Romulus: Efficient Algorithms for Persistent Transactional Memory

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

Byte addressable persistent memory eliminates the need for serialization and deserialization of data, to and from persistent storage, allowing applications to interact with it through common store and load instructions. In the event of a process or system failure, applications rely on persistent techniques to provide consistent storage of data in non-volatile memory (NVM). For most of these techniques, consistency is ensured through logging of updates, with consequent intensive cache line flushing and persistent fences necessary to guarantee correctness. Undo log based approaches require store interposition and persistence fences before each in-place modification. Redo log based techniques can execute transactions using just two persistence fences, although they require store and load interposition which may incur a performance penalty for large transactions. So far, these techniques have been difficult to integrate with known memory allocators, requiring allocators or garbage collectors specifically designed for NVM.

We present Romulus, a user-level library persistent transactional memory (PTM) which provides durable transactions through the usage of twin copies of the data. A transaction in Romulus requires at most four persistence fences, regardless of the transaction size. Romulus uses only store interposition. Any sequential implementation of a memory allocator can be adapted to work with Romulus. Thanks to its lightweight design and low synchronization overhead, Romulus achieves twice the throughput of current state of the art PTMs in update-only workloads, and more than one order of magnitude in read-mostly scenarios.

CCS CONCEPTS

• **Theory of computation** → *Concurrent algorithms*; • **Computer systems organization** → *Reliability*;

KEYWORDS

Persistent memory; crash resilience; failure atomicity; transactions; concurrency

ACM Reference Format:
Andreia Correia, Pascal Felber, and Pedro Ramalhete. 2023. Romulus: Efficient Algorithms for Persistent Transactional Memory. In *Proceedings of (This is the author's version of the work. It is posted here for your personal use. The definitive Version of Record was published in SPAA'18 http://dx.doi.org/10.1145/3210377.3210392).* ACM, New York, NY, USA, 12 pages.

This is the author's version of the work. It is posted here for your personal use. The definitive Version of Record was published in SPAA'18 http://dx.doi.org/10.1145/3210377.3210392, 2018
2023.

1 INTRODUCTION

Persistent mechanisms can provide resilience against power outages and non-corrupting software failures. The advent of storage-class memory or non-volatile memory (NVM) as a commercial product has brought this research to the front stage, allowing data and data structures to be directly persisted to NVM, avoiding serialization and deserialization of data, as is typically done for block devices. Non-volatile memory constitutes an opportunity to build applications that require resilience to failures with improved performance. Persistent application data is not guaranteed to be stored in a consistent state. A consistent state must satisfy application-level invariants or other correctness criteria.

To guarantee consistent persistence, in case of a failure during a modification of application data, one must ensure that only the following two scenarios can occur: either the modification is persisted in its entirety, or application data is reverted to the previous consistent state. This behavior of *all or nothing* fits naturally into the concept of a transaction.

Prior research on this topic has identified the same behavior and expose a persistent transactional memory (PTM) interface to the end user. Most of these PTMs use an underlying software transactional memory (STM), thus providing safe concurrent access in addition to durable transactions. These transactions are sometimes referred in the literature as failure-atomic-sections (FASEs).

To enable transactions, previous works on transactional persistence employ one of two logging techniques: write-ahead logging (WAL) with **undo** [7, 16, 20] or **redo** [16, 31] log. Both techniques require a log, containing the new value (redo log) or the current value (undo log) to be persisted. This adds complexity, 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. Another approach is to use copy-on-write, which consists in copying the data in its existing state, applying the modifications to the new copy, and finally replacing the existing state with the new one with a single atomic operation. This technique is commonly used for disk storage and has also been considered for NVM [9, 26, 30]. Due to the lengthy copy procedure, the copy-on-write technique has not been deemed to be a viable alternative to in-place modifications via logging [1].

In this paper we present Romulus, a novel user-space library for consistent application recovery from a non-corrupting failure, which allows application developers to take advantage of byte-addressable non-volatile memory, with safe memory reclamation. Romulus was developed to be fast, lightweight and simple to use. Its code is written in processor-agnostic C++ and integrates a custom persistent memory allocator.

In short, with Romulus we make the following contributions.

- We introduce a novel PTM algorithm that relies on two copies of the data to be persisted, with three different algorithms that provide different trade-offs in terms of performance and sophistication. Unlike previous approaches, Romulus maintains a redo log in volatile memory instead of persisting it, hence significantly reducing overheads on log accesses. Romulus also allows concurrent access through a highly scalable reader-writer lock and a flat-combining array.
- We specifically address concurrency with a second implementation of Romulus that guarantees wait-free progress for read operations and blocking starvation-free update operations. Concurrent read-only transactions use lightweight synchronization mechanisms and are very efficient.
- Romulus performance is unprecedented with 4 persistence fences per transaction, independent of the transaction size and the number of threads. Another source of potential delay are the number of cache line flushes (pwb), which were also studied and significantly reduced.
- We formally present the different variants of the Romulus algorithm, study and discuss their properties, and give proofs of their correctness.
- Finally, we evaluate experimentally its performance in real-world use cases. In particular, we develop a fully-functional persistent key-value store, RomulusDB, and test it with state-of-the-art benchmarks.

The rest of the paper is organized as follows. We first discuss related work in §2. We then present background concepts in §3 before introducing the basic version of the Romulus algorithm in §4. We extend the original algorithm in §5 to efficiently handle concurrent transactions. We provide an in-depth evaluation of Romulus in §6 and finally conclude in §7.

2 RELATED WORK

Vista [20] was the first system to provide consistent persistent memory, using the Rio file cache. Vista does in-place modifications and guarantees consistent recovery through an undo log stored in persistent memory. To use transactions in Vista, the user must manually interpose every store to persistent memory. For every range of stores, the current value of those memory locations are saved in the log, requiring one persistence fence per range of stores. Each log entry in Vista uses at least three words: one word for the address, one word for the length of the range, and one word for the original value. Although not originally designed for NVM, Vista can be adapted to use it.

Atlas [4] also uses a WAL with an undo log. An entry in the undo log is created 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, which implies one persistence fence per log entry.

JustDo [16] uses WAL with a per-thread redo log that saves the last store and program counter. A failure recovery will re-execute the last store in the log, resuming the program at the exact point where it has stopped execution. The persistent store has to be idempotent, so that if the store is re-executed, it will result in the same modification on the destination address. This approach assumes that both the main memory and cache local memory are persistent. It requires that all memory loads and stores within FASEs access only persistent data, where a FASE, as in Atlas, encompasses the outermost critical sections of all acquired locks (nested or hand-over-hand). The log has two fixed entries, only one of which is active at a time, each contains the destination address of the store, the size of the write, and the value to be placed at the destination. An extra 64-bit variable holds the program counter with one bit indicating which of the two fixed entries is active. During a FASE, a persistence fence is needed after saving the new value and destination address, and another after saving the program counter.

NV-Heaps [7] uses an undo log system that provides extra functionality, like garbage collection and pointer safety, managing references from volatile to non-volatile memory and vice versa. It aims at facilitating the programmer's task of using persistent memory.

Mnemosyne [31] combines a redo log and the TinySTM [25] underlying STM. For every modified word it uses 8 words on the persisted log. The persistent data is modified only at commit time, which means that during a transaction, the data of variables stored in memory still contains unmodified data. When called to read data from persistence by the user code, the PTM interposes the load and checks if the data was modified. If so, it returns the new value from the log and not the value from memory. This approach can incur a high cost for large transactions because every load on the persistent memory region needs to first check for an updated value on the log. The longer the transaction, the bigger the log, and the longer it will take to search for the existence (and updated value) of a given variable. The authors of Mnemosyne mention that the choice of using a redo log was deliberately made, because it reduces the ordering constraints on writes to persistence, thus allowing this technique to perform only two persistence fences per transaction using a torn bit log. However, Mnemosyne's default implementation does not use the torn bit log technique, and therefore requires 4 persistence fences. Furthermore, in our benchmark experiments we noticed an increase in the number of persistence fences in concurrent settings; for example, with two concurrent threads, we identified transactions with more than 50 fences.

Table 1 summarizes the main features for failure-resilient transactional techniques, where N_{ranges} represents the number of contiguous ranges of data modified during a transaction and N_{stores} represents the number of word-sized stores to persistent memory done by the user code in the transaction. Some techniques are capable of logging a contiguous range of data instead of single word-sized stores, thus implying that $N_{ranges} \leq N_{stores}$.

The columns respectively indicate: which log type each of the techniques utilizes; the amount of persistent memory (in words) that may be used during a transaction associated with the persistence of the log; the number of persistence fences (`pfence` and `psync`) needed per transaction; how many stores are done to persistence, taking into account the stores done by the user code; whether the technique needs to interpose only stores, or both loads

	Log type	Persistent memory used per transaction	Nb. pfence+psync per transaction	Interposition type	Write amplification
Vista [20]	undo	$3 \times N_{ranges}$	n/a	stores	300%
Atlas [4]	undo	$4 \times N_{ranges}$	$2 + 3 \times N_{ranges}$	stores	400%
JustDo [16]	done-to-here	3 words	$2 + 3 \times N_{stores}$	stores	400%
Mnemosyne [31]	redo	$8 \times N_{stores}$	4 or more	loads + stores	300 − 600%
Romulus	volatile redo	n/a	4	stores	100%

Table 1: Comparison of the main transactional persistence techniques.

and stores to persistent memory; and the write amplification factor for each of the techniques (see §3). Notice that JustDo is the only algorithm in the table that requires specialized hardware.

3 BACKGROUND AND CONCEPTS

We briefly overview in this section the main concepts and features of persistent memory, as well as programming models and tools to exploit it in software.

3.1 Persistent memory

Persistent storage has been so far mainly based on disk persistence, with data being persisted through the whole I/O stack of the operating system, involving costly user/kernel mode switches from read/write system calls and data copying between kernel space and user space [6]. Recent studies propose to exploit high-performance next-generation NVM storage [6] for memory mapped file I/O. Mapping a file onto user memory space with an `mmap` system call enables users to access the file directly in the same way as data on memory. This approach minimizes overhead and simplifies application development, even though it suffers from some limitations, such as dynamically resizing the file and the pre-defined memory region to which it is mapped. Access to these new types of persistent memory using the direct access (DAX) feature available in both Linux and Windows allows for a much finer granularity of persistence at the cache line level.

For a correct understanding of the concepts used in the paper, we refer to the definitions proposed by Izraelitz et al. and extensively described in [16]. Failures are events that may corrupt application data, for example application process crash or an abrupt failure of the machine. 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. Data are persistent if after a non-corrupting failure are accessible by the recovery procedure. The same is true for persistent memory locations and persistent cache lines where data resides. A persistent memory region is a contiguous range of persistent memory locations, and it is not guaranteed to be in a consistent state after a failure. Data are in a consistent state if the relevant application-level correctness criteria hold. Hereafter, we use the term *failure* to simply refer to a non-corrupting failure.

Another concept, more performance related, is *write amplification* described in [23]. *Write amplification* is the number of additional bytes written to persistent memory for every byte of user data stored in persistent memory during a transaction. Additional bytes are incurred by the log, whether undo or redo, and the memory allocator's metadata.

3.2 Interposition of memory accesses

PTMs require at least the interposition of stores to persistent memory. This interposition is necessary to flush modified cache lines to persistence and, in the case of an undo log, to save the original data. Techniques with a redo log need the interposition of loads from persistent memory as well. This load interposition is required so as to have the latest modified data during a transaction. There are three different ways of achieving interposition: *manual* interposition by the user, *compiler* interposition, and *language* interposition.

With *manual* interposition, it is up to the user to explicitly call a method before a store is done to persistent memory. Techniques like Vista [20] and Atlas [4] use manual interposition of stores. As the user must identify where the stores to persistent memory are done in the code within the transaction, this approach is prone to errors and may yield non-recoverable transactions if the user misses some of the stores.

With *compiler* interposition, the compiler will insert a call to the PTM method responsible for handling stores and loads. This can be achieved by modifying the compiler or by extending an existing framework like `gnu-tm` in gcc. Mnemosyne [31] uses compiler interposition for stores and loads.

With *language* interposition, instrumentation of the stores or loads is done at the level of the programming language itself, typically through operator overloading. When using a language like C++ that supports operator overloading, one can trigger a method responsible for persistence management upon every access to a variable that has been annotated as persistent. This method can, for instance, save the original value in a log. We use this approach in our implementation of the algorithms shown in this paper. Except for rare cases, programs written in C can also be compiled with a C++ compiler and, therefore, provide the same functionality.

4 THE BASIC ALGORITHM

We first describe in this section the basic Romulus algorithm. We will then introduce additional variants that improve on the basic design and work around some of its performance limitations (§5). Proofs of correctness can be found in a companion technical report.

4.1 Model and assumptions

In the following description of Romulus, we rely on the C++ memory model [11]. As in [17], we assume that threads control the ordering and timing of persistence using three special instructions: *(i)* a persist write-back (`pwb`) initiates write-back of a specified location to persistent memory, but does not block; *(ii)* a subsequent persist fence (`pfence`) enforces an ordering between previous and subsequent writes-back in the current thread; finally, *(iii)* a persist sync (`psync`) blocks until all preceding `pfences` in the current

thread have become persistent. In the absence of fences, pwb instructions are allowed to reorder with respect to both ordinary and synchronization instructions. The implementation of each of these instructions on Intel and ARM processors is summarized in the following table:

	NVM on x86		NVM on ARM
pwb	CLFLUSH	CLWB or CLFLUSHOPT	DC CVAC
pfence	nop	SFENCE	DSB
psync	nop	SFENCE	DSB

We also use the persistent cache store order (*PCSO*) protocol [8] that guarantees that: *(i)* each pwb is ordered with respect to each preceding or subsequent pfence in its thread; and *(ii)* for any given thread and location, each pwb is ordered with respect to each preceding pwb of the same location in the same thread.

4.2 Principle and architecture

The design of Romulus has the objective of minimizing the number of persistent fences, knowing that this is a slow operation and represents a bottleneck of existing approaches. To achieve 4 persistent fences per transaction independent of the number of modified memory locations, Romulus uses twin copies of the data stored in persistent memory, so that at every point in time at least one of these copies is consistent. We name these two copies main and back (see Figure 1).

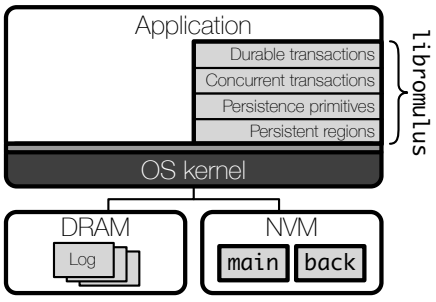

Figure 1: Architecture of Romulus.

The main region is where the user-code executes in-place modifications. At the end of a successful transaction, the modifications in main are copied over to back to bring it up-to-date. The back region is never accessed directly by the user-code and serves solely as a *backup* of the previous state of main. If the user code were to be allowed to access the back data, any pointer dereference would redirect to a memory location inside main.

In the event of a failure of the process that is mutating main, the recovery procedure will copy the contents of back over to main so as to revert any of the incomplete modifications done to main, thus restoring it to its previous consistent state. If a failure occurs during the copy from main to back, then the recovery procedure will copy the full contents of main to back.

Romulus follows a copy-on-write approach and will hence suffer from the associated performance issues when applied to large data sets. We will propose a variant of the algorithm to overcome these limitations in §5.

The architecture of Romulus is shown in Figure 1. The implementation of the algorithm and supporting libraries are entirely in user space, without any modification of the kernel. They are exposed to the application via a unified C/C++ library, libromulus, which provides support for durable and concurrent transactions, mediating all interactions with the NVM in a safe and consistent manner.

4.3 Data structures

The memory layout of Romulus is shown in Figure 2. The persistent memory area is separated into three regions: a persistent header (head) that is placed at the start of Romulus's persistent memory area, immediately followed by the main and back regions themselves.

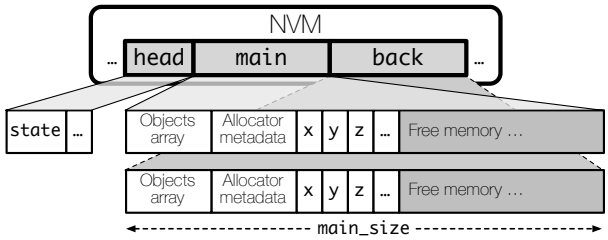

Figure 2: Memory layout of Romulus.

The persistent header holds medadata used to ensure consistency of persistent memory. It is initialized the first time a Romulus instance is created. The most important fields of the header are: *(i)* an atomic variable state that can take any of the three values indicating whether the current thread is executing outside a transaction (IDL), inside an active transaction (MUT), or immediately after the transaction has committed while modifications are being copied from main to back (CPY); *(ii)* a reference to the objects array—also designated as *root pointers*—maintained in the persistent memory region that allows user code to access persisted objects; *(iii)* a pointer to the metadata of the memory allocator, also maintained in the persistent memory region; *(iv)* the amount of bytes effectively used in the main region; *(v)* as well as internal variables to guarantee correct initialization of the header, maintain consistency of the data structure upon recovery, and perform bookkeeping operations.

As can be seen in Figure 2, the persistent header is not replicated to the back region, it is used to guarantee at initialization that the main region is in a valid consistent state before being accessed by user code. Furthermore, all the data present in the main region will be replicated at the end of the transaction, when calling end_transaction. The replication process is simply a raw copy of the bytes from the main to the back region. The back region is a snapshot of the previous consistent state of the main region and is never accessed directly by the user code. It does not contain a second copy of the objects present in the main region, but holds instead pointer values that refer to addresses in the main region.

4.4 Memory allocation and reclamation

One of the main challenges in designing and implementing transactional persistence is providing correct memory allocation and reclamation. Persistent transactions must recover from failures, even if those occur during memory management operations. For most PTMs described in the literature, a failure occurring while updating

the allocator's metadata can lead to inconsistencies. These inconsistencies are categorized as *internal* or *external* with respect to the allocator [2]. Internal inconsistencies can often have disastrous consequences, such as failure to restart properly upon recovery, erroneous re-allocation of memory objects currently in use, or leakage of large chunks of memory. External inconsistencies can occur when a failure happens after a call to the allocator has returned a pointer to a newly allocated block, but before this pointer has been made persistent, thus causing memory leaks.

Previous PTMs in the literature [4, 20, 31] tend to ignore these issues, and for correctness they rely on specialized garbage collector like Makalu [2]. Mnemosyne deliberately accepts the possibility of leaking in the event of a failure for performance reasons [31]. None of the Romulus algorithms suffer from any of these issues and, therefore, can be used with any memory allocator.

The memory allocator used in our implementation is based in the sequential allocator designed by Doug Lea [19]. Both the memory allocator and `objects` refer to data structures that are placed in the persistent region and have to follow the same procedure as any other user code, where every store to a persistent data requires a subsequent `pwb`. This is achieved by wrapping all attributes of every data structure used by the memory allocator with the `persist` class. The `persist<T>` class is a wrapper that allows to interpose every access to the underlying data of type `T` that is meant to be persistent. Every mutative access to the underlying data has its update operation redefined inside of `persist`, where a call to `pwb` is done after executing the store on the persistent data. This technique, which is also used by PMDK [27], significantly reduces the amount of modifications when porting user code written for volatile memory, while not requiring a special compiler. We only had to modify 61 lines in the memory allocator's original code, and we expect that other standard sequential memory allocators can be just as easily adapted.

In our approach, allocating and reclaiming memory is itself part of the transaction, meaning that, in the event of a failure during a transaction, any modifications on the allocator's metadata will be rollbacked along with all modifications of user variables pertaining to that transaction.

4.5 The Romulus algorithm

Romulus provides a persistent transactional memory interface, shown in Algorithm 1, with two main methods `begin_transaction` and `end_transaction` surrounding critical sections of user code accessing persistent data. The transaction provides consistency for the persisted data modified within the critical section. The consistency of the data relies on the `state` variable that indicates in what stage the transaction is. Upon start, a transaction changes the state from `IDL` to `MUT` indicating it is in progress. Thereafter, every access to persistent data from the transaction's user code is instrumented using language interposition, in particular every modification triggers a call to `pwb` for the modified cache line. Once the user code has completed its execution, `end_transaction` updates the state from `MUT` to `CPY` indicating the transaction has already committed to persistent memory, but modifications are still being propagated to the `back` region. The `back` region serves as a consistent state for the next transaction in case a failure occurs during execution of the user code. Finally, state reverts to `IDL` after the transaction has ended.

Algorithm 1 Basic Romulus algorithm.

```
1  void begin_transaction() {
2    state.store(MUT, memory_order_relaxed);
3    pwb(&state);
4    pfence();
5  }
7  void end_transaction() {
8    pfence();
9    state.store(CPY, memory_order_relaxed);
10   pwb(&state);
11   psync();                  // Provide ACID durable modifications on 'main'
12   copyMainToBack();                                  // pwb() for each cache line
13   pfence();                                  // No need for durability on 'back'
14   state.store(IDL, memory_order_relaxed);
15 }
17 void recover() {
18   int lstate = state.load(memory_order_relaxed);
19   if (lstate == IDL) return;
20   if (lstate == CPY) {
21     copyMainToBack();                                // pwb() for each cache line
22   } else if (lstate == MUT) {
23     copyBackToMain();                                // pwb() for each cache line
24   }
25   pfence();
26   state.store(IDL, memory_order_relaxed);
27 }
```

In the event of a failure, the `recover` method is called upon restart to restore the persistent memory to a consistent state. The `state` variable indicates which of the `main` or `back` region is in a consistent state: when `IDL`, both regions are consistent; when `MUT`, the `back` region is consistent; and finally when `CPY`, the `main` region is consistent. Once the restore method has completed, both the `main` and `back` regions are in a consistent state, with `back` containing an exact byte per byte copy of `main`.

4.6 Romulus API

To illustrate the programming interface of Romulus, we provide in Algorithm 2 sample code for a persistent set implemented as a sorted single linked list. Every data structure that must be persisted by Romulus has to be wrapped within the `persist<T>` class; this ensures transparent interposition of accesses to its attributes.

Algorithm 3 illustrates how one can instantiate and invoke persistent objects with Romulus. The linked list must first be created and stored in the persistent memory region (lines 2–8). Thereafter it can be accessed using regular method invocations, with transparent interposition ensuring data persistence and consistency (lines 10–13). Finally, the list is deallocated and removed from persistent memory (lines 15–21). All three steps are wrapped within independent Romulus transactions.

4.7 Log optimization

One of the main drawbacks of Romulus, which is shared with other copy-on-write designs, is that it must copy the whole data from `main` to `back` on each transaction. While simplifying the algorithm, this approach inevitably reduces throughput and wastes resources by repeatedly copying many unchanged bytes. Furthermore, the

Algorithm 2 Romulus-based linked list.

```
1  struct Node {
2    persist<K> key;                          // All node attributes are persisted
3    persist<Node *> next;
4    Node(const K& key) : key { key }, next{ nullptr } { }
5  };
6  persist<Node *> head { nullptr };          // Head and tail are also persisted
7  persist<Node *> tail { nullptr };

9  void find(const K& key, Node*& prev, Node*& node) {
10   for (prev = head; (node = prev->next) != tail; prev = node) {
11     if (node->key >= key) break;
12   }
13 }

15 bool contains(const K& key) {
16   bool found = false;
17   Romulus::read_transaction([&] () {              // Read-only transaction
18     Node *prev, *node;
19     find(key, prev, node);
20     found = (node != tail && node->key == key);
21   });
22   return found;
23 }

25 bool add(const K& key) {
26   bool added = false;
27   Romulus::update_transaction([&] () {            // Update transaction
28     Node *prev, *node;
29     find(key, prev, node);
30     added = !(node != tail && key == node->key);
31     if (!added) return;
32     Node *n = Romulus::alloc<Node>(key);
33     prev->next = n;
34     n->next = node;
35   });
36   return added;
37 }
```

Algorithm 3 Using Romulus-based linked list.

```
1  LinkedListSet<int> *set;
2  Romulus::begin_transaction();                   // Allocate list in NVM
3  set = Romulus::get_object<LinkedListSet<int>>(0);
4  if (set == nullptr) {
5    set = Romulus::alloc<LinkedListSet<int>>();
6    Romulus::put_object(0, set);
7  }
8  Romulus::end_transaction();

10 Romulus::begin_transaction();                   // Invoke operations on list
11 set->add(33);
12 assert(set->contains(33));
13 Romulus::end_transaction();

15 Romulus::begin_transaction();                   // Deallocate list from NVM
16 set = Romulus::get_object<LinkedListSet<int>>(0);
17 if (set != nullptr) {
18   Romulus::free(set);
19   Romulus::put_object(0, nullptr);
20 }
21 Romulus::end_transaction();
```

performance penalty increases as the size of `main` grows and the length of transactions decrease.

To address this issue, we incorporate a logging technique into Romulus, where all persistent stores inside a transaction are logged for later replication to back. The redo log keeps the memory locations that are being modified by the current transaction. At the end of the transaction, only these memory locations are copied from `main` to back rather than the whole persistent region. The log is not required for consistency since, in the event of a failure, the recovery procedure can simply use the same strategy as Romulus (see algorithm 1). Therefore, the log is not stored in persistent memory, residing in faster volatile memory for optimized performance.

With the log optimization, every store done by the user code is interposed and an entry is added to the log, followed by the in-place modification on `main` and the final `pwb` instruction. The order of these three steps is not important as long as the `pwb` is done *after* the in-place modification on `main`.

Unlike other log-based approaches in the literature, Romulus log stores only the addresses and ranges of modified data, but not the actual data. Furthermore, whereas other techniques that rely on persistent logs suffer from high *write amplification* because each byte written to persistent memory causes several additional bytes to be persisted to the log, Romulus's persistent writes are limited to the replication of the back region. Another advantage is that, since the log resides in volatile memory, its size does not need to be fixed beforehand. In our implementation, we simply use a dynamic array of address/range pairs (pseudo-code omitted for space considerations).

5 ALGORITHM EXTENSIONS

We describe in this section extensions of the basic Romulus algorithm, focusing in particular on the efficient support of concurrent transactions. These extensions leverage scalable synchronization mechanisms for achieving good performance even when many threads contend on shared persistent data structures.

5.1 Concurrent transactions

Similarly to most other persistent memory frameworks (e.g., Vista [20], Atlas [4], JustDo [16] and PMDK [27]), Romulus provides durable transactions for single-threaded applications. Support for concurrency in such settings can be as simple as using mutual exclusion locks—typically a single global lock—to embed transactions within critical sections and serialize them. Mnemosyne [31] allows concurrent transactions by leveraging a software transactional memory (STM) library based on TinySTM [25]. NV-Heaps [7] embeds its own lock-based STM.

The global lock approach has the advantage to be simple to implement and reason about, but it does not scale well. On the other hand, the STM approach has the drawback of being tightly coupled with the underlying transactional memory mechanisms and capabilities, and that the price of concurrent synchronization must be paid for every transaction even in single-threaded applications.

Our approach is to provide two different APIs, one for concurrent usage in multi-threaded applications, and another one for single-threaded applications. We have developed two different concurrency mechanisms for Romulus, as described next. Each of these mechanisms provides two types of concurrent transactions, *update* and *read-only* transactions, which are both "irrevocable", i.e., they never need to abort and restart. Read-only transactions are by their very nature *disjoint-access parallel*, which theoretically allows them to scale.

5.2 C-RW-WP synchronization

Our first concurrency approach uses flat combining [12, 14], essentially replacing the mutual exclusion lock with a C-RW-WP reader-writer lock [3]. With flat combining, multiple update transactions can be aggregated and processed with a single lock acquisition and release. An update transaction first announces itself in the flat combining array by registering a pointer to its actual code. Thereafter, the C-RW-WP algorithm acquires the lock, traverses the array, executes all registered transactions, and finally releases the lock.

In our C-RW-WP implementation we replace the cohort lock by a simpler spin-lock, but thanks to the flat combining technique starvation-free progress [15] is still guaranteed for update transactions. We implement the C-RW-WP lock's "read indicator" as an array where each entry is statically assigned to a thread and extends over two cache lines, so as to avoid false sharing; this ensures low contention, and consequently low overhead and high scalability for read-only transactions. The coupling of flat combining with C-RW-WP is a novel approach which takes the best of both techniques, providing scalable read-only transactions and starvation-free update transactions.

All the variables used in the implementation of the C-RW-WP lock are stored in volatile memory, even though the lock protects data that may be persistent. A reader-writer lock grants exclusive access to a single writer. To guarantee *durable linearizability* [18], all modifications to persistent memory must be completely persisted before the lock is released by the writer, with their effects visible to other threads. In addition, reader-writer locks provide linearizability, only allowing visibility of operation effects to other threads after the release of the exclusive lock. As for other writer threads that have published their operations on the flat combine array, visibility and durability is guaranteed as soon as an operation is marked as executed by flushing the associated cache line and resetting its entry in the array (refer to actual code in [10] for implementation details). This approach provides complete separation between durability and linearizability: it allows the actual synchronization to *not* be persisted and avoids extra persistence fences that could adversely impact the performance of readers. No persistence fence is required for shared-lock acquisition.

5.3 Left-right synchronization

Romulus requires two persistent copies of the data to guarantee consistent recovery. The existence of these two copies makes it a natural fit for integration with the left-right (LR) [24] synchronization primitive. LR is a universal construct that provides wait-free population-oblivious progress for read-only operations, and blocking starvation-free progress for update operations. The integration of LR with Romulus creates the first persistent universal construct with partial non-blocking progress. Any user application will not only have durable linearizable transactions, but read-only transactions will also automatically profit from the scalability and low latency provided by wait-free population-oblivious progress.

The LR concurrency control technique is based on two replicas of the same data and allows an unlimited number of readers to access one instance, while a single writer modifies the other instance. This synchronization mechanism can be summarily described as follows, where the two instances of the data are referred as *left* and *right*. The synchronization is achieved using a control variable indicating to readers which of the two copies to access. The writer starts by modifying the *right* instance, whereas the *left* instance is attributed to the readers. Once the writer completes the modification, the control variable is changed and new readers will read from the *right* instance. The writer then waits for the completion of all the readers still running on the *left* instance and finally repeats its modification on the *left* instance, bringing both instances up-to-date. It is hence up to the writer to ensure that readers are always running on the instance that is not being modified.

This approach has some similarities with the basic Romulus algorithm, yet with one significant difference: for LR to work correctly both regions must be traversable by the read-only operations, whereas in Romulus only the `main` region can be traversed. Any attempt to traverse data in `back` will re-direct to memory locations in `main`, given that all pointers refer to objects within the `main`'s memory range. In other words, the two instances are *shallow* copies in LR while they are *byte-per-byte* copies in Romulus. Fortunately, we can overcome this limitation through the use of *synthetic pointers*.

Read-only traversals of the `back` instance are done with interposition of loads, such that an offset is added on every variable access. This offset is equal to the size of the `main` memory region, so as to create a synthetic pointer that refers to the equivalent memory location within `back` (see Figure 3). During a transaction, we use a thread-local variable to indicate to the load-interposing method which of the instances to use and adjust the offset appropriately. This synthetic pointer approach allows us to use an efficient raw memory copy operation at the end of transactions or during recovery, without the need to perform shallow copying.

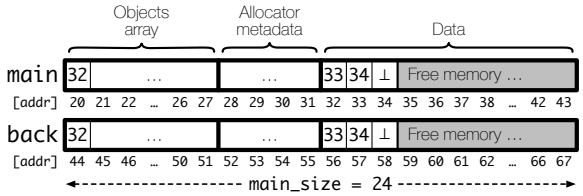

Figure 3: Memory layout of RomulusLR for a minimal example of linked list with three nodes and values 32, 33, 34.

Figure 3 shows an example of the memory layout with RomulusLR. In this specific example, each load instruction executed in the `back` region will be adjusted with an offset of 24 bytes added to references from the main region; for instance, address 32 + 24 = 56 refers to the head of the linked list. This transformation allows readers to always access data on the `back` region when they operate on that region.

Some adaptations to LR were also necessary for update transactions. While in the original LR algorithm writers can start execution on any of the two instances, in RomulusLR the control variable is toggled twice during an update transaction. This guarantees that update transactions always start execution on the `main` region and greatly simplifies the implementation because the memory allocator can operate on just the `main` region.

The integration of LR with Romulus requires that readers can only access the `main` region after it is guaranteed that all updates to that region are persisted by calling `psync` (see algorithm 1).

Thereafter, accesses to the main region are durable linearizable and, in the event of a failure, the recovery method will complete the transaction by copying the entire main region to the back region. From this moment on, the effects of the current update transaction can be made visible and the control variable is changed so that new readers can start executing on the main region.

For both concurrent solutions proposed, there is always a single writer executing an update transaction that aggregates all mutations previously published by other writer threads in the flat combining array. All mutations are hence serialized and performed by a single writer. This approach minimizes lock acquisition and conflicts while providing starvation freedom for update operations. It is also less prone to cache line bouncing and generates less cache coherence traffic while maintaining cache locality for larger update transactions. Furthermore, the average number of persistent fences per mutation can be smaller than 4 because several updates are aggregated within a single update transaction. Regarding memory allocation, providing exclusive access to a single writer thread allows Romulus implementations to directly leverage state-of-the-art sequential memory allocators.

In the rest of the paper, Romulus refers to the basic algorithm with concurrent accesses synchronized using C-RW-WP; RomulusLog denotes the basic algorithm with volatile log optimization and C-RW-WP; and finally RomulusLR corresponds to the basic algorithm with volatile log optimization and LR synchronization.

6 EVALUATION

We present in this section a detailed evaluation of Romulus and compare it against other state-of-the-art techniques using both synthetic benchmarks and real-world applications.

6.1 Experimental setup

Our microbenchmarks were executed on a dual-socket 2.10 GHz Intel Xeon E5-2683 ("Broadwell") with a total of 32 hyper-threaded cores (64 HW threads), running Ubuntu LTS and using gcc 7.2 with the -O3 optimization flag. The CPU does not support the CLFLUSHOPT and CLWB instructions and, therefore, we implement pwb with the CLFLUSH instruction and map the pfence and psync operations to NOP.

We compare Romulus against Mnemosyne [31] and PMDK [27] as they are considered to be the most generic and efficient approaches, they are representative of different designs (respectively redo and undo log), and their code is freely available. Mnemosyne was compiled using the latest available version on github (commit 855f452), which requires compilation with the -O0 flag. As for PMDK, we use the latest libpmemobj from github (commit f697c52) executed with the environment variable PMEM_IS_PMEM_FORCE=1. Since PMDK does not provide built-in support for concurrent transactions, in our evaluation we protect concurrent accesses with a standard reader-writer lock (std::shared_timed_mutex) with reader preference. We rely on a memory-mapped file located in the /dev/shm/ folder for all implementations.

Unless otherwise noted, our evaluations use DRAM to mimic systems that implement persistent memory using supercapacitor-backed DRAM (e.g., Viking NVDIMMs [28] or HPE NVDIMMs [29]), and no artificial delays are added. The data points labeled STT emulate STT-RAM persistent memory, with delays of 140, 200 and 200 ns respectively injected for each pwb, pfence and psync [5]. Data points labeled PCM emulate PCM-RAM, with delays of 340, 500 and 500 ns respectively injected for each pwb, pfence and psync. Similarly to Mnemosyne, delays are measured using rdtsc and no delays are added for loads from persistent memory. Each data point is the median of 5 runs with each run executing for 20 seconds, unless otherwise noted in the figure.

6.2 Data structures

Application data is typically stored in complex structures, such as a hash map or a tree, and therefore our benchmarks cover multiple data structures. We conduct evaluations on a linked list, a resizable hash map and a red-black tree, as representative examples of structures used by real-world applications to store data. Figure 4 shows throughput for update-only and read-only workloads (note the logarithmic scales) with data structures holding 1,000 entries.[1] 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.

Looking at the performance of these data structures, we observe that the linked list has higher throughput than the red-back tree. This can be explained by the small size of the data structures and the fact that the linked list performs on average fewer stores, and hence fewer costly accesses to persistent memory. Further investigation shows that the linked list executes an average of 10 pwb instructions, while the red-black tree has a more disperse histogram, with two peaks at 50 and 130 pwbs per transaction. Another interesting finding is that most of the stores inside transactions are triggered by the memory allocator. RomulusLog performs generally at least 4× better than Mnemosyne thanks to its lower write amplification and lighter synchronization mechanism.

PMDK performs better than would be expected from an undo-log based technique, but one should keep in mind that the tests were conducted on a machine only supporting the CLFLUSH instruction. In this case, all the extra persistence fences that would be necessary to implement an undo log are omitted, because the CLFLUSH instructions already guarantee strict ordering between themselves. On a machine that provides solely CLFLUSH, performance is therefore mainly dominated by the number of pwb instructions per transaction. After investigation, it turns out that Romulus and PMDK execute a similar number of pwb instructions. The memory allocator of PMDK for the hash map issues only one CLFLUSH to allocate memory and one when memory is freed, for a total of 11 CLFLUSH instructions per transaction. This demonstrates that PMDK's memory allocator is highly optimized for small allocations and offers room for improvement for Romulus, which uses a much less efficient allocator. Even so, in update scenarios with small sized transactions, RomulusLog still provides the best results with at least twice the performance of PMDK.

[1] This is the limit beyond which Mnemosyne starts becoming unstable and exhibits random crashes. In addition, Mnemosyne only support up to 31 threads and hence samples are missing beyond this point in scalability tests. We have been in touch with the authors to work around these limitations but did not manage to completely solve them for our experiments.

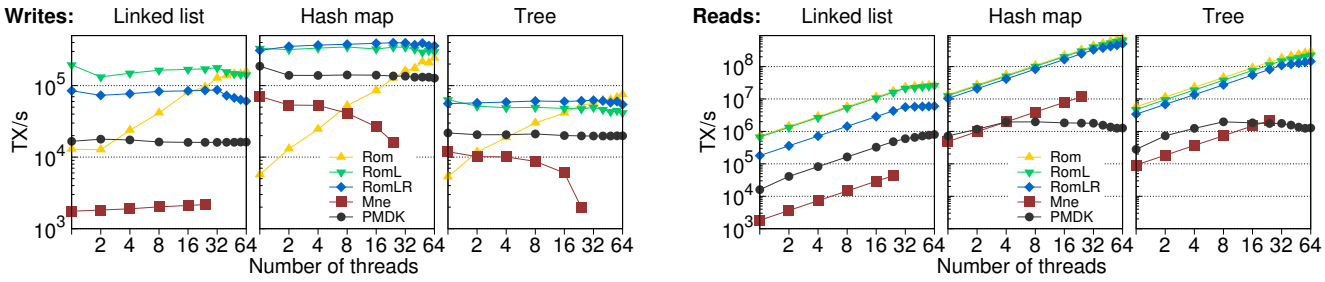

Figure 4: Update (left) and read (right) operations on various data structures (Mnemosyne data points are one run of 20 s).

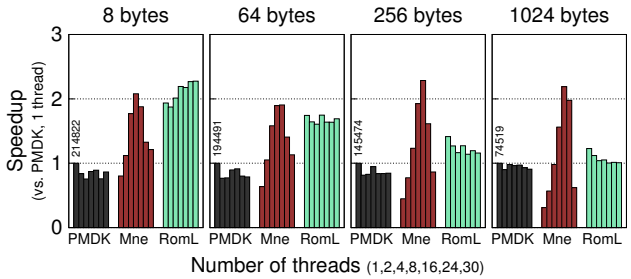

Figure 5: Speedup of a hash map with 100 entries relative to single-treaded throughput of PMDK. Mnemosyne data points correspond to 1 run of 20s.

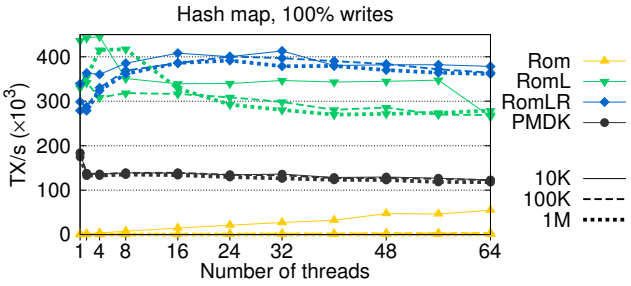

Figure 6: Hash map with varying numbers of keys and threads executing update-only operations.

As expected, read-only transactions represent a good example of workload for which Romulus excels. The throughput of Romulus is 10× to 400× higher than PMDK, while compared with Mnemosyne it is 25× to 50× higher. Indeed, read-only operations can be done in-place, either directly in main memory or more likely within the cache. Furthermore, the reader-writer lock and LR concurrency mechanism provide excellent performance and scalability with read-mostly workloads.

We were intrigued to find out that Mnemosyne's performance degraded on the hash map when increasing the number of threads, since the performance evaluation of the original paper exhibits good scalability. Therefore, in order to better reproduce these results, we have developed a second hash map implementation with a fixed number of 2,048 buckets. As one can observe in Figure 5, the throughput becomes much better for such a statically-dimensioned hash map. This difference can be explained as follows. On the one hand, Mnemosyne supports concurrent access using TinySTM, a software transactional memory that relies on fine grain locking to allow multiple threads executing disjoint transactions to progress. On the other hand, the implementation of our resizable hash map relies on a shared counter to keep track of the number of elements and determine when to resize. Modifications to the counter upon every update operation (insertion or removal) will generate contention and trigger conflicts, which in turn will cause transactions to abort and eventually hamper scalability. The Romulus implementations do not suffer from this issue as they rely on simpler synchronization primitives and transactions never abort.

We also evaluated scalability to larger data structures up to one million entries with update-only workloads.[2] The results, shown in Figure 6, do not differ much from those presented earlier for 1,000 entries and highlight that almost all implementations can support large data structures without performance penalty. As expected, the only exception is the basic Romulus algorithm, which suffers from the data size due to the longer copy procedure necessary to keep the back region up to date.

As previously mentioned, we also emulated STT-RAM persistent memory by injecting a delay for every instruction to persistent memory. Performance results, which are omitted because of space constraints, are highly similar to those shown in Figure 4. We believe this is due to the limited number of persistent stores per transaction and the fact that `CLFLUSH` is a costly instruction as compared to `CLFLUSHOPT` and `CLWB`, which are available on more recent processors. Note that our emulation only affects the throughput of update operations because load instructions have no added delay.

6.3 Read-dominated workloads

Figure 7 shows the number of transactions per second for the hash map data structure as the number of concurrent reader threads increases. A logarithmic scale is used for both axes. The left graph shows the number of read- and update-only transactions with 2 concurrent writer threads and an increasing number of reader threads, while the right graph has no writer thread and only runs read-only transactions. Note that PMDK uses an unfair reader-writer lock that gives preference to readers and prevents writers from running with 16 concurrent reader threads or more.

RomulusLR is the only one of the tested techniques that provides scalability for read operations, even with concurrent update operations. This is possible thanks to the use of the left-right concurrency mechanism that provides wait-free progress for read-only transactions.

[2] Mnemosyne is omitted as the publicly available implementation does not support allocation of sufficiently large amounts of data.

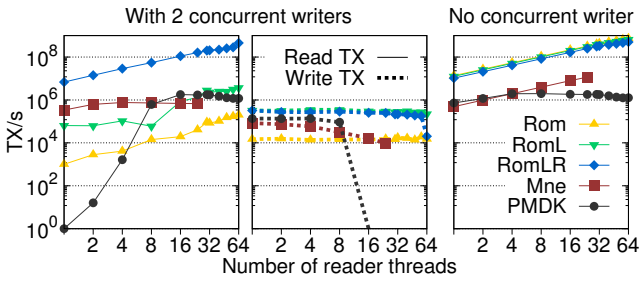

Figure 7: Read operations with 2 concurrent writers (left) and no writer (right) on a hash map with 1,000 entries. Mnemosyne data points correspond to 1 run of 20s.

6.4 RomulusDB

Our three PTMs provide durable concurrent transactions with ACID properties (atomicity, consistency, isolation, durability) and durable linearizability when executed in concurrent settings. These PTMs can be straightforwardly applied to any sequential implementation of a *map* data structure and use it to construct a key-value store with persistence. We used RomulusLog to wrap a hash map and implement the same interface as the popular LevelDB [13] database. We named our persistent key-value store RomulusDB.

Unlike LevelDB, RomulusDB is capable of executing real transactions, with durability ensured for every transaction. By default, update operations in LevelDB are not immediately durable—unless the option `WriteOptions.sync` is enabled—which implies that in the event of a non-corrupting failure, a possibly large amount of recently completed operations may be lost. Furthermore, LevelDB does not provide transactional semantics, but provides instead the simpler model of *write batches*.

LevelDB benchmarks utilize 16-byte keys and 100-byte values. The `fillseq` benchmark measures the time for a thread to insert one million distinct key-value pairs in the database, using sequential keys. The `fillrandom` benchmark performs one million insertions of random keys per thread (note that existing key-value pairs may be overwritten because of the randomness). The `overwrite` benchmark is similar to `fillrandom` but starts from a pre-populated database. The `fillsync` benchmark measures the time to insert 1,000 key-value pairs in a database, with durability enforced by enabling the `WriteOptions.sync` option. `readseq` and `readreverse` do a single read-only iteration over the entire database.

Results are shown in Figure 8. For read-only operations RomulusDB performs extremely well independently of the read pattern (sequential or reverse). Indeed, the traversal order has no effect because the random accesses to a hash map have no particular extra cost. RomulusDB scales linearly with the number of threads for read operations, with an operation execution time remaining constant when increasing the number of threads. It outperforms LevelDB in all of the read benchmarks.

As for update operations, the only truly comparable benchmark is `fillsync` as it provides durable transactions on both systems. With LevelDB, enforcing durable transactions triggers execution of `fdatasync` on every transaction, which brings a huge penalty to key-value databases that rely on disk persistence. For the other update benchmarks, RomulusDB can be up to 50% slower than LevelDB, but it provides durable transactions whereas LevelDB flushes updates to disk approximately every 1,000 kB, hence providing a weaker consistency guarantee referred to in the literature as *buffered durability*. The measured number of `fdatasync` system calls per thread execution was less than 100 for one million insertions of 116-byte objects.

Finally, the `fill-100k` benchmark measures the time it takes to write 1,000 key-value pairs of 100 kB. LevelDB executes around 100 `fdatasync` system calls per thread execution for 1,000 transactions. In contrast, Romulus already persists on every transaction but executes less costly persistence fences instructions and can take advantage of large transaction sizes to aggregate writes and flush full cache lines, hence demonstrating impressive performance. LevelDB transaction throughput is 3× slower than RomulusDB in single-thread execution, as the number of threads increases, performance degrades even more to reach a factor of 25× for 64 threads.

6.5 Recovery cost

Recovery is a lengthy procedure that requires copying data from `main` to `back` or vice versa. Yet, because Romulus can copy only the region up to the last allocated object, the costs actually depend on the size of the data structures stored in persistent memory. Measurements on our machine for a hash map show that the recovery method takes about 114 μs for 1,000 key-values and 127 ms for 1,000,000 key-value pairs. Recovering an entire 1 GB region takes around 1 second. This value grows linearly with the size of the persistent memory region, with the largest source of overhead being the `pwb` calls (measurements were done using `CLFLUSH`).

6.6 Overhead of different fences

To better study the cost of fences, we execute the simple SPS micro-benchmark regularly used in the literature [7, 21, 22] that stores a large array of integers in PM and randomly swap some of its values. This benchmark allows us to understand the performance profile of each PTM under different transaction sizes. Each swap exchanges the values of two randomly selected entries of an array of 10,000 64-bit integers, hence modifying two memory words in PM.

Figure 9 shows the number of *swaps* executed per second as a function of the transaction size (logarithmic scale), with a single-threaded application running on a C5 compute-intensive instance of AWS supporting the `CLWB` instruction. The five plots represent different choices of fence implementations and/or delay injection: *(i)* `pwb` is mapped to a `CLWB` and `pfence/psync` to `SFENCE`; *(ii)* `pwb` is mapped to a `CLFLUSHOPT` and `pfence/psync` to a `SFENCE`; *(iii)* `pwb` is mapped to a `CLFLUSH` and `pfence/psync` to a `nop`; *(iv)* `pwb` is mapped to a delay of 140 ns and `pfence/psync` to a delay of 200 ns, so as to simulate STT-RAM; and *(v)* `pwb` is mapped to a delay of 340 ns and `pfence/psync` to a delay of 500 ns, so as to simulate PCM-RAM. The timings for the injected delays were taken from previous work [5].

There is no memory allocation or de-allocation during execution of the SPS benchmark because it consists of swapping integers on an array. This allows us to examine the throughput of each algorithm, without being affected by the implementation of the NVM memory allocator specific to each PTM.

As observed previously, thanks to their small number of persistence fences and low synchronization overhead, the RomulusLog

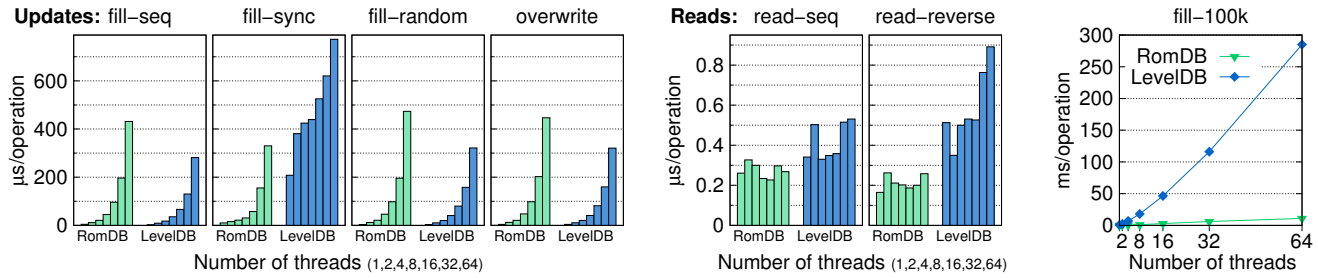

Figure 8: One run of LevelDB benchmark with update (left) and read (center) operations, as well as with sequences of update operations (right).

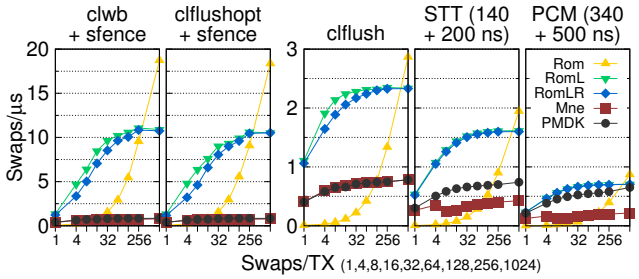

Figure 9: SPS benchmark with varying transaction size for different types of fences. All data points correspond to 1 run of 20s.

and RomulusLR perform best on all workloads except for high transaction sizes (1,024 swaps per transaction), where making a copy of the entire array becomes advantageous enough for Romulus to overtake them. When running short transactions, the pwbs constitute the main bottleneck of RomulusLog and RomulusLR and, as their cost increases, the difference between these two approaches and Mnemosyne or PMDK diminishes. This benchmark highlights that, when using the `CLWB` instruction instead of a more expensive `CLFLUSH`, the gains of Romulus over other approaches becomes even more important.

7 CONCLUSION

Romulus provides consistent recovery of persistent application data from non-corrupting failures. Persistent transactions in Romulus do not distinguish memory allocator metadata from user-defined application data. Any modification during a transaction will only be durable if the `main` region is left in a consistent state, otherwise the recovery procedure will revert to the consistent state at the start of the transaction. This approach, eliminates all issues inherent to NVM memory allocators, such as memory leaks and allocator metadata corruption.

Concurrent transactions in Romulus are durable linearizable. The two synchronization mechanisms integrated into Romulus, C-RW-WP and left-right, have low overhead. Read transactions require a single store-load fence and their announcement mechanism allows uncontended execution. Consequently, reader throughput scales linearly with the number of threads. Moreover, RomulusLR is the first PTM to provide wait-free progress for read-only transactions.

The low synchronization overhead, combined with a reduced number of persistence fences and low write amplification, contributes to the overall performance gains when compared with other solutions. Ultimately, our evaluation reveals that Romulus achieves twice the throughput of current state of the art PTMs in update-only workloads, and more than one order of magnitude in read-mostly scenarios.

ACKNOWLEDGMENTS

This work is supported in part by the Swiss National Science Foundation (SNSF) under project number 200021-178822/1.

REFERENCES

[1] Joy Arulraj, Andrew Pavlo, and Subramanya R Dulloor. 2015. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 707–722.

[2] Kumud Bhandari, Dhruva R Chakrabarti, and Hans-J Boehm. 2016. 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, 677–694.

[3] Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J Marathe, and Nir Shavit. 2013. NUMA-aware reader-writer locks. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 157–166.

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

[5] Himanshu Chauhan, Irina Calciu, Vijay Chidambaram, Eric Schkufza, Onur Mutlu, and Pratap Subrahmanyam. 2016. NVMOVE: Helping Programmers Move to Byte-Based Persistence.. In *INFLOW@ OSDI*.

[6] Jungsik Choi, Jiwon Kim, and Hwansoo Han. 2017. Efficient Memory Mapped File I/O for In-Memory File Systems. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*. USENIX Association.

[7] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *ACM Sigplan Notices* 46, 3 (2011), 105–118.

[8] Nachshon Cohen, Michal Friedman, and James R Larus. 2017. Efficient logging in non-volatile memory by exploiting coherency protocols. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 67.

[9] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 133–146.

[10] Andreia Correia, Pascal Felber, and Pedro Ramalhete. 2018. Romulus github. https://github.com/pramalhe/Romulus. (2018).

[11] CPP-ISO-committee. 2013. C++ Memory Order. http://en.cppreference.com/w/c/atomic/memory_order. (2013).

[12] Dave Dice, Virendra J Marathe, and Nir Shavit. 2011. Flat-combining NUMA locks. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 65–74.

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

[14] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 355–364.

[15] Maurice Herlihy and Nir Shavit. 2011. *The art of multiprocessor programming*. Morgan Kaufmann.

[16] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016. Failure-atomic persistent memory updates via JUSTDO logging. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 427–442.

[17] Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. 2016. Brief Announcement: Preserving Happens-before in Persistent Memory. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 157–159.

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

[19] Doug Lea and Wolfram Gloger. 1996. A memory allocator. (1996).

[20] David E Lowell and Peter M Chen. 1997. Free transactions with rio vista. In *ACM SIGOPS Operating Systems Review*, Vol. 31. ACM, 92–101.

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

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

[23] Sanketh Nalli, Swapnil Haria, Mark D Hill, Michael M Swift, Haris Volos, and Kimberly Keeton. 2017. An Analysis of Persistent Memory Use with WHISPER. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 135–148.

[24] Pedro Ramalhete and Andreia Correia. 2015. Brief Announcement: Left-Right-A Concurrency Control Technique with Wait-Free Population Oblivious Reads. *Distributed* (2015), 663.

[25] T Riegel, P Felber, and C Fetzer. 2010. TinySTM. (2010).

[26] David Schwalb, Markus Dreseler, Matthias Uflacker, and Hasso Plattner. 2015. NVC-hashmap: A persistent and concurrent hashmap for non-volatile memories. In *Proceedings of the 3rd VLDB Workshop on In-Memory Data Mangement and Analytics*. ACM, 4.

[27] Pmem team. 2017. Persistent Memory Programming. http://pmem.io. (2017).

[28] Viking Technology. 2017. Persistent Memory Technologies. http://www.vikingtechnology.com/products/nvdimm/. (2017).

[29] Viking Technology. 2017. Persistent Memory Technologies. http://www.vikingtechnology.com/products/nvdimm/. (2017).

[30] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, Roy H Campbell, et al. 2011. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory.. In *FAST*, Vol. 11. 61–75.

[31] Haris Volos, Andres Jaan Tack, and Michael M Swift. 2011. Mnemosyne: Lightweight persistent memory. In *ACM SIGARCH Computer Architecture News*, Vol. 39. ACM, 91–104.