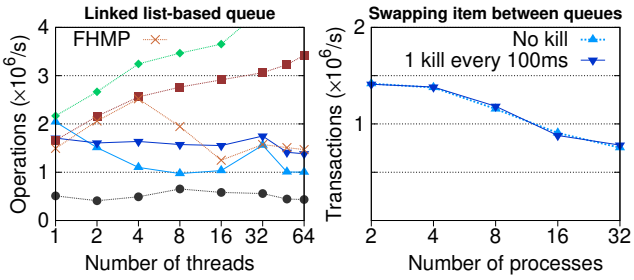

Figure 12: Persistent queues (left) and impact of failures (right).

On the hash set shown in Fig. 11, the number of traversed nodes is rarely above 2, bringing OneFile even closer to the Romulus implementations and again surpassing PMDK by an order of magnitude in most scenarios. For an update ratio of 100% the OneFile-PTM performs 30% less than RomulusLog but on the other hand, provides lock-free progress.

On the left plot of Fig. 12 we show multiple persistent queues, all singly-linked list based. At the exception of the

FHMP queue, which is the hand-made lock-free designed by Friedman, Herlihy, Marathe and Petrank [63], all other queues are sequential implementations wrapped in a PTM. The original design for the FHMP queue uses the system allocator which means that it is blocking for allocation, has no embedded memory reclamation (nodes are never deleted and the memory will eventually fill up), and the allocator does not work on NVM. Up until now, no NVM allocator has been presented with lock-free progress, and neither has a lock-free memory reclamation scheme. Seen as FHMP does not allocate in persistent memory, all `pwb` and `pfence` associated with memory allocation/de-allocation are omitted during its execution. If an NVM allocator and reclamation scheme is added to FHMP, its performance will decrease. The queues made with the OneFile PTMs have a performance close to the hand made lock-free queue FHMP and surpass it in single thread workloads, while providing failure-resilience memory allocation/de-allocation and reclamation.

In addition, the advantage of lock-free PTMs over hand-made lock-free data structures is relevant when having operations over multiple instances. Consider two instances of a lock-free queue designed for NVM, `q1` and `q2`. If the user wants to dequeue an item `x` from `q1` and place it in `q2`, with lock-free progress and in an *atomic* way, there is no easy way to do it. If a failure occurs after the dequeue of item `x` from `q1` and before the enqueue of `x` on `q2`, upon restart, the contents of `q1` and `q2` will be recovered to consistent state, with neither of the queues having the item `x`. In this case, the item `x` will be effectively *lost*. With OneFile-PTM the user can create a transaction that encompasses the dequeue from `q1` and the enqueue in `q2`, thus preventing the loss of the item in the event of a failure. Moreover, as memory allocation and reclamation are part of the transaction, in the event of a failure there will be no memory leakage or allocator metadata corruption when removing the node in `q1` and creating a new node in `q2`.

Based on this scenario, we implemented a test where multiple processes execute a transaction which modifies two shared queues, while one of these processes is randomly killed. The test consists of executing N processes each with a single thread. Its thread continually executes a transaction that takes an item out of a queue and places the item in another queue. There are two queues in persistent memory, shared among the N processes. Enqueuing allocates one node of the queue while dequeueing de-allocates another node, per transaction. This test executes during 100 seconds, with N being 2, 4, 8, 16 or 32 processes and the number of transactions per second are shown on the right-side plot of Fig. 12 as *no kill*. We then repeated the same test, but with a script that randomly kills one of the N processes, every 100 milliseconds and immediately re-spawns a similar process, also shown on the right plot of Fig. 12. This means there are 1,000 failures for the duration of the test.

From these results we can conclude the following: due to the null-recovery property, OneFile provides fast recovery time, with no measurable performance impact for the tested scenario of 1,000 failures during 100 seconds; we never observed memory leaks nor allocator metadata corruption and no breakage of application invariants either; and the lack of performance difference with and without processes being killed shows that OneFile is resilient to failures, allowing the non-failed processes to continue executing normally.

To help understand the factors that impact the performance of the different PTMs in §V, we summarize in the following table the number of `pwb`, `pfence` and synchronization primitives on update transactions, as a function of the number of modified words N_w in a transaction.

	`pwb`	`pfence`	CAS or DCAS
PMDK	2.25 N_w	2 + 2 N_w	1
RomulusLog	3 + 2 N_w	4 or less	1
OF (Lock-Free)	1 + 1.25 N_w	0	2 + N_w
OF (Wait-Free)	2 + 1.25 N_w	0	3 + N_w

VI. Conclusion

STMs have a bad reputation for mishandling large transactions and high contention, and deservingly so. The wait-free OneFile is immune to starvation and has predictable latency, regardless of contention or transaction size.

Transactions in OneFile-PTM require two persistence fences, are durable linearizable, and have bounded wait-free progress including memory allocation, de-allocation and reclamation. The novel technique for memory reclamation in OneFile-PTM uses an optimistic approach that allows to safely access memory locations with no need to announce its access to concurrent threads. Furthermore, a transaction with a de-allocation and allocation of an object of the same size can re-use the same memory block, which reduces the number of `pwb`s and improves cache locality. This immediate re-usage of blocks is not possible on previously known lock-free memory reclamation schemes.

So far in the current literature, a single lock-free data structure has been shown for NVM, a lock-free queue [63]. We provide a generic approach with integrated memory reclamation. With OneFile-PTM we have implemented for NVM, a wait-free queue, a wait-free linked list set, a wait-free resizable hash map, and a wait-free balanced tree. And other containers can be implemented.

The initial intent of Herlihy and Moss's paper [4] was to propose Transactional Memory as a simple way of producing lock-free data structures. It took more than 25 years but finally, an efficient and easy to use lock-free STM is now available for users to develop their own lock-free and wait-free data structures. Furthermore, OneFile-PTM goes above this goal by giving end users the ability to design their own failure-resilient wait-free data structures for NVM, enabling them to create reliable applications with fast wait-free ACID transactions with serializable isolation.

REFERENCES

[1] E. W. Dijkstra, "Cooperating sequential processes," in *The origin of concurrent programming*. Springer, 1968, pp. 65–138.

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

[3] M. Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 1, pp. 124–149, 1991.

[4] M. Herlihy and J. E. B. Moss, *Transactional memory: Architectural support for lock-free data structures*. ACM, 1993, vol. 21, no. 2.

[5] N. Shavit and D. Touitou, "Software transactional memory," *Distributed Computing*, vol. 10, no. 2, pp. 99–116, 1997.

[6] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1. ACM, 2011, pp. 91–104.

[7] A. Correia, P. Felber, and P. Ramalhete, "Romulus: Efficient algorithms for persistent transactional memory," in *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*. ACM, 2018, pp. 271–282.

[8] J. Izraelevitz, H. Mendes, and M. L. Scott, "Linearizability of persistent memory objects under a full-system-crash failure model," in *International Symposium on Distributed Computing*. Springer, 2016, pp. 313–327.

[9] D. E. Lowell and P. M. Chen, "Free transactions with rio vista," in *ACM SIGOPS Operating Systems Review*, vol. 31, no. 5. ACM, 1997, pp. 92–101.

[10] J. Izraelevitz, T. Kelly, and A. Kolli, "Failure-atomic persistent memory updates via justdo logging," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 2, pp. 427–442, 2016.

[11] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories," *ACM Sigplan Notices*, vol. 46, no. 3, pp. 105–118, 2011.

[12] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging locks for non-volatile memory consistency," *ACM SIGPLAN Notices*, vol. 49, no. 10, pp. 433–452, 2014.

[13] PMDK team, "Persistent Memory Programming," http://pmem.io, 2018.

[14] P. Felber, C. Fetzer, P. Marlier, and T. Riegel, "Time-based software transactional memory," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 12, pp. 1793–1807, 2010.

[15] P. Felber, C. Fetzer, and T. 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*. ACM, 2008, pp. 237–246.

[16] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir, "Flat combining and the synchronization-parallelism tradeoff," in *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2010, pp. 355–364.

[17] S. M. Fernandes and J. Cachopo, "Lock-free and scalable multi-version software transactional memory," in *ACM SIGPLAN Notices*, vol. 46, no. 8. ACM, 2011, pp. 179–188.

[18] J. Cachopo and A. Rito-Silva, "Versioned boxes as the basis for memory transactions," *Science of Computer Programming*, vol. 63, no. 2, pp. 172–185, 2006.

[19] C. Fu, Z. Wu, X. Wang, and X. Yang, "A review of software transactional memory in multicore processors," *Information Technology Journal*, vol. 8, no. 8, pp. 1269–1274, 2009.

[20] P. Felber, V. Gramoli, and R. Guerraoui, "Elastic transactions," in *International Symposium on Distributed Computing*. Springer, 2009, pp. 93–107.

[21] D. Dice, O. Shalev, and N. Shavit, "Transactional locking ii," in *International Symposium on Distributed Computing*. Springer, 2006, pp. 194–208.

[22] T. L. Harris, K. Fraser, and I. A. Pratt, "A practical multi-word compare-and-swap operation," in *International Symposium on Distributed Computing*. Springer, 2002, pp. 265–279.

[23] K. Fraser and T. Harris, "Concurrent programming without locks," *ACM Transactions on Computer Systems (TOCS)*, vol. 25, no. 2, p. 5, 2007.

[24] M. Pavlovic, A. Kogan, V. J. Marathe, and T. Harris, "Brief announcement: Persistent multi-word compare-and-swap," in *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*. ACM, 2018, pp. 37–39.

[25] T. Wang, J. Levandoski, and P.-A. Larson, "Easy lock-free indexing in non-volatile memory," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 2018, pp. 461–472.

[26] A. Israeli and L. Rappoport, "Disjoint-access-parallel implementations of strong shared memory primitives," in *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*. ACM, 1994, pp. 151–160.

[27] H. Sundell, "Wait-free multi-word compare-and-swap using greedy helping and grabbing," *International Journal of Parallel Programming*, vol. 39, no. 6, pp. 694–716, 2011.

[28] S. Feldman, P. LaBorde, and D. Dechev, "A practical wait-free multi-word compare-and-swap operation," in *Many-Core Architecture Research Community (MARC) Symposium at SPLASH*, vol. 2013. Citeseer, 2013.

[29] S. Doherty, D. L. Detlefs, L. Groves, C. H. Flood, V. Luchangco, P. A. Martin, M. Moir, N. Shavit, and G. L. Steele Jr, "Dcas is not a silver bullet for nonblocking algorithm design," in *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2004, pp. 216–224.

[30] S. R. Dulloor, "Systems and applications for persistent memory," Ph.D. dissertation, Georgia Institute of Technology, 2015.

[31] N. Cohen, M. Friedman, and J. R. Larus, "Efficient logging in non-volatile memory by exploiting coherency protocols," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, p. 67, 2017.

[32] M. Herlihy, "A methodology for implementing highly concurrent data structures," in *ACM SIGPLAN Notices*, vol. 25, no. 3. ACM, 1990, pp. 197–206.

[33] P. Fatourou and N. D. Kallimanis, "A highly-efficient wait-free universal construction," in *Proceedings of the twenty-*

third annual ACM symposium on Parallelism in algorithms and architectures. ACM, 2011, pp. 325–334.

[34] V. Bushkov and R. Guerraoui, "Liveness in transactional memory," in *Transactional Memory. Foundations, Algorithms, Tools, and Applications*. Springer, 2015, pp. 32–49.

[35] P. E. McKenney and J. D. Slingwine, "Read-copy update: Using execution history to solve concurrency problems," in *Parallel and Distributed Computing and Systems*, 1998, pp. 509–518.

[36] M. M. Michael, "Hazard pointers: Safe memory reclamation for lock-free objects," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 6, pp. 491–504, 2004.

[37] M. Herlihy, V. Luchangco, and M. Moir, "The repeat offender problem: A mechanism for supporting dynamic-sized lock-free data structures," 2002.

[38] A. Braginsky, A. Kogan, and E. Petrank, "Drop the anchor: lightweight memory management for non-blocking data structures," in *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2013, pp. 33–42.

[39] N. Cohen and E. Petrank, "Efficient memory management for lock-free data structures with optimistic access," in *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*. ACM, 2015, pp. 254–263.

[40] ——, "Automatic memory reclamation for lock-free data structures," in *ACM SIGPLAN Notices*, vol. 50, no. 10. ACM, 2015, pp. 260–279.

[41] P. Ramalhete and A. Correia, "Brief announcement: Hazard eras-non-blocking memory reclamation," in *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 2017, pp. 367–369.

[42] D. Alistarh, P. Eugster, M. Herlihy, A. Matveev, and N. Shavit, "Stacktrack: An automated transactional approach to concurrent memory reclamation," in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 25.

[43] T. A. Brown, "Reclaiming memory for lock-free data structures: There has to be a better way," in *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*. ACM, 2015, pp. 261–270.

[44] D. Alistarh, W. M. Leiserson, A. Matveev, and N. Shavit, "Threadscan: Automatic and scalable memory reclamation," in *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*. ACM, 2015, pp. 123–132.

[45] D. Dice, M. Herlihy, and A. Kogan, "Fast non-intrusive memory reclamation for highly-concurrent data structures," in *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*. ACM, 2016, pp. 36–45.

[46] H. Wen, J. Izraelevitz, W. Cai, H. A. Beadle, and M. L. Scott, "Interval-based memory reclamation," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2018, pp. 1–13.

[47] M. Pöter and J. L. Träff, "Stamp-it: A more thread-efficient, concurrent memory reclamation scheme in the c++ memory model," *arXiv preprint arXiv:1805.08639*, 2018.

[48] K. Bhandari, D. R. Chakrabarti, and H.-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*. ACM, 2016, pp. 677–694.

[49] T. David, A. Dragojevic, R. Guerraoui, and M. I. Zablotchi, "Log-free concurrent data structures," Tech. Rep., 2017.

[50] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. ACM, 1996, pp. 267–275.

[51] P. Ramalhete and A. 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*. ACM, 2017, pp. 453–454.

[52] A. Morrison and Y. Afek, "Fast concurrent queues for x86 processors," in *ACM SIGPLAN Notices*, vol. 48, no. 8. ACM, 2013, pp. 103–112.

[53] P. Ramalhete and A. Correia, "Faaarrayqueue — mpmc lock-free queue," http://www.concurrencyfreaks.com/2016/11/faaarrayqueue-mpmc-lock-free-queue-part.html, 2016.

[54] M. M. Michael, "High performance dynamic lock-free hash tables and list-based sets," in *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*. ACM, 2002, pp. 73–82.

[55] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *International Symposium on Distributed Computing*. Springer, 2001, pp. 300–314.

[56] A. Natarajan and N. Mittal, "Fast concurrent lock-free binary search trees," in *ACM SIGPLAN Notices*, vol. 49, no. 8. ACM, 2014, pp. 317–328.

[57] T. Brown, F. Ellen, and E. Ruppert, "A general technique for non-blocking trees," in *ACM SIGPLAN Notices*, vol. 49, no. 8. ACM, 2014, pp. 329–342.

[58] Y. Lu, J. Shu, and L. Sun, "Blurred persistence in transactional persistent memory," in *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*. IEEE, 2015, pp. 1–13.

[59] Y. Lu, J. Shu, L. Sun, and O. Mutlu, "Loose-ordering consistency for persistent memory," in *Computer Design (ICCD), 2014 32nd IEEE International Conference on*. IEEE, 2014, pp. 216–223.

[60] V. Gogte, S. Diestelhorst, W. Wang, S. Narayanasamy, P. M. Chen, and T. F. Wenisch, "Persistency for synchronization-free regions." PLDI, 2018.

[61] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. 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*. ACM, 2017, pp. 329–343.

[62] A. Memaripour, A. Badam, A. Phanishayee, Y. Zhou, R. Alagappan, K. Strauss, and S. Swanson, "Atomic in-place updates for non-volatile main memories with kamino-tx," in *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 2017, pp. 499–512.

[63] M. Friedman, M. Herlihy, V. Marathe, and E. 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*. ACM, 2018, pp. 28–40.