RESEARCH-ARTICLE

Persistent HyTM via Fast Path Fine-Grained Locking

GAETANO COCCIMIGLIO, University of Waterloo, Waterloo, ON, Canada

TREVOR ALEXANDER BROWN, University of Waterloo, Waterloo, ON, Canada

SRIVATSAN S RAVI, University of Southern California, Los Angeles, CA, United States

Open Access Support provided by:

University of Waterloo

University of Southern California

Published: 16 July 2025

Citation in BibTeX format

SPAA '25: 37th ACM Symposium on Parallelism in Algorithms and Architectures
July 28 - August 1, 2025
OR, Portland, USA

Conference Sponsors:
SIGARCH
SIGACT

https://doi.org/10.1145/3694906.3743314
ISBN: 9798400712586

Persistent HyTM via Fast Path Fine-Grained Locking

Gaetano Coccimiglio
gccoccim@uwaterloo.ca
University of Waterloo
Waterloo, Ontario, Canada

Trevor Brown
trevor.brown@uwaterloo.ca
University of Waterloo
Waterloo, Ontario, Canada

Srivatsan Ravi
srivatsr@usc.edu
University of Southern California
Los Angeles, California, USA

Abstract

Utilizing hardware transactional memory (HTM) in conjunction with non-volatile memory (NVM) to achieve persistence is quite difficult and somewhat awkward due to the fact that the primitives utilized to write data to NVM will abort HTM transactions. We present several persistent hybrid transactional memory (HyTM) that, perhaps counterintuitively, utilize an HTM fast path primarily to read or acquire fine-grained locks which protect data items. Our implementations guarantee durable linearizable transactions and the STM path satisfies either weak progressiveness or strong progressiveness. We discuss the design choices related to the differing progress guarantees and we examine how these design choices impact performance. We evaluate our persistent HyTM implementations using various microbenchmarks. Despite the challenges and apparent awkwardness of using current implementations of HTM to achieve persistence, our implementations achieve up to 10x improved performance compared to the existing state of the art persistent STMs and up to 2.6x improved performance compared to the existing state of the art persistent HyTMs.

CCS Concepts

• **Computing methodologies** → **Concurrent algorithms**.

Keywords

Non-volatile memory, Transactional Memory, Hybrid Transactional Memory, Durable linearizability

ACM Reference Format:
Gaetano Coccimiglio, Trevor Brown, and Srivatsan Ravi. 2025. Persistent HyTM via Fast Path Fine-Grained Locking. In *37th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '25), July 28-August 1, 2025, Portland, OR, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3694906.3743314

1 Introduction

The recent commercial availability of byte-addressable Non-Volatile Memory (NVM) has sparked increased efforts towards designing persistent concurrent data structures capable of recovering from system crashes. These efforts have proven to be challenging since effectively utilizing NVM introduces new issues on-top of the already difficult task of ensuring correctness and efficiency in a completely crash free setting. We consider system crashes caused by power failures. In the first few generations of processors with support for NVRAM, the NVRAM itself is non-volatile and retains its contents in the event of a power failure, but the caches and registers (and, of course, DRAM) remain volatile. Data in the caches must be explicitly *flushed* to NVM in order to be persisted. In doing so, one must consider issues such as when should data be written back to NVM, write back ordering and at what frequency to issue write backs. Data in the cache can also be automatically flushed in the background at any time by the system without the programmer's knowledge. To guarantee that the data is persisted in a timely manner, one nominally adds flushes at key places in the program. The most recent Intel processors are equipped with *eADR*. This makes the processor cache effectively non-volatile as it is automatically flushed to NVM in the event of a power failure [19]. eADR removes the need for explicit flushes. However, eADR does not remove the complexity of designing correct persistent data structures. The programmer still needs to carefully order writes to any persistent data or risk corrupting the state of the data structure in NVM (we discuss this further in § 2.1). In this work, we simultaneously solve two problems: correctly persisting a program's data and synchronizing its threads, relieving the programmer from the burden of having to meticulously design synchronization like fine-grained locking protocols.

Much of the existing research regarding persistent concurrent data structures has focused on hand-crafted approaches often involving persisting an existing volatile data structure [26, 31–33, 59, 61]. Unfortunately, designing efficient and correct hand-crafted persistent data structures is often difficult, time consuming and error prone. Much of the challenge stems from the fact that it is already difficult to achieve correct volatile concurrent data structures.

Synchronization between threads can be achieved through the use of *transactional memory, (TM)*. TMs are synchronization mechanisms that allows users to execute sequences of memory accesses as atomic *transactions*. A transaction either *commits* and appears as a single indivisible step, or *aborts* and has no visible effect. Similarly, a *persistent TM (PTM)* provides the same functionality but also ensures that the effects of transactions are written back to NVM. We say that a TM *instruments* memory accesses if it performs additional accesses to metadata for each memory access.

Originally *software TMs (STMs)* were the only available option for TM. STMs typically require read/write instrumentation. Some processors now have *hardware TM (HTM)*, offering instructions that allow users to mark blocks of code as transactional where the processor is responsible for ensuring the atomicity of these *hardware transactions* [20, 46]. Current implementations of HTM can abort unconditionally, requiring a software based fallback code path. TMs that combine a *hardware path* executed on HTM and

SPAA '25, Portland, OR, USA

ACM ISBN 979-8-4007-1258-6/2025/07
https://doi.org/10.1145/3694906.3743314

a *software path* executed in software are known as *hybrid TMs (HyTMs)*.

In the volatile setting, HyTMs can typically avoid instrumenting memory accesses on the hardware path which generally leads to better performance compared to STMs. By extension, one might think that for a *persistent HyTM*, it is obvious that the use of HTM would similarly lead to better performance. However, *flush* instructions, which are required for persistence, force hardware transactions to abort! This means that if we want to persist the effects of a hardware transaction, we must flush *after* the hardware transaction has completed. Moreover, we need additional synchronization if we want to ensure that the transaction appears to be persisted at the same point that it commits, As a result, extra care is needed to ensure that the flush instructions and synchronization required to persist transactions do not cause poor performance.

In this work, we present a family of persistent HyTMs that we call NV-HALT (Non-Volatile Hardware Assisted Locking Transactions). As the name suggests, hardware transactions in our persistent HyTM are mainly used to read and acquire fine-grained locks. These locks are used to prevent threads from observing transactional stores that have not yet been written back to NVM. Locking in this manner allows us to ensure a transaction appears to be persistent atomically at the same time as it commits. While one might hope to leave synchronization entirely to hardware, we show that this type of instrumentation performs well on real hardware. Unlike the existing state of the art, NV-HALT utilizes a non-trivial fallback path allowing for concurrency between hardware and software path transactions. We design and implement several versions of NV-HALT offering different progress guarantees and levels of performance in practice. We evaluate and compare our persistent HyTM against a recent state of the art persistent STM as well as a state of the art persistent HyTM. Our evaluation shows that NV-HALT achieves up to 10x improved performance compared to the state of the art persistent STM and up to 2.6x improved performance compared to the state of the art persistent HyTM.

Contributions. We provide the following specific contributions:

- We present several versions of NV-HALT, a novel persistent HyTM that features a distinct usage of fine-grained locks to ensure that transaction commits and persistence appear (together) to be atomic. NV-HALT allows for greater concurrency and offers improved progressiveness compared to the prior state of the art persistent HyTM, SPHT [7]. Unlike prior work, NV-HALT has a non-trivial fallback path that allows for concurrency between hardware and software transactions. NV-HALT does not require additional background threads. In NV-HALT only transactions with intersecting accesses can abort one another which is an improvement compared to SPHT where transactions that access unrelated data can still abort each other § 3.
- We implement NV-HALT on a real system that supports hardware transactions and is equipped with NVM § 4.
- As part of this implementation we provide a custom allocator and memory reclamation mechanism. Our approach to memory management avoids memory leaks unlike SPHT. Moreover, our allocator avoids introducing additional aborts unlike the existing state of the art PSTM § 4.1.
- We experimentally evaluate NV-HALT against the state of the art PSTM and state of the art persistent HyTM. Our evaluation demonstrates a clear improvement compared to the existing state of the art. In many of our benchmarks, NV-HALT achieves more than 2x more throughput compared to existing PTMs and up to 3.5x better energy efficiency § 5.
- We introduce the notion of O(1)-abortable transactions to address the fact that there does not exist any formalism to describe how many aborts can be induced by a hardware fast path in a HyTM § 2.2.1. Our definition is a more realistic model of the behavior of real hybrid TM algorithms.

2 Background and Related Work

We consider an asynchronous shared memory system in which N processes communicate via shared *objects* which offer *operations*. A thread can *invoke* an operation on an object, which can (optionally) change the state of the object and return a *response*. The system can crash at any time in which case all N processes crash simultaneously. After a crash, a recovery procedure is invoked. New operations cannot be invoked until the recovery procedure returns.

Current implementations of NVM and HTM exist only on Intel systems which use a total store ordering (TSO) memory model. Thus, we assume a TSO memory model. This default assumption of sequential consistency aligns with the default behavior for atomics in C++. Implementers can rely on this default behavior, or fence as appropriate for their architecture.

2.1 Persistent Memory

Persistent memory is a part of main memory that retains its contents after a system crash. The DRAM along with processor registers and caches are not persistent (unless you have eADR in which case the cache is effectively non-volatile). In order to guarantee that the effects of a write are persisted, the programmer must explicitly *flush* data from the cache to NVM. On Intel platforms the prescribed way of flushing is via the `clflushopt` instruction. The `clflushopt` instruction initiates a flush of an entire cache line (nominally 64 bytes) to NVM and invalidates the cache line in all caches. It returns before the flush has been completed, allowing many flushes to be initiated in parallel and pipelined. The `clwb` instruction is similar, but it does not necessarily invalidate the cache line, which may make it more efficient [10]. However, many processors with NVM support (including the ones in our experimental system) implement `clwb` identically to `clflushopt`. In systems with such processors, `clwb` is no more or less efficient than `clflushopt`. The programmer can utilize fence instructions (`sfence` on Intel) to block until all previously initiated flushes have completed.

The processor may also arbitrarily flush data to NVM at any time, even if no explicit flush instructors have been issued. We refer to this as *background flush*. Background flushes typically occur as a result of the cache coherence protocol. These background flushes can cause data to be written back to NVM earlier than the programmer expects. Depending on the order of writes, this can leave the contents of NVM in an inconsistent state. To understand how such problems might arise consider a simple example of updating a singly linked list. One might add a key to a list by allocating a new node, locking the node containing the predecessor key and then performing the following writes: 1) update the new node's

next pointer to point to the old successor then 2) update the predecessor's next pointer to point the new node. Suppose that we do not perform any explicit flushes but a background flush causes the effects of the second update to be written back to NVM. If we crash after this background flush completes the first update will not be reflected in NVM and as a result of this, we will not be able to recover the latter part of the list.

eADR the Silver Bullet. The need to perform explicit flushes complicates the design of persistent algorithms. eADR removes the need for flush instructions by making the cache effectively non-volatile. Processors equipped with eADR will ensure that if a write has occurred in the cache, then it will eventually be written back to NVM. With eADR, the issues caused by background flushes as in our linked list example will not be a problem. However, a volatile cache is not the only concern when designing persistent algorithms.

eADR is not a Silver Bullet. eADR does not solve all of our NVM problems, it only removes the need for explicit flushes and it does not remove the need for careful write ordering which is the main challenge of utilizing NVM. Moreover, fences are still required to maintain write ordering correctness [17].

Recall the linked list example. Suppose that we swap the order of the updates to be the following: 1) update the predecessor's next pointer to point to the new node then 2) update the new node's next pointer to point to the old successor. In a volatile only setting, the order of the writes does not matter since we have already locked the predecessor and thus there is no problem with this ordering. However, in the persistent setting, if a crash occurs before the last update, we will not be able to recover part of the list. This is true even if we have we have eADR. In fact, with this new order of writes, eADR has the same effect as the problematic background flush that we discussed before. This example demonstrates how easy it is to corrupt the state of a data structure after a crash if one does not carefully order writes to persistent data, even if we have eADR. Moreover, this demonstrates that conventional designs that work fine with volatile memory may not work when using NVM. Ultimately, while eADR helps by removing the need for flushes, it does not save us from the challenges of designing correct and efficient persistent algorithms. Furthermore, there are still many systems with earlier generation hardware that are not equipped with eADR and we want our algorithms to function on these systems.

To summarize, with eADR, flushing is made trivial, but persistent algorithm design is not. This is similar to how fencing is made trivial by memory_order_seq_cst in C++, but concurrent algorithm design remains non-trivial. Our algorithm confers all of the same benefits with eADR, as without. In the following sections we specify flush and fence locations in our algorithm that suffice for systems without eADR, and on systems with eADR they can be safely omitted (as we have ordered writes to ensure data is not lost).

Correctness in NVM. Durable linearizability [40] is one of the most common correctness properties for persistent algorithms. It defines how to apply linearizability to models where crashes are possible. Intuitively, linearizability states that each operation appears to take effect atomically (as a single indivisible step) at some time during the operation. That time is called the linearization point (LP) for that operation. Moreover, each operation must return the same value as it would if all of the operations were run sequentially in the order of their LPs. Durable linearizability requires that any operation that completed before a crash will be reflected in the state of the object after recovery by requiring executions, excluding crashes, to be linearizable.

2.2 Transactional Memory

A transaction is a sequence of transactional accesses, (reads and writes), performed on a set of *transactional addresses*. A TM implementation provides operations to start a transaction, read and write transactional addresses, commit a transaction, and voluntarily abort a transaction. These operations are implemented using a set of *base objects*. If two transactions are concurrent and one or both of the transactions writes to an address that the other has already accessed then we say that these transactions have a *data conflict* (or simply that the transactions conflict). A transaction that aborts due to a conflict will typically be retried until it either succeeds and commits, or until it invokes a voluntary abort operation. A HyTM implementation additionally specifies whether a transaction is executed in hardware or software.

Hardware TM. We consider a system equipped with HTM with the same semantics as Intel's RTM [20]. Hardware transactions maintain a *tracking set* comprised of a *read set* and a *write set*. Conflicts are tracked via the cache coherence protocol. A conflict occurs whenever a thread writes to an address that is in the tracking set of an ongoing transaction. If two transactions conflict, *at least* one will abort, but hardware transactions can also abort spuriously (for any reason). Note that a non-transactional access can also abort a transaction.

While we model all involuntary aborts other than conflict aborts as spurious, a common cause of HTM non-conflict aborts is related to the bounded capacity of HTM read and write sets. For example, with Intel RTM, the write set of a hardware transaction is limited to the L1 cache. (Capacity aborts can occur when as few as 9 addresses are accessed on an Intel CPU with an 8-way associative L1-cache.)

To initiate a hardware transaction we utilize the `xbegin` instruction. Calling the `xend` instruction attempts to commit the hardware transaction. We can also explicitly (voluntarily) abort a hardware transaction with `xabort`. When a hardware transaction is aborted, control returns to the call of `xbegin` which will then immediately return a failure code signifying that the last hardware transaction aborted. Since hardware transactions can abort for any reason a fallback path for execution via software transactions is required to make any kind of progress guarantee.

One cause of spurious aborts is the use of flush instructions. On any hardware that we are aware of, flush instructions force hardware transactions to abort (although the specification of these instructions does not guarantee that they will abort a hardware transaction). Similarly, it is not entirely clear how eADR and HTM (specifically Intel RTM) are intended to function together. While hardware transactions appear to take effect atomically, each cache line modified during a hardware transaction must be transitioned to a state in which other threads can access it. The actual implementation details of Intel's RTM are proprietary. We suspect that caches lines modified during an RTM transaction are held in exclusive mode and are moved to shared mode once the transaction commits. Regardless of the specifics, for eADR to be of any use in this case,

the entire write set of the hardware transaction would need to be somehow marked as completed via a single atomic write and we are not aware of any evidence to suggest this is the case. Anything that performs multiple distinct updates, say to individual cache lines, would allow for a write back of partially completed hardware transactions. So, once again, eADR cannot save us. We must have a software based fallback path and therefore, we still have problems related to write ordering even with a persistent cache.

Correctness in TM. We consider the TM-correctness property of *opacity* [37]. Intuitively, opacity requires that the set of all concurrent transactions (including aborted ones) be equivalent to a sequential one in which every read returns the value of the latest written write. For PTMs, we say that a TM offers durable transactions, if it guarantees that the effects of all committed transactions will be reflected in the state of the transactional addresses following recovery.

Progress in STM. For STMs, the de facto *progress* properties are (weak) progressiveness and strong progressiveness. Intuitively, progressiveness states that if a transaction is aborted then there exists a concurrent conflicting transaction. Strong progressiveness states that if a set of concurrent transactions conflict, then at least one of them is not aborted.

In principle, progressiveness can be defined for hybrid TMs as well, but since current HTM implementations can abort transactions for any reason (and thus offer no progress guarantees), one cannot actually achieve progressiveness in HyTMs in practice. Ignoring spurious aborts allows us to reason in terms of these existing progress properties, however, this is unrealistic—it is surprisingly easy to write software in which all transactions will spuriously abort, forever.

2.2.1 A New Progress Guarantee for Hybrid TM. We suggest the following alternative: Think of a transaction as a sequence of *attempts* culminating in a *commit* or *voluntary abort*. One can then imagine a transaction having a bounded number of attempts that run on the hardware path, possibly followed by some number of attempts on the software path. We say that a TM is *C-abortable* weak (resp., strong) progressive if, for all executions of the TM, each transaction in an execution can abort unconditionally (for any reason) at most C times, after which any subsequent aborts by that transaction must satisfy the requirements of weak (resp., strong) progressiveness. When C is a constant we can also say that the TM is O(1)-abortable weak (resp., strong) progressive.

This generalizes these classical definition of progressiveness to deal with the reality of current HTM systems, without relying on an unrealistic assumption that spurious aborts can simply be ignored. Note that 0-abortable progressiveness is simply progressiveness (and similarly for strong progressiveness). As an example, a hybrid TM that first attempts a transaction up to 10 times on the hardware path before falling back to a progressive software path could be shown to be 10-abortable progressive.

2.3 Related Work

2.3.1 Volatile STMs. Volatile STMs were originally proposed by Herlihy and Moss [39]. Since then there has been extensive research on the topic of STMs [27–30, 36, 57]. There has also been attempts to integrate volatile TM into programming languages including Haskell, C++ and Java.

TL2. Transactional Locking II (TL2) [27] is one of the most well known volatile STMs. In the basic TL2 algorithm, each transaction first obtains a version by reading a global version number. This global version number is incremented by each transaction that performs writes. Every memory address is protected by a versioned lock that contains the version of the last transaction that wrote to it. Each transaction uses its version to *validate* the values it reads, ensuring that it only sees values written before it began.

The public implementation by Dice et al. uses *buffered writes* and deferred *commit time locking*, meaning the write set is locked only when the transaction attempts to commit. This is in contrast to *in-place writes* and *encounter time locking*. After acquiring the write set locks, the global version is incremented and the read set re-validated. This validation is skipped if, after the increment, the global version is only 1 greater than than the transaction's version. If the write set is locked in some total order (for example, by increasing memory address), then TL2 guarantees strong progressiveness. Numerous variants of TL2 have since appeared, suggesting different ways to update the global version number [48, 54].

2.3.2 Persistent STMs. There are many existing persistent STMs (PSTMs) [1, 12, 16, 21, 22, 35, 44, 55, 58]. These algorithms typically rely on techniques such as logging (both undo and redo logging), data replication and custom memory management mechanisms/allocators. We provide the details of the state of the art PSTM, Trinity.

Trinity. Ramalhete et al. presented Trinity, a technique that is combined with TL2 or flat combining [38] to produce a PSTM [55]. It is effectively undo-logging where the logs are colocated with user data in persistent memory. It exploits the fact that x86 processors guarantee that writes to the same cache line will never be persisted out of order [14]. Trinity uses a shadow data approach where a copy of user data exists in both volatile and persistent memory.

In Trinity, every word of memory that will be accessed transactionally is augmented with an adjacent replica word which the authors call *back* and an adjacent sequence number. Collectively, these fields must fit in a single cache line. The metadata is required only for recovery, so rather than having a one-to-one copy of all data in volatile and persistent memory, volatile memory need only contain user data while persistent memory must contain the user data along with the metadata.

Writes to any word must first update *back*, then the sequence number and finally the user word, after which the cache line containing these fields is flushed. At the end of a transaction a fence is performed after which the global sequence is incremented and persisted (requiring another flush and fence). After a crash one must search through persistent memory to find any annotated words with sequence numbers that differ from the global sequence number and revert each such word to its old value stored in *back*.

2.3.3 Volatile Hybrid TMs. There are many volatile hybrid TMs [3, 6, 23, 25, 45, 49, 51, 53, 56]. HyTMs often utilize a *two path* approach wherein hardware transactions represent the fast path and software transactions represent the fallback slow path. Some HyTMs such as hybrid NOrec and reduced hardware NOrec do not allow concurrency between hardware and software transactions [23, 49]. Others, such as the work of Damron et al. and that of

Kumar et al. allow for concurrency between hardware and software transactions [25, 45]. Brown and Ravi present two examples of two path HyTMs with different levels of concurrency between hardware and software transactions [6].

2.3.4 Persistent HyTMs. There is relatively little existing work on persistent HyTMs. Many proposed algorithms do not work with existing hardware and instead are based on proposals for new hardware or extensions to existing HTM [2, 4, 41, 42]. There are even fewer persistent HyTMs that are supported by existing hardware [7, 8, 34]. We describe SPHT, a state of the art persistent HyTM.

SPHT. Scalable Persistent Hardware Transactions (SPHT), which is based on NV-HTM [8], is a redo-logging based HyTM that runs on current hardware [7]. The software fallback path immediately claims a global lock and all hardware transactions read this lock and abort if it is held. SPHT logs operations in NVM and orders these logs using timestamps. When the logs are full they must be replayed in NVM, both to ensure that NVM is up to date, and to allow the logs to be emptied (avoiding the need for unbounded logs).

SPHT has a global *persistent marker* that stores the timestamp of the most recently persisted transaction. Each thread has its own persistent log, its own timestamp and its own persistent marker. The thread's timestamp is marked to indicate whether its last hardware transaction has been persisted. When a thread updates the global persistent marker it also updates its local persistent marker to match its timestamp.

Before beginning a new hardware transaction, the thread updates its timestamp and marks it as not persistent. During a hardware transaction writes are logged. At the end of the transaction the thread gets an updated timestamp. After the hardware transaction ends, the thread's log is written to NVM. The log must be ordered relative to concurrent transactions. There is a non-trivial algorithm for negotiating the ordering, involving threads collecting timestamp vectors and blocking until timestamps are marked. Notably, this means that transactions can be blocked by other concurrent transactions even if they access disjoint data.

2.3.5 Miscellaneous. SpecPMT [54] can be used to retrofit a program to ensure crash consistency, assuming the program *already* correctly synchronizes its threads [60]. SpecPMT is effectively an optimized form of undo logging. SpecPMT is neither a STM nor a hybrid TM since it does not offer any method of synchronizing between threads to guarantee atomicity, isolation or consistency (see 4.3.3 in [60]). These types of algorithms are only tangentially related to persistent hybrid TMs since TMs solve a much harder problem. Algorithms like SpecPMT that only guarantee crash consistency cannot be meaningfully compared against TMs. In fact, SpecPMT would crash on any program in our benchmarks since the benchmarks rely on TM for synchronization.

3 NV-HALT Design

In this section we present the details of our two-path persistent HyTM, NV-HALT. We begin with a simple overview of NV-HALT. We then describe the software path utilized by NV-HALT after which we discuss our hardware path and our approach to persisting transactions that execute in hardware.

```
tid; sRdSet; sWrdSet; pVerNum # Thread local data

def TxRead(addr):
 if found = sWrSet.find(addr): return found.val
 sRdSet.insert(addr, getLock(addr))
 if !validate(sRdSet): abort()
 return val

def TxWrite(addr, val):
 sWrSet.insert(addr, getLock(addr), val)

def TxCommit():
 if sWrSet.size == 0: return
 if !acquireLocks(sWrSet): abort()
 if !validate(sRdSet): abort()
 for (addr, val) in sWrSet:
   old = *addr; pA = vmemAddrToPmem(addr)
   pA.old = old; pA.pver = {tid, pVerNum}; pA.new = val;
   flush(&pA)
   *addr = val
 pVerNum++; flush(&pVerNum);
 releaseLocks(sWrSet)
```

Figure 1: NV-HALT software path. ■ code for persistence.

3.1 High Level Overview

NV-HALT is an O(1)-abortable progressive persistent HyTM that provides durable transactions and guarantees opacity. NV-HALT aims to overcome the challenges related to persisting transactions that execute in hardware while avoiding the shortcomings of the existing state of the art, SPHT. Unlike SPHT which sacrifices concurrency of disjoint transactions to enable persistent hardware transactions, our design allows disjoint transactions to execute concurrently. Moreover, in a crash-free execution, SPHT has added overhead related to ordering and replaying persistent logs; NV-HALT avoids this overhead. SPHT also relies on a global lock for its software fallback path which completely eliminates concurrency by blocking other transactions whenever the lock is held. NV-HALT offers a more practical approach utilizing a non-trivial software fallback path that allows transactions on the software path to execute concurrently with transactions on the hardware path. This is especially important for update heavy workloads which we discuss further in § 5. With these goals and considerations in mind, we now discuss the high level details of NV-HALT.

NV-HALT is a word based TM with a hardware fast path and a software fallback path. NV-HALT attempts transactions a fixed number of times in hardware before falling back to a progressive software path, meaning NV-HALT is O(1)-abortable progressive. We utilize versioned locks to protect transactional addresses. These fine grained locks serve a dual purpose in NV-HALT. Broadly speaking, the fine grained locks are used to guarantee consistency by ensuring threads synchronize on the locks before modifying or reading a transactional address. The fine grained locks are also used to enable durable transactions. Our persistence mechanism exploits the fact that a transactional address can be non-durable only while its corresponding lock is held.

3.2 Software Path Transactions

The software path of NV-HALT instruments both reads and writes. The pseudocode for the software path of NV-HALT is shown in Figure 1. Throughout this pseudocode, we omit the implementations

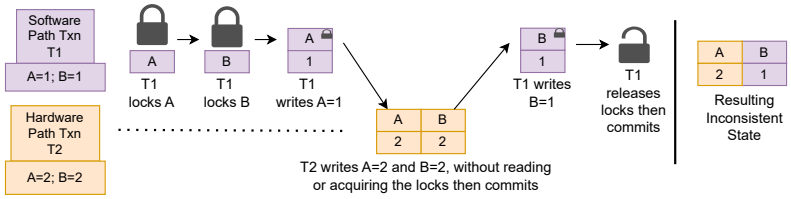

Figure 2: Execution of a HyTM with a lock based software path and hardware path that ignores metadata leading to an inconsistent state. This execution violates opacity because the resulting state could not be produced by any sequential execution.

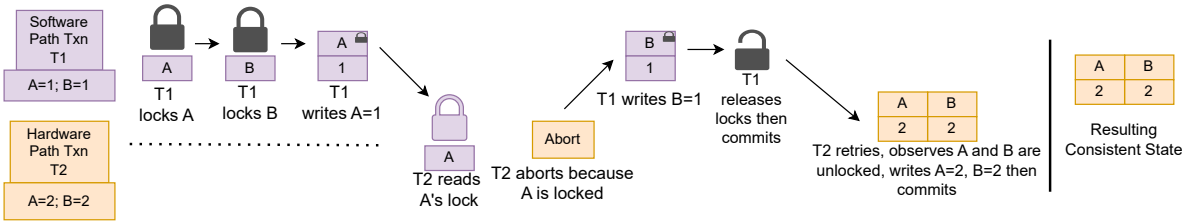

Figure 3: Example execution of a corrected version of the HyTM shown in Figure 2 where, in the volatile setting, instrumenting the hardware path to read shared metadata is sufficient to guarantee opacity.

of various simple helper functions when the implementation can be inferred from the name.

During a software path transaction, threads track read and write sets which, for each access, log the address and its associated lock version. NV-HALT utilizes deferred (commit time) locking for writes on the software path meaning that the actual writes are buffered until commit time. Prior to commit time while accumulating the write set, the thread will check to ensure that the lock corresponding to the address that it plans to write is unlocked, aborting otherwise.

Transactions also maintain a read set. The read set is revalidated on each read, and the transaction aborts if validation fails. Validation can fail if an address in the read set is currently locked (by a different thread) or if its associated lock version has changed since the thread first encountered it.

At commit time, threads on the software path attempt to acquire their write set locks. Each entry in the write set will contain the encounter time version of the lock, that is, the version that the thread observed when it first encountered the associated address during the transaction. A thread acquires a lock by attempting a *compare and swap (CAS)* on the lock's version to swap it to one greater than the encounter time version. Once all of the write set locks have been acquired, the thread revalidates its read set. If successful, the thread performs the writes, then releases the write set locks by incrementing the lock versions a second time.

Persisting Software Transactions. The effects of all committed transactions need to be persisted. A transactional address modified by a software path transaction will be non-durable for some period of time after it is written to and before it is written back to persistent memory. The difficulty in persisting software transactions is primarily related to the necessary synchronization in volatile memory. More specifically, we need to ensure that threads are blocked when attempting to access non-durable data. With correct synchronization, the process of persisting the data is straightforward.

Since writes on the software path are buffered until the thread has claimed all of its write set locks, a transactional address will become non-durable at some point while the address is locked (and the transaction is guaranteed to commit). If we ensure that the all addresses in the write set are written back to persistent memory before we release the write set locks then we can guarantee that other threads will always observe durable data (so long as they respect the address locks). This approach only adds overhead to the commit phase of software transactions that write.

While we hold the write set locks, we utilize the undo logging technique of Trinity to persist the write set. Note that the *back* and a version number fields used by are only required for recovery and do not need to be stored in volatile memory. With this approach each thread has its own (thread local) sequence number. We will refer to this as the thread's *persistent version number*. Persisting the write set follows the same steps as in Trinity. We reiterate these steps below.

When applying the write set logs of a transaction we iterate over each entry in the write set and do the following: first, we get the persistent memory address corresponding to the entry. This gives us the address of the cache line that we need to modify and write back to persistent memory. Next, we update this cache line by writing the old value, a tuple combining the thread's persistent version number and thread ID, and the new value. We need to combine the thread ID and the thread's persistent version number since multiple threads might have the same version. We then flush the cache line after which we write the new value in volatile memory. After persisting the write set, we increment the thread's persistent version number and flush it to NVM.

3.3 Persisting Hardware Transactions

Before we describe the hardware path of NV-HALT, it is helpful to understand the minimum synchronization and instrumentation

needed by the hardware path in order to correctly allow for persisting hardware transactions. The amount of required hardware path instrumentation differs between the volatile and persistent setting.

In the volatile setting, it is known that for a (C-abortable) progressive HyTM guaranteeing opacity requires the hardware path to access shared metadata [6]. Figure 2 shows an example execution of a HyTM with a software path which relies on metadata in the form of fine grained locks as in the software path of NV-HALT. The hardware path in this example HyTM does not access any metadata. As a result of this, the given execution results in an inconsistent state violating opacity. If we want a progressive HyTM that guarantees opacity then transactional accesses on the hardware path must be instrumented in some manner.

The obvious next question is how much instrumentation is necessary? The execution in Figure 2 specifically demonstrates a need for instrumenting the hardware path to read shared metadata. Figure 3 depicts the corrected example HyTM which guarantees opacity by forcing the hardware path to read the locks. In the volatile setting, there is no need for the hardware path to write to metadata. This is because a hardware path transaction that writes to some address X will be aborted if any other concurrent transaction (hardware or software) also writes to X. For any hardware path transaction, in any execution, the hardware path transaction will commit iff any conflicting writes occur strictly before the hardware transaction begins or strictly after it commits. Either way it is easy to construct an equivalent sequential execution meaning opacity will not be violated. Note that so far we have ignored the possibility of crashes.

In the persistent setting, we require more hardware path instrumentation. There are two challenges related to persisting the effects of transactions that execute in hardware. The first is the fact that current implementations of flush instructions force hardware transactions to abort. As a result of this, the effects of a hardware transaction can only be written back to persistent memory after the hardware transaction has completed (i.e. after it successfully returned from xend()). This inherently requires hardware transactions to keep track of a write set, otherwise we would not know which addresses need to be written back to persistent memory. However, this represents instrumentation to write to private metadata.

Correctly persisting hardware path transactions is not as simple as tracking a write set. Figure 4 shows an example execution of a HyTM demonstrating that, in the persistent setting, instrumenting the hardware path to read shared metadata is not sufficient to guarantee opacity after a crash. This execution specifically depicts a problem that arises when hardware path transactions do not utilize any mechanism to synchronize persisting modified addresses after the hardware transaction has completed. Avoiding this requires the hardware path to write to shared metadata. With this in mind we can now discuss the hardware path of NV-HALT which uses hardware assisted locking.

3.4 Hardware Assisted Locking

The hardware path of NV-HALT is shown in Figure 5. As we have already mentioned addresses cannot be flushed inside of a hardware transaction. To this end, NV-HALT instrument writes to keep track of all addresses modified by the hardware transaction using a thread-local append-only log. After the hardware transaction has completed, we can persist all of the addresses in the log since, the transaction will no longer be executing in hardware so we are free to flush as needed.

As with the software path, the difficulty in persisting the modified addresses is the synchronization needed to ensure that other threads do not access non-durable data and to ensure that the state of persistent memory remains consistent. Addresses modified by a hardware transaction will be non-durable immediately after the hardware transaction completes. Moreover, there is a theoretically infinite amount of time between the hardware transaction completing and the next instruction performed by the thread. This means that addresses modified by a hardware transaction need to be protected after the hardware transaction has completed. This also means that whatever mechanism we utilize to prevent concurrent threads from accessing data modified by our hardware transaction must be activated during the hardware transaction. On the software path, we utilize the fine grained locks to protect addresses while they are non-durable. This raises the question, why not apply a similar approach on the hardware path? This leads to our approach to persisting hardware transactions: hardware assisted locking.

Our hardware path further instruments writes to acquire the associated fine grained lock. Before acquiring a lock, the transaction must confirm that it is unlocked or locked by the current thread. Importantly, during the hardware transaction, the thread will only acquire locks. This means that the addresses will remain locked even after the hardware transaction completes. After a hardware transaction completes, with the addresses locked, we proceed as in the software path by persisting the write set, updating the thread's persistent version number and releasing the locks in that order.

The hardware path of NV-HALT also instruments reads to check that the associated lock is unlocked (or locked by the current thread). This is necessary both to ensure that hardware path transactions and software path transactions can execute concurrently without violating opacity and to ensure that hardware transactions do not observe non-durable data. One would likely find this counterintutive since, in a volatile only setting, we would prefer to avoid instrumenting reads on the hardware path. However, in a non-volatile setting, we have limited other options. Allowing hardware transactions to claim locks and hold them after the hardware transaction completes necessitates instrumenting hardware path reads. This is true even if we disallow concurrency with software path transactions (for example by using a global lock fallback path like SPHT). One could instead choose an approach like SPHT. This introduces an interesting trade-off. One can trade instrumenting hardware path reads and a non-trivial software path (NV-HALT) for uninstrumented reads, allowing disjoint transactions to block each other and a trivial global locking fallback path (SPHT). We argue that the former (hardware assisted locking) is preferable.

3.5 Recovery

Since we persist individual addresses via Trinity's persistence mechanism, our recovery procedure follows the same approach as in Trinity. This involves traversing persistent memory and reverting to their old value any addresses that have a version number greater than the version number of the thread that last modified the address.

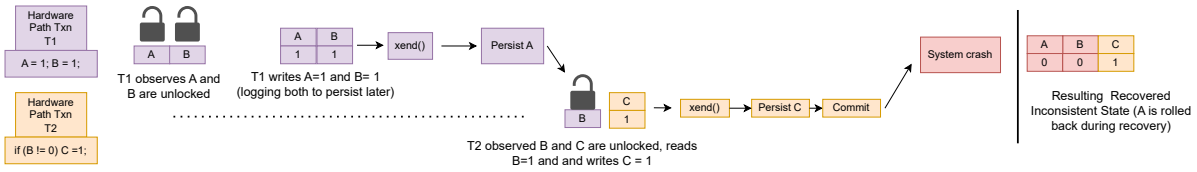

Figure 4: Example execution of the same HyTM as Figure 3 demonstrating that instrumenting the hardware path to read shared metadata is not sufficient to guarantee opacity in the persistent setting.

```
1  def TxRead(addr):
2   lock = getLock(addr)
3   if isLocked(lock) and lock.owner != tid: xabort()
4   return *addr
5
6  def TxWrite(addr, val):
7   if !htmAcquireLock(getLock(addr): xabort()
8   hWrSet.insert(addr, *addr)
9   *addr = val
10
11 def TxCommit():
12  xend()
13  for (addr, old) in hWrSet:
14   pA = vmemAddrToPmem(addr)
15   pA.old = old; pA.pver = {tid, pVerNum}; pA.new = *addr;
16   flush(&pA);
17  pVerNum++ flush(&pVerNum); releaseLocks(hWrSet)
```

Figure 5: NV-HALT hardware path. ■ code for opacity and persistence. ■ code for persistence only.

3.6 Correctness and Progress

Theorem 3.1. *NV-HALT offers durable transactions, guarantees opacity and O(1)-abortable weak progressiveness.*

Aborts on the software path of NV-HALT occur when a transaction fails to claim its write set locks or fails to validate its read set. Both of these scenarios can occur iff some other concurrent transaction has claimed a lock meaning the transactions conflict. In these scenarios there is no guarantee that one of the conflicting transactions will commit so the software path is progressive. Since NV-HALT attempts transactions a fixed number of times in hardware, the means NV-HALT guarantees O(1)-abortable weak progressiveness.

NV-HALT guarantees opacity since it is prevents leaking any information from aborted transactions. No transaction can observe the effects of a concurrent transaction that aborts since all modifications are made only once the transaction has claimed its write set locks and is guaranteed to commit.

As in Trinity, following a crash an address that was last updated by a committed transaction will have a version number that is less than the persistent version number of the owning thread (all other addresses will be reverted to their old values during recovery).

Guaranteeing C-Abortable Strong Progressiveness. It is possible for a progressive TM to experience live locking. Consider a simple array being accessed by two threads as shown in Figure 6. Here, we have one transaction, T_1, reads all elements in ascending index order then updates the last element while another concurrent transaction, T_2, reads all elements in descending index order and updates the first element. A C-abortable progressive TM like NV-HALT would consistently abort both transactions since both T_1 and T_2 will consistently fail to validate their read sets after acquiring all of their write set locks. Similarly, if both T_1 and T_2 wanted to update every element neither would be able to acquire their write set locks since they will eventually need a lock that the other has claimed. A C-abortable strongly progressive TM avoids these issues.

Guaranteeing O(1)-abortable strong progressiveness requires our software path to be strongly progressive. The latter scenario where a C-abortable progressive TM can live lock can be avoided by acquiring write set locks in a fixed order similar to TL2. In the former scenario, the two transactions are both about to claim their write set locks but both abort. To avoid this, we can use a global *clock*. Transactions start by reading the clock. The clock is incremented after acquiring the write set locks. Read set validation is not necessary if, after the increment, the clock is exactly one greater than the value read at the start of the transaction, (i.e. no concurrent transactions have performed any writes).

In our case, the use of a global clock on the software path is not too concerning, however, this would be a major bottleneck if we were to increment a shared global on the hardware path. To avoid this, we instead change the definition of the fine-grained locks to include two version numbers. The first, which we will call `sLock`, remains the same. It is incremented whenever a transaction (software or hardware) acquires or releases the lock. The added version number, which we will call `hLock`, is incremented only by hardware transactions. Software transaction can use the `hLock` version to identify conflicts with concurrent hardware transactions.

To summarize, we utilize a global clock on the software path where, at the beginning of the transaction the thread reads the clock and, after acquiring its write set locks, attempts a CAS to advance the clock by one greater than the value it read at the start. If the CAS is successful, the transaction must check for conflicts with concurrent hardware transactions by iterating over its read set to confirm that the `hLock` versions have not changed. If the CAS fails then the transaction performs standard read set validation, checking that the `sLock` versions have not changed. This gives us a O(1)-abortable strongly progressive version of our persistent HyTM. We refer to this version as NV-HALT-SP (StrongProgressive). We show the pseudocode for the changes compared to NV-HALT in Figure 7. For the hardware path, the only difference is the addition of line 5.

4 Implementation Details

In this section we discuss some relevant technical details of NV-HALT. We begin by discussing how we allocate and manage memory within transactions.

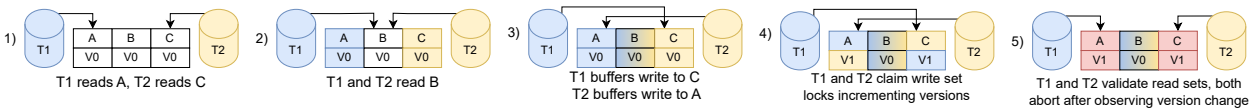

Figure 6: The need for strong progressiveness: A partial execution of the O(1)-abortable (weakly) progressive NV-HALT where two transactions will abort forever.

```
global gClock ; thread_local rClock  #new fields

def htmAcquireLock(lk):          #H/W Path Changes
 if isLocked(lk): return false
 lk.sLockVer++; lk.hLockVer++;
 return true

def TxStart(): rClock = gClock  #S/W Path Changes

def TxCommit():
 if sWrSet.size == 0: return
 sort(sWrSet)
 if not acquireLocks(sWrSet): abort()
 if CAS(gClock, rClock, rClock+1):
  if foundHtxConflict(sRdSet): abort()
 elif not validate(sRdSet): abort()
 #... same as 16-22 in Figure 1
```

Figure 7: Pseudocode for the O(1)-abortable strongly progressive version of NV-HALT.

4.1 Memory Allocation in Transactions

Memory management within a TM is an important and challenging task. The allocation and freeing of memory needs to be directly tied to transaction commits and aborts. Otherwise a transaction that aborts could leak memory or free memory that is still in use.

One could choose to ignore the problem simply by never freeing memory. Allocations within transactions that abort would leak memory and any committed deallocations are effectively no-ops which also leak memory. Obviously this is not ideal, but it is not too uncommon to defer solving the problem of safe memory reclamation. In fact, SPHT leaks memory since it never frees memory. This is particularly problematic when using NVM since memory leaks are essentially persistent forever. If we wanted to utilize this algorithm in practice we would need to solve the problem of correct memory reclamation.

There are two approaches to actually managing memory within transactions. One method is to utilize an allocator where the internal state is accessed via transactions (meaning the TM is used to implement the allocator). This is the approach taken by Trinity. With durable transactions, this strategy allows one to recover the allocator state following a crash. However, this approach increases the write set size of transactions and can also cause additional data conflicts. For HyTM algorithms like NV-HALT, this can lead to increased transaction aborts, especially on the hardware path.

Quantifying the Cost of Trinity's Allocator. We attempt to quantify the additional cost imposed by accessing allocator metadata through transactions in Trinity. One might think that the best way to quantify the cost would be to compare Trinity with another version in which we utilize a different allocator that does not rely on transactions; however, using a different allocator would lead to changes in the memory layout, which can drastically affect performance. Instead, we constructed some controlled workloads and used the authors publically available implementation of Trinity combined with TL2 (TrinityVRTL2 in [55]) to extract some easily quantifiable metrics that summarize the cost of Trinity's approach to memory management.

The workloads that we utilize perform the following common actions: First, threads execute transactions to populate an array of pointers to 16-byte objects. Each thread will access a randomly generated and disjoint subset of the array. When accessing an element in the array, a thread executes a single transaction which will perform 3 transactional stores. One store writes the pointer into the array and two stores update the data fields of the object. We refer to these types of transactions as *type A transactions* (for Alloc). Once the array has been populated, the threads access a new randomly generated and unique subset of the array. At each access a thread will again perform 3 transactional stores. Two stores update the data fields of the object and one store removes the object from the array by replacing the pointer with `null`. We refer to these types of transactions as *type F transactions* (for Free). We utilize an array of 1 million objects, meaning that we will execute 2 million transactions. It is important to note that since each thread accesses a unique subset of the array, this workload has no data conflicts outside of those that might arise in the allocator. This means that there we will be 0 aborts excluding those that are caused by accesses made by the allocator (and those that can arise due to collisions in the TL2 lock table).

We use this template to construct 3 different workloads that evaluate Trinity's approach to memory management. In the first workload, we use Trinity's allocator to perform allocations and deallocations. In type A transactions, each thread will allocate a new object to insert into the array. In type F transactions, each thread will free the object that they remove from the array. This workload captures the full cost of Trinity's approach to memory management. In the second workload, we utilize Trinity's allocator to perform allocations but we do not perform any freeing. Here type A transactions will again require threads to allocate a new object but type F transactions are the same as in the template. In the last workload, we preallocate all objects before the measurement period begins (these allocations still use Trinity's allocator but they do not contribute to our data collection). In this case, both transaction types are unmodified compared to the template (meaning there are no allocations or frees in any measured transactions). Note that in all 3 cases, outside of allocator functions, the number of transactional accesses is equivalent.

Table 1 shows the results of this experiment for the machine described in § 5 using 64 threads. In particular, we show the averages of the number of aborts, the average read and write set sizes overall commits and aborts, as well as the maximum read and write set size observed at an abort. One alarming difference is the significantly larger number of aborts when accessing allocator metadata via transactions. We observe nearly a 100x increase in aborts when we utilize the full allocator functionality. This result is especially

Trinity Allocator Functions Accessed via Transactions	Average Aborts	Average Data Set Sizes				Max Data Set Sizes			
		Overall		At Abort		Overall		AtAbort	
		Read Set	Write Set	Read Set	Write Set	Read Set	Write Set	Read Set	WriteSet
Both Alloc and Free	186338.4	4.86	6.02	7.80	5.68	15	13	13	8
Alloc only	10504.5	2.50	4.48	2.86	0.45	7	9	5	6
None	2153.7	2.50	2.99	0.55	0.75	1	3	1	3

Table 1: Evaluation of the cost of accessing allocator metadata via transactions in Trinity. Each row shows the results from running one of the synthetic array modification workloads described in § 4.1 using 64 threads. All workloads execute 2 million transactions. Excluding allocator accesses, there are 0 data conflicts and each transaction performs 3 transactional stores.

important since these aborts would never happen if the allocator were not accessed transactionally. We can also see that the average and maximum read and write set sizes are much larger for this workload. One might expect the third workload where we do not utilize Trinity's allocator to have exactly 0 aborts. All aborts in this case are a result of collisions in the TL2 lock tables. These results demonstrates that there is a significant cost associated with Trinity's memory management strategy.

NV-HALT Memory Management. In NV-HALT, we utilize a custom memory management system where synchronization is achieved without the use of transactional accesses. Our algorithm ensures that memory allocated during a transaction is freed if the transaction ultimately aborts and defers freeing memory at least until the transaction that tried to free has committed. To achieve this, we utilize a custom memory allocator based on mimalloc [47] combined with epoch-based reclamation to ensure safe memory reclamation. When a user tries to allocate memory in a transaction, the allocation occurs immediately, but we keep track of these allocated objects so that we can reclaim them if the transaction aborts. Deallocations on the other hand are handed via epoch based reclamation. At the start of each transaction, the thread will read a global epoch counter and post its current epoch to a global announcement array. When a transaction commits, threads can read the global epoch to determine if it is safe to reclaim retired objects and threads can read the announcement array to determine if it is safe to advance the global epoch counter. This strategy allows us to avoid the performance concerns that exist with Trinity's allocator while still guaranteeing that we do not leak memory.

Similar to many algorithms that utilize NVM, our allocator initially maps a large contiguous address range in virtual memory, and then distributes it to various threads, on demand. The use of a contiguous address range allows for a direct mapping between volatile and persistent memory addresses. Unlike Trinity, the internal state of our allocator is not persisted. This means that we need to reconstruct the allocator state during recovery. To make this possible, the user must provide an iterator that the allocator can utilize to determine which parts of memory are in use. Trinity's approach optimizes for faster recovery whereas our approach optimizes for improved performance during regular crash free execution. We argue that ours is preferable since system crashes model power failures, which are assumed to occur infrequently.

4.2 Fine-Grained Locks

NV-HALT relies on fine-grained locks. Similar to TL2, we utilize a fixed size lock table. This means multiple addresses might map to the same lock, but it ensures that the memory layout of user data is unaffected. In some cases it is favorable to colocate the locks with user data. In this case each address has a unique lock. Depending on the layout of user data, this also allows prefetching locks when caching the associated user data. Our NV-HALT-CL implementation utilizes colocated locks. We achieve this via the use of custom types. For languages where custom types are not supported other strategies such as compiler extensions could be used to colocate the locks.

5 Evaluation

We compare the performance of NV-HALT against the state of the art PSTM, TrinityVR-TL2, which is Trinity combined with TL2 (we refer to this as Trinity), and the state of the art persistent HyTM, SPHT. We test the performance in terms of throughput, in operations per second, across various workloads. We utilize the publicly available source code for both Trinity and SPHT. We used the same benchmark as [5] and [13] for evaluating the TMs. We ran all experiments on a NUMA system with 2 Intel Xeon Gold 5220R 2.20GHz processors, each of which has 24 cores and 48 hardware threads. The system has a 36608K L3 cache, 1024K L2 cache, 64K L1 cache and 1.5TB of NVRAM. The NVRAM modules installed on the system are Intel Optane DCPMMs [15] configured in app-direct mode. We implemented all versions of NV-HALT in C++. The code was compiled with GCC 10.3.0 with an optimization level of -O2. We plan to make the code public after publication. In each trial we prefill the data structure to 50% capacity before beginning performance measurements. The measurement period of each trial is 20 seconds. We report the average of 5 trials.

5.1 Comparing to State of the Art PSTM

One may wonder why we compare our persistent HyTM against the state of the art for PSTM, Trinity. It has been shown to perform well for various workloads and we have already discussed the added overhead related to persisting hardware transactions. Comparing against the state of the art PSTM provides a better perspective on whether or not our design is effective despite these overheads.

(a,b)-Tree. Figure 8 shows the throughput of NV-HALT and Trinity for an (a,b)-tree with a=4 and b=16 where keys are accessed according to a uniform distribution. NV-HALT significantly outperforms Trinity for workloads dominated by update operations. For workloads dominated by read-only transactions, Trinity achieves throughput more comparable to NV-HALT Updates to the (a,b)-tree involve expensive rebalancing operations. As a result, hardware transactions are more likely to experience conflicts causing aborts

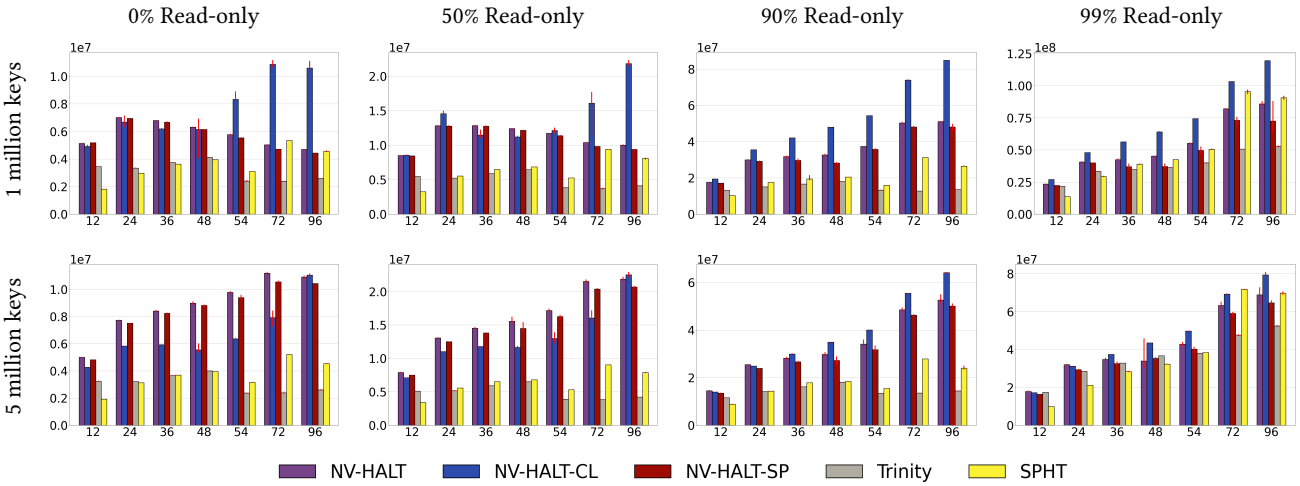

Figure 8: Throughput for (a,b)-tree with uniform key access pattern. Y-axis is ops/sec. X-axis is number of threads.

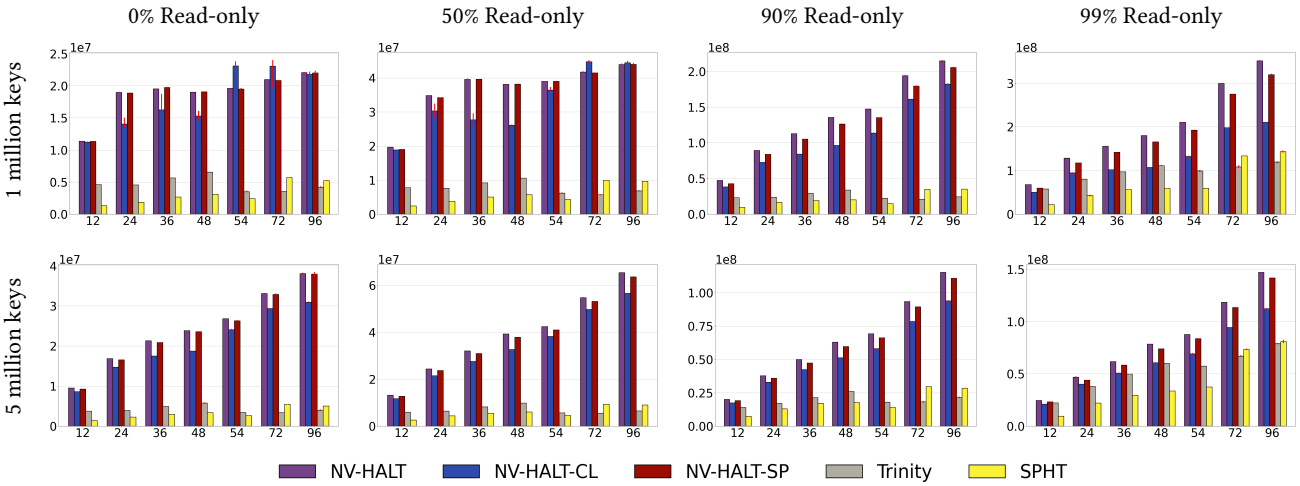

Figure 9: Throughput for hashmap with uniform key access pattern. Y-axis is ops/sec. X-axis is number of threads.

forcing more transactions onto the software path. In this experiment NV-HALT performs best when using using colocated locks. Another interesting observation is that our O(1)-abortable strongly progressive version performed well compared to the O(1)-abortable progressive alternatives. We also show the results for this (a,b)-tree when keys are accessed according to a zipfian distribution in Figure 10. The results are similar to the experiment with a uniform key access pattern. Even when the key access pattern is skewed, NV-HALT consistently outperforms the existing state of the art, especially for the update-heavy workloads.

Hashmap. Figure 9 shows the throughput of NV-HALT and Trinity for a hashmap with a fixed number of buckets corresponding to the number of keys. In this case NV-HALT outperforms Trinity for all workloads across all thread counts. With this configuration, hardware path transactions are unlikely to experience conflicts. In this case, due to the layout of user data, the use of colocated locks negatively effects throughput. As with the (a,b)-tree, we also tested the hashmap when keys are accessed according to a zipfian distribution. We omit plots for the hashmap with zipfian access pattern due to space constraints. The results were similar to the uniform access experiment, with NV-HALT performing better in all workloads.

5.2 Comparing to State of the Art Persistent HyTM

SPHT is the state of the art persistent HyTM. We compare against SPHT with its forward linking optimization enabled. The timestamp based synchronization used by SPHT is very much unlike our hardware assisted locking approach. We integrate the author's publicly available implementation of SPHT into our benchmark. SPHT utilizes a very simplified custom memory allocator to allocate memory during transactions. This allocator has artificially low overhead as it does not implement freeing. A thread in SPHT allocates memory from a fixed sized per-thread pool simply by incrementing a pointer. In practice this would make SPHT unusable. Unfortunately, the log replay mechanism utilized by SPHT directly

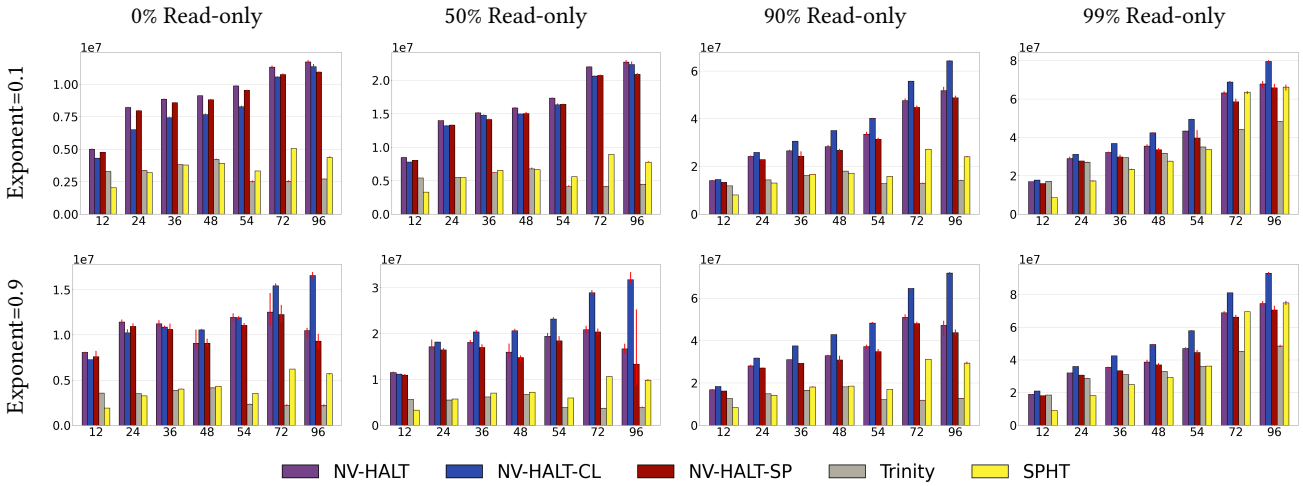

Figure 10: Throughput for (a,b)-tree with max 1 million keys. Y-axis is ops/sec. X-axis is number of threads. Keys are accessed according to a zipfian distribution with the exponent 0.1 (first row) and 0.9 (second row).

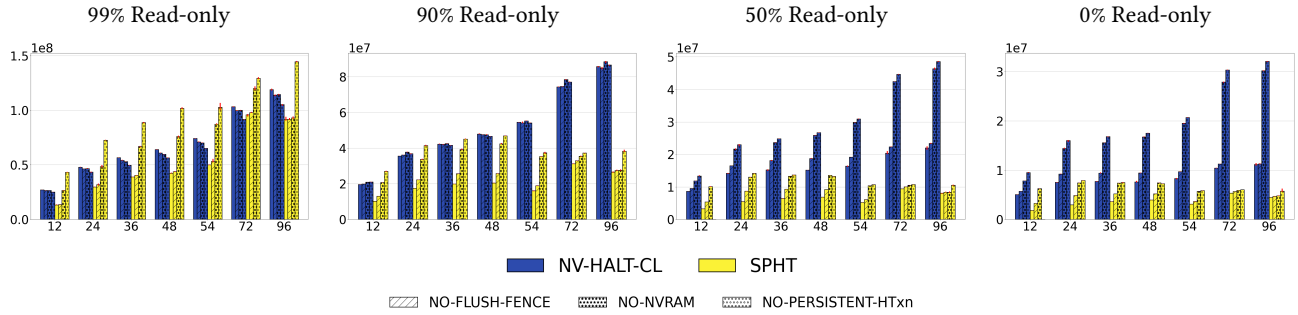

Figure 11: Ablation study of NV-HALT-CL and SPHT comparing their base implementation to implementations with progressively fewer enabled features for the same (a,b)-tree as row 1 of Figure 8. Y-axis is ops/sec. X-axis is number of threads. Keys are accessed according to a uniform distribution.

relies on this custom allocator, so we cannot modify the allocator or utilize a different allocator without impacting log replay.

The log replay mechanism of SPHT must be configured since even in a crash-free execution, the log will eventually need to be replayed. In [7], the author's show that the replay mechanism does not scale well. As SPHT is configured out of the box, it does not replay the log until the TM is shutdown which occurs only after the regular benchmark operations have completed. Recall that, system crashes model power failures which occur infrequently. When crashes are infrequent, it is not realistic to delay log replay until the TM is shutdown. Consider a long-running server application. In this setting, the logs would need to be replayed at runtime or else they will become full and cause threads to block. We enable SPHT's concurrent replay setting to allow a single background thread to replay the logs concurrently with regular benchmark operations. We attempted to utilize more than 1 concurrent log replay thread but this resulted in SPHT producing errors. We are engaged in private communication with the author, but the cause of the error is not yet known. For our microbenchmarks, we configure SPHT to utilize a size of 512MB for each per-thread log (48GB in total at the max thread count of 96). This log size is likely much larger compared to what one would want to use in practice. A larger log reduces the time that worker threads can be blocked by the replay mechanism. By default, SPHT sets the log size to 2GB per thread, however, we believe that this is an unrealistic configuration. At 2GB per thread, SPHT can typically avoid ever being blocked by the the log replay in our microbenchmarks. Moreover, this would require 192GB of memory just for the log for a single data structure in our benchmarks at max thread count on our machine. This is significantly worse for larger machines. For example to utilize all threads in an 8 socket system with 8x Intel 56 core Xeon 8480+ processors would require 1.75TB of memory just for the log in a single program that uses SPHT. Our target log size was 2MB per thread but this resulted in SPHT exceeding the 4 minute timeout used in our benchmarks. 2MB is equivalent to a typical operating system huge page. It is also a common page size and default limit for per-thread buffers in many allocators. For example 2MB is the default size for thread caches in tcmalloc. 512MB per thread is a generous size that reveals a small fraction of the cost of SPHT's replay mechanism, whereas 2GB completely obscures this cost.

(a,b)-Tree. NV-HALT outperforms SPHT for all workloads in the (a,b)-tree microbenchmark. SPHT's performance is best when the workload is read-dominant. This is not surprising. For the 99% read-only workload there are fewer hardware aborts since conflicts

are less likely. This means that SPHT's trivial software path is rarely utilized. In this workload SPHT also benefits from having uninstrumented hardware path reads. However, when we have even a moderate amount of update operations, NV-HALT significantly outperforms SPHT, especially at higher thread counts. There are two reasons for SPHT's lower throughput. First, in these workloads, aborts of hardware transactions are more common. In this case, the software fallback path utilized by SPHT can become a bottleneck since it immediately claims a global lock. Second, these workloads showcase the expensive cost of the synchronization mechanism utilized by the commit phase of SPHT as well as the cost of replaying the persistent logs. Both of these represent overhead that does not exist in NV-HALT. Moreover, NV-HALT has non-trivial overhead related to memory management which SPHT artificially avoids.

Hashmap. NV-HALT outperforms SPHT for all workloads and all thread counts for the hashmap microbenchmark especially for workloads with more update operations. The read and write set size of transactions in the hashmap is typically small which leads to fewer capacity related aborts. Remove operations in this hashmap mark nodes as empty rather than freeing them. This is somewhat of a more fair comparison in terms of memory management since neither algorithm will be performing memory reclaimation (memory allocation is still trivial in SPHT and non-trivial in NV-HALT).

5.2.1 Ablation Study. We can categorize overhead of a persistent HyTM into 3 classes: 1) the overhead of flush and fence instructions. This represents the cost necessary to guarantee writes in volatile cache memory are written back to NVM. Note that a system eADR would not require these instructions. 2) the overhead of reading and writing to NVRAM. This represents the cost of persisting transactions once the transaction is guaranteed to commit. In other words, this is the cost of the persistence mechanism without considering the cost of synchronization. 3) and finally the overhead of additional volatile synchronization. This is the cost of enabling the TM to persist hardware path transactions. For the PTMs that we evaluate there is no additional synchronization needed to persist software path transactions.

To understand these costs for NV-HALT and SPHT we compare the throughput using implementations where we progressively remove each overhead. The results are shown in Figure 11. For each TM the left most bar is the base implementation with all features enabled. Each successive bar removes a feature. NO-FLUSH-FENCE removes overhead class 1 by using no-ops for flush and fences. NO-NVRAM removes overhead class 1 and 2 by extending the previous implementation to also mmap all memory utilized by the TM in DRAM instead of NVRAM. NO-PERSISTENT-HTxn removes all three overhead classes by extending the previous implementation to also remove any synchronization specifically necessary to allow persisting hardware path transactions.

For both TMs, in the 99% read-only workload, there is little difference in throughput between any of the implementations. This is not surprising, when we perform fewer writes we engage with the mechanisms for persisting transactions less so we do not pay their cost. For the 90% read-only workload we can see some difference in throughput as we disable features particularly at larger thread counts. This result is also true for the other update-heavy workloads. For these workloads, NV-HALT has a larger increase in throughput when we disable the use of NVRAM. Neither TM gains much throughput when the third overhead class is also removed.

A naive interpretation of this result would be to assume that the persistence mechanism of NV-HALT is inferior, however, NV-HALT achieves higher throughput compared to SPHT for these workloads. This demonstrates the importance of the underlying volatile synchronization of the TM. In these workloads hardware aborts more common. SPHT's synchronization becomes a major bottleneck. Under high contention, SPHT can spend upwards of half of the entire measurement period in the fallback path where concurrency is disabled. If there are many concurrent hardware path transactions that write and commit, SPHT will block some of these transactions (even though their write sets are disjoint). Thus, these results demonstrate that when there is data to persist (10% or more updates) SPHT is close to the volatile only throughput but this maximum is still lower than throughput of the persistent NV-HALT. This shows that NV-HALT will benefit more from advancements in NVRAM technology and it reinforces our suggestion that hardware assisted locking is preferable compared to approaches like SPHT.

5.2.2 STAMP. We compare NV-HALT to SPHT using some of the STAMP [50] benchmarks. Specifically, we show the kmeans-high, intruder, labyrinth and ssca2. We omit the other STAMP benchmarks where SPHT experienced crashes. The STAMP benchmarks are most interesting from the perspective of volatile synchronization. STAMP was not designed with persistence in mind. This manifests in the benchmarks in various ways that can effect the performance and correctness of a persistent TM. For example, there are various places in the STAMP benchmarks where stack allocated memory or memory allocated outside of a transaction via the system allocator is later accessed within a transaction. None of the PTMs that we know allow for recovering data that was stack allocated. Likewise, PTMs including NV-HALT, SPHT and Trinity cannot persist data allocated by the system allocator. In this case, we can simply replace the calls to `malloc` and `free` with calls to the TM allocator. As with our microbenchmarks, there is still a significant difference between NV-HALT and SPHT with regards to memory management. All allocations by SPHT utilize their trivial allocator and all frees are no-ops whereas NV-HALT performs a non-trivial amount of work for allocating and freeing.

The benchmarks we tested differ in terms of data set size and contention. Transactions in the labyrinth benchmark have the largest data set sizes while transactions in the kmeans and ssca2 benchmarks have the smallest. The data set size of transactions in the intruder benchmark is lower than that of labyrinth and higher than that of kmeans and ssca2. Labyrinth and intruder are high contention benchmarks while ssca2 and kmeans are low contention. Though we use the higher contention version of kmeans, it is still low compared to labyrinth and intruder. When executing the STAMP benchmarks we utilize a timeout of 4 minutes for all algorithms, which is multiple minutes beyond the worst case runtime for NV-HALT in any workload. With our previous microbenchmarks we allow SPHT to obscure the log replay by using a large per-thread log size. For the STAMP benchmarks, we attempt to get a better portrayal of the cost of SPHT's log replay mechanism by using a lower per-thread log size, specifically 8MB per thread.

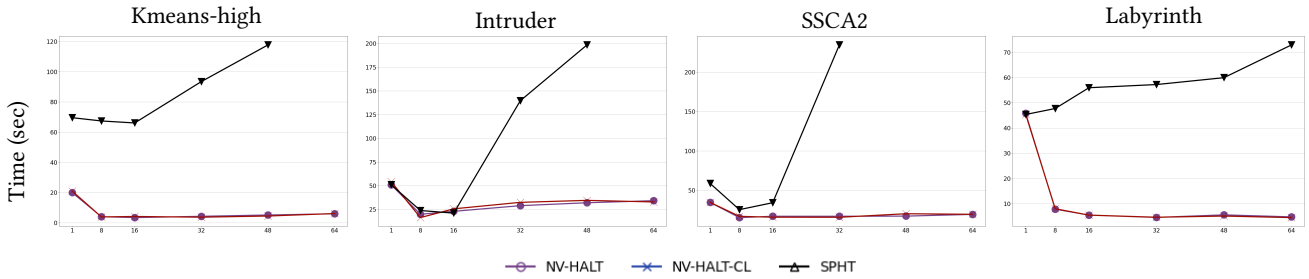

Figure 12: Comparing NV-HALT to SPHT using 4 of the STAMP benchmarks. All benchmarks are using ++ workloads. X-axis is number of threads. Y-axis is time in seconds, lower is better. Missing data points indicate exceeding the 4 minute timeout.

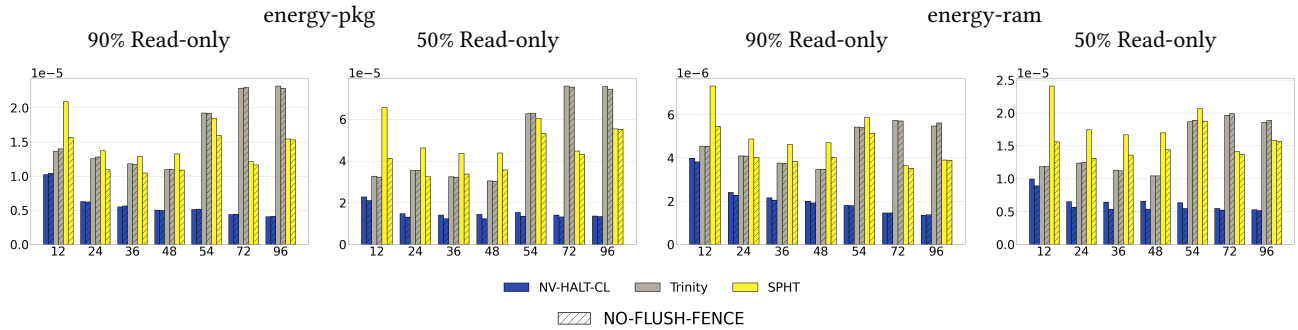

Figure 13: Comparison of energy consumption of persistent TMs with and without flushes/fences via `perf`. First two columns show the energy consumption of the CPU package and the latter two show the energy consumption of RAM. The workload and key access pattern is the same as row 1 of Figure 8. Y-axis is average Joules per commit. X-axis is number of threads.

Figure 12 shows the results of running the STAMP benchmarks. We compare TMs in terms of average processing time (lower is better) for different thread counts. NV-HALT outperforms SPHT in these benchmarks, requiring less processing time in all cases. Both algorithms do not scale well, however, this is not surprising since it is known that the STAMP workloads do not scale well [9, 24]. SPHT experiences negative scaling past 16 threads in all cases. In many of the benchmarks, SPHT exceeds the 4 minute timeout for all trials at some of the higher thread counts. This is primarily due to blocking caused by the log replay mechanism. This is a more accurate representation of the behavior one should expect from SPHT in practice where we cannot delay replaying the log until some arbitrary time in the future and we cannot afford to dedicate hundreds of gigabytes of memory for the log in a single program. These results further reinforce the effectiveness of NV-HALT.

5.3 Power Consumption

One may wonder about the energy consumption of persistent TMs and the energy implications related to forcing write backs from the cache. To address this, we utilized `perf` to measure the energy consumption of NV-HALT, Trinity and SPHT. We specifically track the `perf` events `energy-pkg` and `energy-ram`. These events correspond to the total system-wide energy consumption of the CPU packages (all cores) and RAM respectively. Note that it is not possible to measure the energy consumption of a specific program or core [43, 52]. We measure both the standard implementations as well as the implementations where flushes and fences are replaced with no-ops. The results of these experiments are shown in Figure 13. Note that the thermal design power (TDP) of each processor installed in our machine is 150W. The TDP represents the average power, in watts, the processor dissipates when operating at base frequency with all cores active under an Intel-defined, high-complexity workload [18].

The energy consumption per commit of each algorithm reflects the differences in their throughput. NV-HALT consumes significantly less energy per commit, achieving up to 3.5x improved energy efficiency compared to Trinity and SPHT. SPHT specifically suffers from its use of the WBINVD instruction to flush the entire cache during log replay [11]. In § 5.2.1 we discussed how replacing flushes and fences with no-ops does not significantly effect throughput for SPHT and NV-HALT, especially for workloads dominated by read-only operations. The difference in energy consumption per commit with and without flushes and fences follows this same trend. In most cases replacing flushes and fences with no-ops does not noticeably effect the power consumption per commit of the TMs. SPHT specifically shows a more noticable difference. This also aligns with the results of ablation study shown in Figure 11.

6 Conclusion

In this work we presented NV-HALT, a family of persistent HyTMs utilizing a new technique for persisting hardware path transactions. We implemented several versions of our TM with different liveness guarantees. We demonstrate that, despite the difficulties caused by the inability to persist data within hardware transactions, our persistent HyTM achieves improved performance compared to the existing state of the art.

Acknowledgments

We thank the reviewers for their helpful comments and suggestions. This work was supported by: the Natural Sciences and Engineering Research Council of Canada (NSERC) Collaborative Research and Development grant: CRDPJ 539431-19, the Canada Foundation for Innovation John R. Evans Leaders Fund with equal support from the Ontario Research Fund CFI Leaders Opportunity Fund: 38512, NSERC Discovery Launch Supplement: DGECR-2019-00048, NSERC Discovery Program grant: RGPIN-2019-04227, and the University of Waterloo.

References

[1] Gal Assa, Andreia Correia, Pedro Ramalhete, Valerio Schiavoni, and Pascal Felber. 2023. TL4x: Buffered Durable Transactions on Disk as Fast as in Memory. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. Association for Computing Machinery, New York, NY, United States, 245–259.

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

[3] Trevor Brown. 2017. A template for implementing fast lock-free trees using htm. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*. Association for Computing Machinery, New York, NY, United States, 293–302.

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

[5] Trevor Brown, Aleksandar Prokopec, and Dan Alistarh. 2020. *Non-Blocking Interpolation Search Trees with Doubly-Logarithmic Running Time*. Association for Computing Machinery, New York, NY, USA, 276–291. https://doi.org/10.1145/3332466.3374542

[6] Trevor Brown and Srivatsan Ravi. 2017. Cost of Concurrency in Hybrid Transactional Memory. In *31 International Symposium on Distributed Computing*. Schloss Dagstuhl – Leibniz Center for Informatics, Wadern, Germany.

[7] Daniel Castro, Alexandro Baldassin, João Barreto, and Paolo Romano. 2021. SPHT: Scalable persistent hardware transactions. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, USA, 155–169.

[8] Daniel Castro, Paolo Romano, and João Barreto. 2019. Hardware transactional memory meets memory persistency. *J. Parallel and Distrib. Comput.* 130 (2019), 63–79.

[9] Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick Marlier, et al. 2010. Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. In *Proceedings of the 5th European conference on Computer systems*. Association for Computing Machinery, New York, NY, United States, 27–40.

[10] Félix Cloutier. 2025. CLWB — Cache Line Write Back. https://www.felixcloutier.com/x86/clwb.

[11] Félix Cloutier. 2025. WBINVD — Write Back and Invalidate Cache. https://www.felixcloutier.com/x86/wbinvd.

[12] 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 SIGARCH Computer Architecture News* 39, 1 (2011), 105–118.

[13] Gaetano C Coccimiglio, Trevor A Brown, and Srivatsan Ravi. 2022. PREP-UC: A Practical Replicated Persistent Universal Construction. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*. Association for Computing Machinery, New York, NY, USA, 217–229.

[14] 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), 1–24.

[15] Intel Corporation. 2017. Intel Optane Persistent Memory. Available at: https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.

[16] Intel Corporation. 2018. Persistent Memory Development Kit. https://pmem.io/pmdk/.

[17] Intel Corporation. 2020. Build Persistent Memory Applications with Reliability Availability and Serviceability. https://www.intel.com/content/www/us/en/developer/articles/technical/build-pmem-apps-with-ras.html.

[18] Intel Corporation. 2020. Intel Xeon Gold 5220R Processor Specifications. https://www.intel.com/content/www/us/en/products/sku/199354/intel-xeon-gold-5220r-processor-35-75m-cache-2-20-ghz/specifications.html.

[19] Intel Corporation. 2021. eADR for Persistent Memory. https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html.

[20] Intel Corporation. 2021. Intel Transactional Synchronization Extensions. https://www.intel.com/content/www/us/en/support/articles/000059422/processors.html.

[21] Andreia Correia, Pascal Felber, and Pedro Ramalhete. 2018. Romulus: Efficient algorithms for persistent transactional memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*. Association for Computing Machinery, New York, NY, United States, 271–282.

[22] Andreia Correia, Pascal Felber, and Pedro Ramalhete. 2020. Persistent memory and the rise of universal constructions. In *Proceedings of the Fifteenth European Conference on Computer Systems*. Association for Computing Machinery, New York, NY, United States, 1–15.

[23] Luke Dalessandro, Francois Carouge, Sean White, Yossi Lev, Mark Moir, Michael L Scott, and Michael F Spear. 2011. Hybrid norec: A case study in the effectiveness of best effort hardware transactional memory. *ACM SIGARCH Computer Architecture News* 39, 1 (2011), 39–52.

[24] Luke Dalessandro, Michael F Spear, and Michael L Scott. 2010. NOrec: streamlining STM by abolishing ownership records. *ACM Sigplan Notices* 45, 5 (2010), 67–78.

[25] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. 2006. Hybrid transactional memory. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. Association for Computing Machinery, New York, NY, United States, 336–346.

[26] Tudor David, Aleksandar Dragojević, Rachid Guerraoui, and Igor Zablotchi. 2018. Log-Free Concurrent Data Structures. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, USA, 373–385.

[27] Dave Dice, Ori Shalev, and Nir Shavit. 2006. Transactional locking II. In *International Symposium on Distributed Computing*. Springer, Springer-Verlag, Berlin, Heidelberg, 194–208.

[28] Robert Ennals and Robert Ennals. 2006. *Software transactional memory should not be obstruction-free*. Technical Report. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report.

[29] Pascal Felber, Christof Fetzer, and Torvald Riegel. 2008. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. Association for Computing Machinery, New York, NY, United States, 237–246.

[30] Keir Fraser and Tim Harris. 2007. Concurrent programming without locks. *ACM Transactions on Computer Systems (TOCS)* 25, 2 (2007), 5–es.

[31] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. 2020. NVTraverse: In NVRAM Data Structures, the Destination is More Important than the Journey. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 377–392. https://doi.org/10.1145/3385412.3386031

[32] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. 2018. A Persistent Lock-Free Queue for Non-Volatile Memory. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Vienna, Austria) *(PPoPP '18)*. Association for Computing Machinery, New York, NY, USA, 28–40. https://doi.org/10.1145/3178487.3178490

[33] Michal Friedman, Erez Petrank, and Pedro Ramalhete. 2021. *Mirror: Making Lock-Free Data Structures Persistent*. Association for Computing Machinery, New York, NY, USA, 1218–1232. https://doi.org/10.1145/3453483.3454105

[34] Kaan Genç, Michael D Bond, and Guoqing Harry Xu. 2020. Crafty: Efficient, HTM-compatible persistent transactions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, USA, 59–74.

[35] Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. Pisces: A scalable and efficient persistent transactional memory. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, United States, 913–928.

[36] Rachid Guerraoui and Michał Kapałka. 2010. *Principles of transactional memory*. Vol. 4. Morgan & Claypool Publishers, California USA.

[37] Tim Harris, James Larus, and Ravi Rajwar. 2022. *Transactional memory*. Springer Nature, Berlin, Heidelberg.

[38] 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*. Association for Computing Machinery, New York, NY, United States, 355–364.

[39] Maurice Herlihy and J Eliot B Moss. 1993. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on Computer architecture*. Association for Computing Machinery, New York, NY, USA, 289–300.

[40] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg, 313–327. https://doi.org/10.1007/978-3-662-53426-7_23

[41] Jungi Jeong, Jaewan Hong, Seungryoul Maeng, Changhee Jung, and Youngjin Kwon. 2020. Unbounded hardware transactional memory for a hybrid DRAM/NVM memory system. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, IEEE, New Jersey, United States, 525–538.

[42] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. 2018. Dhtm: Durable hardware transactional memory. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, IEEE, New Jersey, United States, 452–465.

[43] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K Nurminen, and Zhonghong Ou. 2018. Rapl in action: Experiences in using rapl for power measurements. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)* 3, 2 (2018), 1–26.

[44] Nicolas Krauter, Patrick Raaf, Peter Braam, Reza Salkhordeh, Sebastian Erdweg, and Andre Brinkmann. 2021. Persistent software transactional memory in Haskell. *Proceedings of the ACM on Programming Languages* 5, ICFP (2021), 1–29.

[45] Sanjeev Kumar, Michael Chu, Christopher J Hughes, Partha Kundu, and Anthony Nguyen. 2006. Hybrid transactional memory. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. Association for Computing Machinery, New York, NY, United States, 209–220.

[46] Hung Q Le, Guy L Guthrie, Dan E Williams, Maged M Michael, Brad G Frey, William J Starke, Cathy May, Rei Odaira, and Takuya Nakaike. 2015. Transactional memory support in the IBM POWER8 processor. *IBM Journal of Research and Development* 59, 1 (2015), 8–1.

[47] Daan Leijen, Benjamin Zorn, and Leonardo de Moura. 2019. Mimalloc: Free list sharding in action. In *Programming Languages and Systems: 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1–4, 2019, Proceedings 17*. Springer, Springer, Berlin, Heidelberg, 244–265.

[48] Yossi Lev, Victor Luchangco, Virendra Marathe, Mark Moir, Dan Nussbaum, and Marek Olszewski. 2009. Anatomy of a scalable software transactional memory. In *Proc. 4th ACM SIGPLAN Workshop on Transactional Computing*. Association for Computing Machinery, New York, NY, United States.

[49] Alexander Matveev and Nir Shavit. 2015. Reduced hardware norec: A safe and scalable hybrid transactional memory. *ACM SIGPLAN Notices* 50, 4 (2015), 59–71.

[50] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. 2008. STAMP: Stanford transactional applications for multi-processing. In *2008 IEEE International Symposium on Workload Characterization*. IEEE, IEEE, New Jersey, United States, 35–46.

[51] Dan Nussbaum, Yosef Lev, and Mark Moir. 2007. PhTM: Phased transactional memory. In *Second ACM SIGPLAN Workshop on Transactional Computing*.

[52] Guillaume Raffin and Denis Trystram. 2025. Dissecting the software-based measurement of CPU energy consumption: a comparative analysis. *IEEE Transactions on Parallel and Distributed Systems* 36 (2025), 96–107.

[53] Ravi Rajwar and James R Goodman. 2001. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*. IEEE, IEEE, New Jersey, United States, 294–305.

[54] Pedro Ramalhete and Andreia Correia. 2024. Scaling Up Transactions with Slower Clocks. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. Association for Computing Machinery, New York, NY, United States, 2–16.

[55] Pedro Ramalhete, Andreia Correia, and Pascal Felber. 2021. Efficient algorithms for persistent transactional memory. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Association for Computing Machinery, New York, NY, United States, 1–15.

[56] Torvald Riegel, Patrick Marlier, Martin Nowack, Pascal Felber, and Christof Fetzer. 2011. Optimizing hybrid transactional memory: The importance of nonspeculative operations. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*. Association for Computing Machinery, New York, NY, United States, 53–64.

[57] Nir Shavit and Dan Touitou. 1995. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*. Association for Computing Machinery, New York, NY, USA, 204–213.

[58] Haris Volos, Andres Jaan Tack, and Michael M Swift. 2011. Mnemosyne: Lightweight persistent memory. *ACM SIGARCH Computer Architecture News* 39, 1 (2011), 91–104.

[59] Yuanhao Wei, Naama Ben-David, Michal Friedman, Guy E. Blelloch, and Erez Petrank. 2022. FliT: A Library for Simple and Efficient Persistent Algorithms. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 309–321. https://doi.org/10.1145/3503221.3508436

[60] Chencheng Ye, Yuanchao Xu, Xipeng Shen, Yan Sha, Xiaofei Liao, Hai Jin, and Yan Solihin. 2023. Specpmt: Speculative logging for resolving crash consistency overhead of persistent memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. Association for Computing Machinery, New York, NY, United States, 762–777.

[61] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. 2019. Efficient lock-free durable sets. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–26.