Durable Hardware Transactional Memory for Extended Asynchronous DRAM Refresh Architectures

João Miguel Santos Falcão Moreno Pinheiro

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisors: Prof. João Pedro Faria Mendonça Barreto
Prof. Paolo Romano

Examination Committee

Chairperson: Prof. Mário Jorge Costa Gaspar da Silva
Supervisor: Prof. João Pedro Faria Mendonça Barreto
Member of the Committee: Prof. João M. S. Lourenço

June 2022

Acknowledgments

I would like to thank my parents for their patience, encouragement, and support over all these years and without whom this project would not be possible.

I would also like to thank my dissertation supervisors Prof. João Pedro Barreto and Prof. Paolo Romano as well as Daniel Castro and Prof. Alexandro Baldassin for their insight, time, and dedication that has made this Thesis possible.

And last but not least, my friends and colleagues who helped me grow as a person and were always there for me.

To each and every one of you – Thank you.

Abstract

Byte-addressable Persistent Memory (PM) technologies present a new paradigm for interacting with non-volatile memory, allowing applications to access PM directly and with much lower latency than before. Unfortunately, the combination of PM with Hardware Transactional Memory (HTM) has been far from trivial to implement due to the volatile nature of CPU caches, requiring the use of software instrumentation and techniques like Shadow Paging (SP) to guarantee durable HTM to PM. The commercial release of Intel Optane DC PM and the support for systems with Enhanced Asynchronous DRAM Refresh allows for CPU caches to be considered persistent as well, greatly simplifying the model for durable HTM. However, the use of software instrumentation techniques like Shadow Paging can still provide significant benefits for durable HTM solutions by taking advantage of the higher performance and lower latency of DRAM. This dissertation presents and evaluates a new solution based on the use of DRAM shadow paging for architectures with PM and durable CPU caches in order to improve performance and throughput.

Keywords

Persistent Memory; Transactional Memory; Shadow Paging; Extended Asynchronous DRAM Refresh; Hardware Transactional Memory

Resumo

As tecnologias de Memória Persistente Endereçável por Byte (PM) apresentam um novo paradigma para interagir com memória não volátil, permitindo que aplicações tenham acesso directo a PM e com latência muito inferior do que anteriormente. Infelizmente, a combinação de PM com Memória Transacional de Hardware (HTM) está longe de ser trivial de implementar devido à natureza volátil das caches de CPU, exigindo o uso de técnicas de instrumentação de software como Shadow Paging (SP) para garantir HTM durável para PM. A disponíbilidade comercial de Intel Optane DC PM e o suporte para sistemas com Atualização de DRAM Assíncrona Melhorada (eADR) permitem que as caches do CPU também sejam consideradas persistentes, simplificando bastante o modelo para HTM durável. No entanto, o uso de técnicas de instrumentação de software como Shadow Paging ainda podem fornecer benefícios significativos para soluções HTM duráveis, aproveitando o maior desempenho e a menor latência da DRAM. Esta dissertação apresenta e avalia uma nova solução baseada no uso de Shadow Paging em DRAM para arquiteturas com PM e caches de CPU duráveis com o objetivo de melhorar o desempenho.

Palavras Chave

Memória Persistente; Memória Transacional; Shadow Paging; Extended Asynchronous DRAM Refresh; Memória Transacional de Hardware

Contents

List of Figures

List of Algorithms

1

Introduction

Contents

1.1 Introduction

1.1.1 Motivation

Persistent storage options for computing systems have traditionally been limited to mass storage devices such as hard disk drives (HDD) and solid-state drives (SSD), whose performance is orders of magnitude slower than volatile main memory (DRAM). These mass storage devices are not directly accessible to applications and instead require the use of Application Programming Interfaces (API) provided by the Operating System (OS), along with costly serialization and deserialization processes to translate the data between its volatile representation and a format that can be stored on non-volatile devices.

The emergence of new byte-addressable Persistent Memory (PM) technologies like Intel Optane DC Persistent Memory opened the door to a new paradigm for interacting with non-volatile memory. These new PM technologies offer performance that is closer to DRAM and can be connected to the processor's memory bus.

This allows applications to address PM directly, without the need for OS-level API calls or serialization and deserialization steps. Additionally, these new PM technologies have lower energy consumption and provide higher capacity than DRAM at a lower cost.

Given that multicore processors are the standard architecture for current computing systems, it is imperative to consider how new technologies fit in a concurrent computing environment. However, developing parallel applications is not an easy process, and abstractions like transactional memory are an important area of research for taking advantage of multiple cores while reducing the complexity of parallel application development. Hardware Transactional Memory (HTM) implementations are particularly desirable due to their focus on minimizing the overhead of instrumentation.

The development of concurrent applications that can take advantage of these new PM technologies has led to significant attention in research into the implementation of Persistent Transactional Memory in systems equipped with PM and HTM.

However, HTM's reliance on volatile CPU caches means that committed transactions cannot be guaranteed to be atomically persisted to PM due to the possibility of remaining in the cache. It is thus required to complement transactions with complex software instrumentation in order to ensure durable HTM.

This approach of combining transactions with additional software instrumentation in order to guarantee atomicity and durability has been successfully utilized by state-of-the-art approaches such as NV-HTM [1], DudeTM [2], cc-HTM [3], Crafty [4], and SPHT [5].

One technique that is particularly notable is the use of Shadow Paging (SP) [1, 2, 5] in combination with Write-Ahead Logging (WAL), which is used in all these solutions, with the exception of Crafty. With this technique updates are performed on local private copies rather than directly over the original data, allowing easier modification without the issue of consistency constraints. The original data is then replaced by the shadow copy, making the updates durable.

More recently, the introduction of new persistence domains for computing systems with small amounts of reserve power, such as Intel Extended Asynchronous DRAM Refresh (eADR), offered the possibility of treating CPU caches as non-volatile. In this new environment, HTM transactions that deal exclusively with PM data are able to rely entirely on hardware-level instructions without the need for any additional instrumentation in order to ensure durability.

However, despite offering performance that is significantly faster than persistent mass storage devices, current PM modules still have higher latency and slower write speeds than DRAM. It can thus be desirable to use an algorithm that makes use of DRAM in order to increase performance and reduce latency while taking advantage of HTM and the new eADR persistence domain for persistence [6].

Shadow Paging is a possible solution for achieving this goal. However, existing durable HTM solutions based on Shadow Paging have been designed for an ADR environment and are thus more computationally expensive than necessary in eADR. These solutions include mechanisms which are no longer required and do not take into consideration particular idiosyncracies of eADR environments, such as the possibility of flush operations being used to improve the performance of write operations.

1.1.2 Contributions

This work revisits proposals for durable HTM based on DRAM shadow paging for architectures with volatile caches and, based on those state-of-the-art proposals (SPHT [5], NV-HTM [1], DudeTM [2], etc), introduces a new solution optimized for systems with durable caches, such as eADR-based architectures.

This work revisits proposals for durable HTM based on DRAM shadow paging for architectures with volatile caches and presents four main contributions.

As a first contribution, it includes a preliminary study which shows that, counter to common intuition, the use of shadow copying in an eADR environment can bring performance benefits when compared to the use of HTM directly on persistent memory.

As a second contribution, it introduces SPHT-eADR, a new solution based on SPHT that has been designed for systems with durable caches and reconsiders several mechanisms that are no longer necessary in an eADR environment.

As a third contribution, it considers the results of recent studies in eADR [7] and explores an optimization of SPHT-eADR that improves the performance of write operations in eADR by explicitly flushing log data with low temporal locality.

As a fourth and final contribution, it presents an experimental evaluation of SPHT-eADR compared to the standard version of SPHT and an almost pure HTM approach, using STAMP [8], and TPC-C [9], and synthetic no-contention benchmarks. This experimental evaluation shows that it significantly improves on the performance of previous state-of-the-art solutions, reaching 2–3x the throughput performance of SPHT and 4x the throughput of HTM+SGL in synthetic (no contention) benchmarks and 1.5x the throughput of SPHT and 2x the throughput of HTM+SGL at similar thread counts on TPC-C.

1.2 Organization of the Document

This thesis is is organized as follows: Chapter 2 discusses topics that form the background and related work, covering the subjects of Persistent Memory, Transactional Memory, techniques for Combining Transactional Memory with Persistent Memory, and state-of-the-art Durable HTM Solutions. Chapter 3 presents the design of SPHT-eADR. Chapter 4 discusses the benchmarks used and presents the evaluation of SPHT-eADR, comparing it with previous state-of-the-art approaches. And finally, Chapter 5 summarizes conclusions for this work and discusses potential future directions that can be explored.

2

Background and Related Work

Contents

This section presents an overview of the related work in the areas of Durable Hardware Transactional Memory. Section 2.1 begins with Persistent Memory (PM) and discusses the topics of PM technologies, Memory Hierarchies and Persistence Domains. Section 2.2 presents Transactional Memory (TM) in a volatile environment and covers Software Transactional Memory (STM), Hardware Transactional Memory (HTM), and Hybrid Transactional Memory (HyTM) approaches. Section 2.3 combines the topics of PM and TM and covers Write-Ahead Logging (WAL) and Shadow Paging (SP) techniques, along with the changes introduced by an eADR Environment. Finally, Section 2.4 presents SPHT and NV-HTM, two state-of-the-art solutions for Durable HTM.

2.1 Persistent Memory

The advent of byte-addressable PM technologies has the potential to revolutionize the way in which data-intensive applications are developed. When compared to DRAM, these new technologies have significantly higher storage density, lower power consumption, and the ability to retain their contents for extended periods of time in the absence of power. Additionally, unlike traditional NVM mass storage devices, these new technologies are byte-addressable and connect directly to the computer's memory bus, allowing applications to address them directly without the need for translation steps to serialize and deserialize between representations or to go through any Operating System APIs.

However, despite the groundbreaking new features, these new persistent memory technologies also have notable drawbacks. Latency times for write operations are significantly higher than for read operations, which may cause serious performance degradation in write-intensive applications. There is finite write endurance, meaning that there is a limited number of times each bit may be written before failure. And finally, bandwidth and performance are still limited compared to DRAM.

2.1.1 Intel Optane Persistent Memory

Intel Optane is the first, and currently the only, PM technology commercially available on the market. Optane DC memory is available in two different formats: as an NVMe SSD storage module that connects to the PCIe bus just like all other NVM mass storage devices, and in a DIMM format, which uses the physical DDR4 packaging and memory bus. It is this latter format, Optane DC, that is of most interest for PM applications, allowing it to be byte-addressable and much closer to DRAM in terms of latency. Optane DC PM offers a latency of up to about 350 ns, several orders of magnitude lower than the typical 10,000–100,000 ns latency of NAND-based SSD mass storage devices, but still higher than the 10–20 ns latency range for DDR4 DRAM [10].

However, despite the dramatic improvement in terms of latency, the bandwidth performance of Intel Optane, especially in terms of write operations, is still significantly worse than DRAM and closer to SSD mass storage devices [11].

Intel advertises a write endurance of 60 drive writes per day for an average lifetime of 5 years. While this level of endurance is significantly higher than the write endurance of NAND-based SSD mass storage devices, it can still become problematic when used as main memory in write-intensive applications [11].

2.1.2 Memory Hierarchy

Introducing a new type of memory that combines the relatively fast speed and byte-addressability of main memory with the larger capacities and non-volatile nature of secondary memory allows for different configurations of the memory hierarchy. Intel Optane DC PM is capable of operating in two distinct modes: Memory Mode and App Direct Mode.

In Memory Mode, DRAM operates as an L4 cache layer between Optane DC and the CPU caches, being exposed to the application as a single pool of addressable memory. This configuration is transparent to the application and offers Optane DC's larger memory capacity while taking advantage of DRAM's performance in order to hide Optane DC's higher write latency. However, data in Memory Mode is considered volatile and does not take advantage of Optane DC's persistent memory properties.

In App Direct Mode, DRAM and Optane DC PM are exposed to the application as two distinct addressable pools of memory: a persistent pool for Optane DC and a volatile pool for DRAM. Operating in this mode requires applications to be programmed specifically with these independent pools in mind, but allows for applications to take advantage of whichever medium is ideal for the task.

2.1.3 Persistence Domains

One of the ideas that are central to computing systems with PM devices is the concept of a Persistence Domain (PD); the definition of the region of a computing system that is able to guarantee persistence. Once data reaches the PD, it can be guaranteed to have been persisted and recoverable upon system restart. The PD is not just limited to PM devices though; it may also include volatile devices that are able to hold state for long enough to be able so that it can be guaranteed to reach a persistent device. Intel Optane DC PM supports two different persistence domains, Asynchronous DRAM Refresh (ADR) and Enhanced Asynchronous DRAM Refresh (eADR).

In an ADR environment, the system has enough reserve energy to flush the memory controller's Write Pending Queue (WPQ) to PM. This means that it is sufficient for the application data to reach the memory controller in order to guarantee that it can be considered persistent under ADR.

In an eADR domain, computing systems have a higher amount of reserve energy than in ADR which, in addition to providing enough power to flush the memory controller's WPQ, also have enough power to allow the system to execute the required instructions to guarantee that CPU caches are also flushed to PM. The inclusion of the CPU caches in the eADR persistence domain brings significant advantages over ADR to the PM programming model. In an ADR environment data becomes visible to other cores via the CPU L3 cache layer before it has a chance to reach the memory controller and become persistent. It is also necessary for applications to explicitly flush caches in order for a store to become persistent. In an eADR environment, it is no longer necessary for applications to flush the caches in order to ensure persistence, and data becomes persistent before it becomes visible to other cores through the L3 cache layer. This offers the opportunity to simplify the programming model for durable HTM transactions by allowing durable HTM transactions to be executed entirely without the need for software instrumentation since values on the CPU cache can now be considered persistent.

2.2 Transactional Memory

The concept of a transaction was initially developed for database systems that needed to perform a set of operations that could manipulate data atomically in order to allow concurrent access to the data without sacrificing consistency. Database transactions guarantee this by ensuring 4 essential properties: atomicity, consistency, isolation, and durability (ACID) [12].

These properties are also essential in the context of parallel and concurrent programming where critical sections of code need to be executed sequentially and in isolation. This can be ensured through the use of locks, but lock-based programming is difficult to tune and notoriously known for being prone to programming errors [13, 14]. Global or coarse-grained locks are easier to implement, but can seriously degrade performance and restrict parallelism. Fine-grained locking allows for better parallelism and performance but is complex to implement and prone to errors that can be difficult to identify.

Transactional Memory takes advantage of the transaction abstraction from database systems in order to provide a much easier and safer programming model for parallel and concurrent computing, where the programmer is only required to identify and annotate the critical section of code as a transaction, making it much easier to focus on the application logic rather than how to coordinate lock-based synchronization mechanism. A transaction can only have one of two possible outcomes: either the transaction is committed and its changes are atomically applied, or the transaction is aborted and its changes are discarded.

The use of Transactional Memory allows for critical sections to be run speculatively and only committed if no conflicts are detected. If a conflict is detected, the transaction is aborted and no changes are applied.

One way to define the set of guarantees provided by Transactional Memory is through the correctness criterion of Opacity [15, 16]. The concept of Opacity is strongly related to Serializability, which is the strongest correctness criterion that is typically applied in the context of database systems. Opacity is a stronger criterion and can be thought of as an extension of Serializability with the enforcement of two additional requirements:

- Even aborted transactions are not allowed to access an inconsistent state.
- Committed transactions have to be serialized in the real-time order of the execution of the transactions.

2.2.1 Software Transactional Memory

The Transactional Memory abstraction can be achieved through a purely software-based runtime approach, without the need for support from the underlying hardware [17, 18]. Using a Software Transactional Memory (STM) implementation allows for code portability across a wide range of hardware. STM is implemented by instrumenting the read and write operations performed in the context of a transaction in order to track changes and identify potential conflicts between transactions.

While the portability and hardware independence is an attractive advantage, STM implementations typically suffer from poor performance due to the high instrumentation costs and overheads.

2.2.2 Hardware Transactional Memory

Unlike with STM, Transactional Memory implemented at the hardware level does not require the instrumentation of read and write operations. Instead, it relies on the cache coherence protocol to achieve atomicity and isolation without suffering from the high costs and overheads of instrumentation. This allows for better performance but also means that currently available HTM implementations are considered best-effort [19, 20]. Best-effort HTM is limited by cache capacity and cannot handle transactions that are too large to fit in the private CPU cache.

There are proposals for Unbounded HTM which overcome the size limitations of best-effort HTM, but their implementations introduce significant additional complexity at the hardware level [21–23].

2.2.3 Hybrid Transactional Memory

In order to overcome the limitations of STM and HTM, there have been proposals for a Hybrid Transactional Memory (HyTM) approach [24]. HyTM solutions attempt to take advantage of best-effort HTM whenever possible but provide an STM fallback solution in order to guarantee progress whenever the HTM-based transaction aborts or is unable to handle the transaction.

2.3 Combining Transactional Memory with Persistent Memory

The use of byte-addressable persistent memory technology enables applications to access and modify durable state without the need for an intermediate layer like the OS filesystem API, but still requires developers to ensure that modifications to the data are consistently applied to PM in the event of a failure. This is especially true when trying to guarantee the failure-atomicity of multiple operations that should either all be persisted together, or none should be persisted at all [25].

There are multiple durability abstractions for programming with PM, such as Persistent Data Structures [26–29] and Failure Atomic Sections (FASE) [30] with Epoch [31–35], Lock [30,36–38], and Transaction [39–41] based approaches. The scope of this work is related to transactions and will therefore focus primarily on describing that programming model.

Similarly to TM in a volatile computing environment, early literature and solutions combining TM with PM focused on software-based approaches. One particularly important early contribution in the topic was Mnemosyne [40], which borrowed ideas such as write-ahead redo logging from database systems [12] to provide a lightweight persistent transactional memory framework.

As hardware support for transactional memory started becoming available in commercial computing systems, some hardware-based solutions were also published, although early proposals required changes to the existing hardware. In order to bridge the gap between the properties offered by HTM and the properties required to guarantee durability with persistent memory, durable HTM solutions took advantage of hardware-based transactions and complemented them with software instrumentation. The majority of systems employed the use of two techniques from the world of database systems: write-ahead logging and shadow paging. NV-HTM [1], described in more detail below, along with DudeTM [2] and cc-HTM [3] make use of both of these techniques.

2.3.1 Write-Ahead Logging

Typically used by database systems to provide failure atomicity and durability. This technique requires updates to PM to be recorded and persisted in a log before actually being written to their memory locations in PM. There are two main logging schemes in use in current PM systems: undo logging and redo logging. Undo logging approaches save the old version of the data in the log before changing the data in persistent memory, which allows the system to roll back changes in case of a failure. Redo logging, on the other hand, records the new version of the data in the log, before writing it directly to PM. In case of a failure, it is possible to replay the log and rebuild the state of the system.

Current commodity HTM systems do not allow flushing cached logs to PM within the context of a transaction, which prevents the use of write-ahead logging techniques.

2.3.2 Shadow Paging

The Write-Ahead Logging approach described earlier performs in-place updates, meaning that all changes are performed directly on PM, overwriting the previous value. Shadow Paging allows for updates to be performed out-of-place. Instead of performing the update directly on the original object, it first creates a private copy of the object so that persistent updates can be applied to it without disturbing or modifying the original object. Since these private objects are local, they can be modified without worrying about the order of persist instructions. When a transaction is about to commit, the original objects are then replaced with the updated copies.

This approach presents two significant advantages in the context of PM. Private copies are stored in DRAM and are able to take advantage of its lower latency, allowing for better write performance, especially in the case of hot objects that are overwritten multiple times. A second advantage is that it can help improve the write endurance of PM systems by absorbing repeated overwrites in DRAM before issuing a single write operation to PM.

2.3.3 Early Studies of Plain HTM on eADR Systems

Even though the previously described approaches of Shadow Paging and Write-Ahead Logging are no longer necessary to guarantee durable HTM transactions in an eADR environment, they can still be used to mitigate some of the limitations of current PM technologies, such as limited write endurance and the significantly higher latency for write operations [10].

Pantea Zardoshti et al. [6] studied the performance of Intel Optane DC PM in ADR and eADR environments using a variety of different Persistent Transactional Memory algorithms. They also evaluated the same algorithms running in DRAM (not persistent) in order to compare the performance difference to Optane. They concluded that using Optane in an eADR environment provided substantial performance gains over an ADR environment in almost all tested workloads. The single exception was a workload with a significant amount of work between transactions, which resulted in only a small fraction of the execution-time being transaction-related.

However, despite the performance advantage of running in an eADR environment, the authors found that it still does not reach the performance of DRAM, and that scalability on Optane is worse than on DRAM.

2.4 Durable HTM Solutions

In this section I illustrate an approach to durable HTM through the use of Shadow Paging by presenting two complete solutions that are representative of the state-of-the-art in this area.

First I present NV-HTM [1], which combines Shadow Paging with Write-Ahead Logging and illustrates an approach used by a larger group of solutions, like DudeTM [2] and cc-HTM [3]. Next I present SPHT [5], which improves on the work done in the previous mentioned solutions and represents the state-of-the-art in durable HTM via Shadow Paging.

2.4.1 NV-HTM

NV-HTM, proposed by Daniel Castro et al. [1], combines unmodified commodity HTM with PM in order to provide durable HTM. It achieves this through the use of an additional redo log saved in persistent memory that tracks all the write operations issued by transactions. In order to be able to provide crash-atomicity, it ensures all transactional updates are persisted in the log before any values in persistent memory can be modified.

However, current commodity HTM implementations are not able to flush the cached log of a transaction to persistent memory before committing the transaction. In order to overcome this, NV-HTM makes use of a hardware-software co-design by instrumenting write operations issued through HTM transactions and tracking them in a persistent redo log before committing. Once committed in hardware, the NV-HTM transaction is able to make its updates and logs visible to other threads, but not necessarily persisted in PM. At this point, the transaction is considered non-durably committed.

In order for a thread to durably commit an HTM transaction, it needs to postpone the commit event until the log of that transaction, along with the logs of any other transactions it may depend on, have been persisted to PM. It is at this point that the transaction can be considered durably committed.

This ensures that when a transaction's commit is made visible, all of its log entries have already been persisted and can thus be replayed. However, the application's state persisted in memory may not yet reflect all of the durably and non-durably committed transactions. In case of failure and recovery, the key insight behind NV-HTM is to discard the application state and reconstruct it by replaying the logs of all durably committed transactions on top of a consistent checkpoint.

NV-HTM utilizes a decentralized design where each thread maintains its own local log that only stores information related to transactions that it executed. It also builds a non-durable log during the execution of a transaction that only gets persisted after the non-durable HTM commit event. This log is marked as persistent through a commit marker added via software, which turns it into a durable commit. This design overcomes the issues of the log becoming a contention spot in highly concurrent systems and also overcomes the limitation of commodity HTM systems not allowing the log to be flushed during the execution of the transaction.

In order to reduce the duration of the recovery process and the unbounded growth of the redo log, NV-HTM introduces a checkpointing process called Backward Filtering Checkpointing to create consistent snapshots persisted to PM from which the application is able to recover. Backward Filtering

Checkpointing persists all the updates of durably committed transactions to the consistent persistent snapshot and is designed in order to filter repeated flushes to cache lines or writes to the same memory location by different transactions in the logs. It only needs to persist the most recent update and is able to discard all previous updates to the same location. This improves not only performance but also write endurance by minimizing the number of write operations made to each location in persistent memory.

In case of a failure, NV-HTM is able to recover its state by replaying the durably committed logs on top of the consistent persistent snapshot.

At a high level, the architecture of NV-HTM is comprised of:

- A working process that runs a set of parallel worker threads which execute hardware transactions on the working snapshot. The working snapshot is a private memory pool that is initially created using copy-on-write to transparently create a volatile copy of the PM page. It starts by mapping pages that are stored entirely in PM but will evolve to contain a mix of clean pages in PM and dirty page copies in DRAM.
- A durable log which is stored in PM and is used to track updates from durably committed transactions. Each thread maintains its own private log in order to avoid contention.
- A checkpointing process that applies the updates stored in the logs with the goal of building a consistent persistent snapshot that reflects all the durably committed transactions. This checkpointing process also prunes the logs in order to ensure the log size never exceeds a predefined maximum size. The checkpointing process makes use of Backward Filtering Checkpointing.

Algorithm 1 presents the pseudo-code for the behaviour of an NV-HTM worker thread in an environment with volatile caches, as proposed by Daniel Castro et al. [1]

Algorithm 1 NV-HTM: transaction processing at thread t

```
    Shared variables
 1: $log[N]$                                        ▷ One log per thread, stored in PM
 2: $ts[N] \leftarrow \{+\infty, \ldots, +\infty\}$
                       ▷ Per-thread timestamp of active tx; $+\infty$ if none is active
    Thread local variables
 3: $locTs$                                         ▷ timestamp of committing transactions
 4: $isRO$                                          ▷ flag used to identify read-only tx
    Functions
 5: function BEGIN
 6:     $isRO \leftarrow$TRUE
 7:     $ts[t] \leftarrow$READTS()
 8:     mem_fence                                   ▷ Ensure other threads know we are in a tx
 9:     htm_begin()                                 ▷ Start hw tx
10: function WRITE(addr, value)
11:     $isRO \leftarrow$FALSE
12:     if logCheckSpace ($log[t]$)=FULL then
13:         ABORT(LOG_FULL)
14:     $*addr \leftarrow value$                    ▷ Write to working snapshot
15:     $log[t]$.append($< addr, value >$)
16: function ABORT(abort_code)
17:     htm_abort($abort\_code$)
18:     $ts[t] \leftarrow +\infty$
19: function COMMIT
20:     if isRo then                                ▷ Commit logic for read-only txs
21:         htm_commit()
22:         $ts[t] \leftarrow +\infty$              ▷ Others do not need to wait for RO tx
23:         WAITCOMMIT()
24:     else                                        ▷ Commit logic for update txs
25:         $locTs \leftarrow$ READTS()
26:         htm_commit()
27:         $ts[t] \leftarrow locTs$
28:         logFlush($log[t]$)                      ▷ Flush current log entries
29:         WAITCOMMIT()
30:         $log[t]$.append($<$ COMMIT$, locTS >$)
31:         $log[t].endP \leftarrow locEndP$
32:         logFlush($log[t]$)                      ▷ Flush commit marker and $endP$
33:         $ts[t] \leftarrow +\infty$
34: function WAITCOMMIT
35:     for all $t^* \in [1, N]$ $s.t.$ $t^* \neq t$ do
36:         wait until $ts[t^*] > ts[t]$
```

2.4.2 SPHT

SPHT, proposed by Daniel Castro et al. [5], builds on the concepts utilized in NV-HTM [1] and introduces novel mechanisms aimed at improving scalability during transaction processing and recovery. It addresses scalability challenges connected with redo logging by introducing a new highly scalable commit protocol that amortizes the cost of ensuring immediate durability [42] across multiple concurrent transactions.

The architecture for SPHT is comprised of 2 main processes:

- The transaction executer, which is comparable to the working process from NV-HTM. It spawns multiple worker threads responsible for executing the transactions and creates a shadow copy of the persistent heap shared by all threads and serves as a working snapshot that transactions access directly. Updates performed by HTM transactions on the working snapshot are not immediately written to the persistent heap and are thus still volatile.

- The log replayer, which spawns the replayer threads responsible for replaying the durably committed logs and updating the persistent heap.

Similarly to NV-HTM, each worker thread has its own private durable redo log used to track updates performed by each transaction. Since the results of a transaction can remain in the cache, the redo log needs to be explicitly flushed to persistent memory after the HTM commit. Once the redo log has been persisted, a timestamped commit marker is used to mark the transaction as durable.

SPHT takes advantage of the observation that at a high thread count, multiple transactions are likely to be concurrently attempting to commit. It takes advantage of this by ensuring the immediate durability of all transactions that are trying to commit through a single update of the persistent global marker with the timestamp of the most recent durable transaction. Just like NV-HTM, SPHT uses physical clocks to establish the order of transactions.

After an HTM commit, SPHT allows threads to flush their logs out of order, without considering thread synchronization. However, those logs cannot be marked as durable yet since they may depend on logs from other threads that may not yet have been flushed. It overcomes this by ensuring that each thread transaction waits for all threads with earlier timestamps to finalize persisting their logs. During this waiting phase, it also determines which transaction in the commit phase has the highest timestamp. If there is a transaction with a higher timestamp, the current transaction avoids updating the global marker. Only the transaction with the highest timestamp updates the global marker, which reduces the number of updates and flushes to the marker. This marks all transactions with earlier timestamps as durable.

SPHT also improves the scalability of log replay by employing 2 novel ideas, a log-linking mechanism that spares the replayer threads from the cost of having to determine which transaction should be replayed next, and parallelization of the log replay process in a Non-Uniform Memory Access (NUMA) aware fashion.

SPHT also improves the log replay process with the introduction of two new mechanisms:

- Linked transactions in the logs through the addition of an entry that stores a pointer to the beginning of the next transaction in the replay order and spares the replayer threads from the cost of having to determine which transaction should be replayed next. This ordering can also be done through backward linking, which allows for filtering techniques that reduce the number of writes by absorbing repeated writes to the same address and writing only the most recent value for each address.
- NUMA-Aware Parallel log replay, in which replayer threads target disjoints regions of memory, ensuring that the log can be processed in parallel without violating the sequential order established in the logs.

3

SPHT-eADR

Contents

The study of the state of the art presented in Chapter 2 shows that ensuring persistence in an ADR environment requires complex software instrumentation in order to guarantee that values do not remain lingering in volatile cache. The introduction of the new eADR domain in which caches can be considered durable means that it is now possible to implement durable HTM without the need for complex software instrumentation in order to identify which cache lines have been modified and guarantee they have been persisted. However, even though it is now possible to guarantee persistence entirely through HTM without the need for any additional software instrumentation, PM modules still suffer from finite write endurance and higher access latencies, particularly regarding write operations. Thus, it can still be beneficial to use some of these software techniques, such as shadow paging, in order to improve performance by taking advantage of the lower latency and higher performance of DRAM.

This section presents an analysis of how durable HTM applications can be developed in an eADR environment and describes the work that was done to develop a new optimized solution that makes use of software instrumentation techniques. Section 3.1 presents a study of how current off-the-shelf approaches can be used to develop applications that make use of eADR. Section 3.2 presents some of the fundamental mechanisms of SPHT that were required for an ADR environment, and Section 3.3 describes how these mechanisms can be simplified or modified to take advantage of eADR. Finally, Section 3.4 describes the implementation for SPHT-eADR.

3.1 Study of Current Off-the-Shelf Approaches

The introduction of eADR leaves open the question of how to develop applications that are able to take advantage of this new persistency domain. Given that caches can now be considered persistent, one possible approach would be to avoid the use of software instrumentation techniques and rely entirely on existing HTM mechanisms. Another possible approach would be to use an existing state-of-the-art solution like SPHT [5], which was designed with ADR in mind but can also be used in an eADR environment.

3.1.1 Pure HTM Approach

A pure HTM approach has the advantage of the simplicity of implementation, operating directly on the data stored in PM. It relies on mechanisms provided by the hardware without the added complexity of software instrumentation to ensure data has been persisted. However, even though it is now possible to guarantee the persistence of a successful transaction entirely through mechanisms provided by the hardware, it is still not possible to guarantee progress by relying exclusively on these mechanisms.

HTM is still a best-effort synchronization mechanism [19, 20] and requires a software-based fallback path in order to ensure progress. This fallback path can be implemented in the form of a Single Global

Lock (SGL) that is used whenever a transaction repeatedly aborts over a pre-configured number of times. The SGL causes any concurrent transactions to be aborted, ensuring there are no conflicts with other transactions. Since this mechanism is implemented via software and memory access is performed directly on PM, any write operations made inside the SGL fallback path are written directly to PM. If an SGL transaction fails or aborts after having performed any write operations, those partial changes will still be present in PM, violating the principle of atomicity and leading to an incorrect system state. This requires the use of additional instrumentation to maintain consistency.

Algorithm 2 SGL with Undo Log

```
    Persistent Variables
 1: ${}^{p}undoLog[]$
    Thread Local Volatile Variables
 2: ${}^{v}SGL$

 3: function BEGINTX
 4:     while !CAS(&${}^{v}SGL$, 0, 1) do                ▷ Take SGL
 5:         WAITPRECEDINGTXS                        ▷ Writing the SGL causes an HTM abort
 6:     CREATEUNDOLOG                               ▷ Creates a new undo log
 7: function WRITE(addr, val)
 8:     prevVal← *addr                              ▷ Save previous value
 9:     undoLogWrite(addr, prevVal)                 ▷ Log previous value PM
10:     *addr← val                                  ▷ Execute write
11: function COMMITTX
12:     CLEARUNDOLOG                                ▷ Undo log no longer necessary
13:     SGL← 0                                      ▷ Release SGL, needs memory barrier
```

A possible solution for this problem when operating directly on PM is the use of an undo log used to track write operations performed by an SGL transaction. In case of a failure or an aborted transaction, this undo log can be used to revert changes that had already been written to persistent memory and restore the consistent state of the system before that SGL transaction had been initiated. The algorithm for this is shown in Algorithm 2. As far as it was possible to determine, this thesis is the first work that implements and evaluates HTM for eADR using an instrumented fallback path.

3.1.2 Using SPHT

Even though SPHT was designed for an ADR environment and does not take advantage of the durable caches introduced with eADR, it can still be used in this new environment. All the properties SPHT guarantees in ADR are also enforced in the eADR environment. This approach is more complex and does not provide any benefits over executing in an ADR environment, but can potentially provide better performance due to the use of shadow paging in DRAM.

3.1.3 Comparative Study

A small comparative study of both these approaches was conducted in order to determine the feasibility of each solution and to determine if existing state-of-the-art solutions can still provide any benefits over the simpler approach of using pure HTM mechanisms with a software fallback path to ensure progress. The test for the study was performed with a synthetic benchmark in which each transaction generates a total of 5 read and 5 write operations at random over a uniform persistent heap.

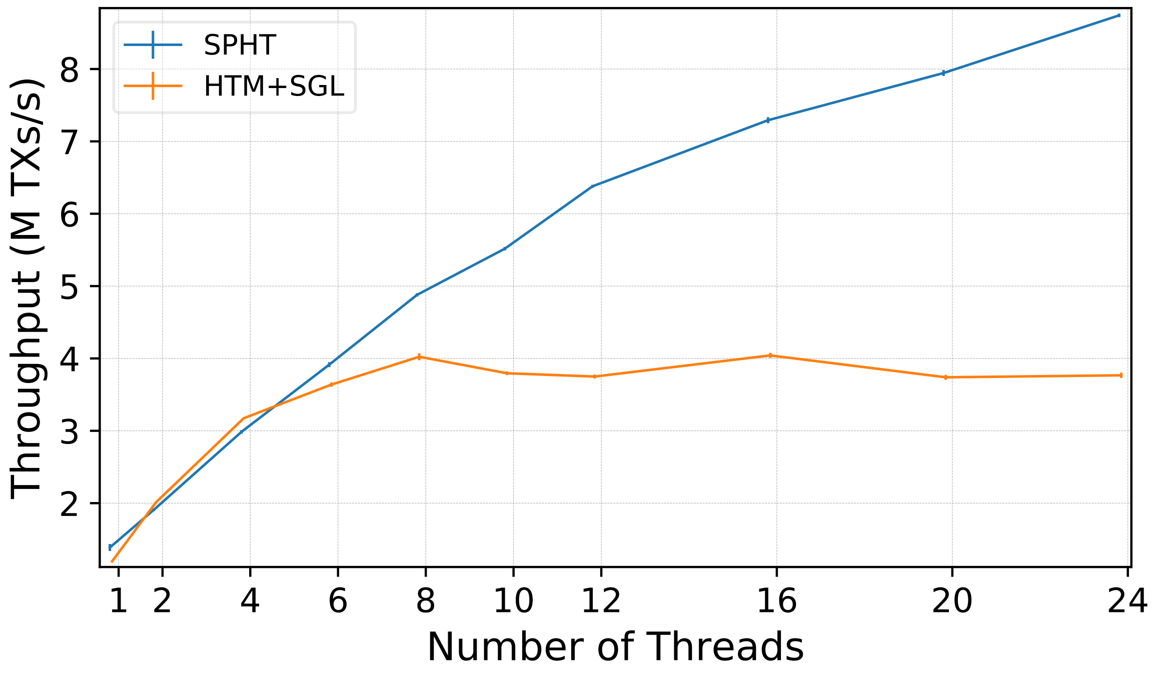

Figure 3.1: Throughput comparison of SPHT and HTM+SGL with mixed access pattern

This study was conducted on an Intel Xeon processor with 12 cores and 24 threads; more detailed specifications for the machine are available in Section 4.1. The tests were performed with a synthetic benchmark in which each transaction generates a fixed number of read and write operations at random over a uniform 1 GB persistent heap space in which each thread accesses its own private memory pool and all results are the average of 10 runs of each test.

In a first scenario, shown on Figure 3.1 and Figure 3.2, the workload is comprised of small transactions with a mixed access pattern in which each transaction generates a total of 5 read and 5 write operations. These results show that even though SPHT does not take advantage of the durable caches offered by the eADR environment, it still provides a significant advantage in throughput and scalability over the HTM approach, which is limited by the bandwidth capacity of the PM module.

In a second scenario, shown on Figure 3.3 and Figure 3.4, the workload is comprised of large transactions with a read-intensive access pattern in which each transaction generates a total of 45 read and 5 write operations. Latency times for write operations are significantly higher than for read operations in currently available PM modules [10]. A read-intensive access pattern could help avoid performance degradation in the HTM+SGL solution, but the results show that SPHT still provided a

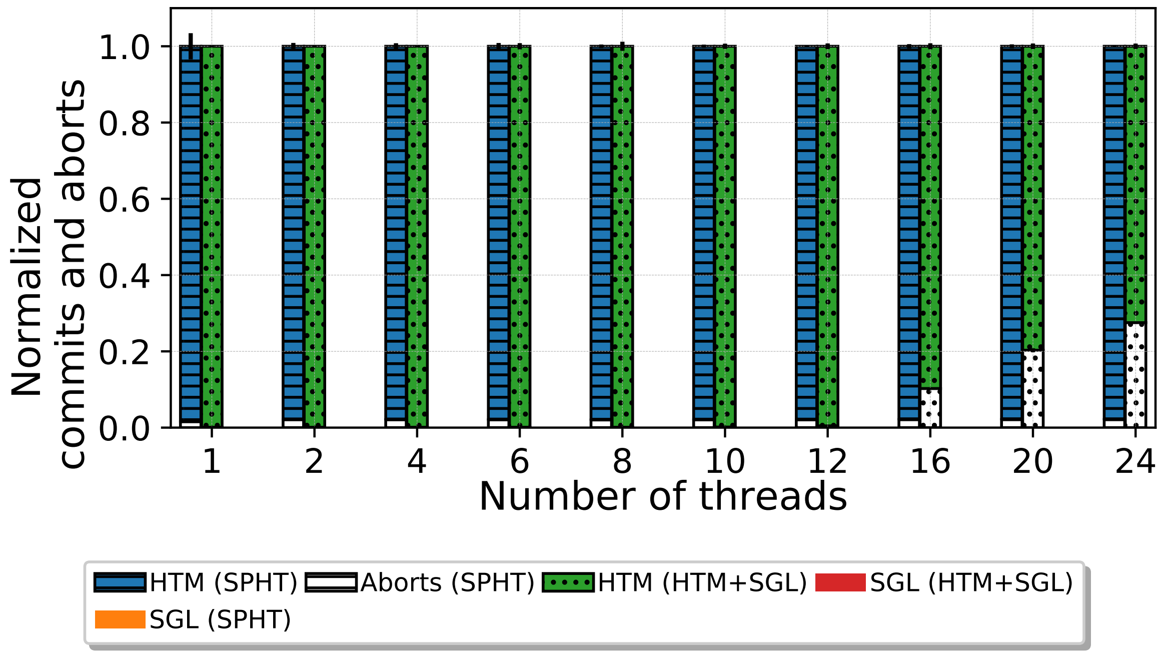

Figure 3.2: Breakdown of committed (via the HTM and SGL paths) and aborted transactions for SPHT and HTM+SGL with mixed access pattern

significant throughput and scalability advantage over HTM+SGL operating directly on PM.

These results show that the software instrumentation techniques used in state-of-the-art solutions like SPHT still provide benefits over operating directly on PM and motivate the need to develop a new solution that improves on existing state-of-the-art solutions by taking advantage of the new possibilities introduced with eADR.

3.2 Overview of SPHT

As mentioned previously, SPHT was originally developed for an ADR environment in which caches are considered volatile. This required the mechanisms described below, which can be seen in Algorithm 3, in order to ensure immediate durability and visibility of changes across threads.

3.2.1 Log Commit Marker

Each worker thread in SPHT has a private durable redo log that is used to log updates performed by each transaction. However, since caches are volatile, the updates performed by a transaction commit may still be lingering in volatile cache and not considered durable. This requires explicit flushing of the redo log to PM after the HTM commit terminates successfully. Since the log is only persisted after the

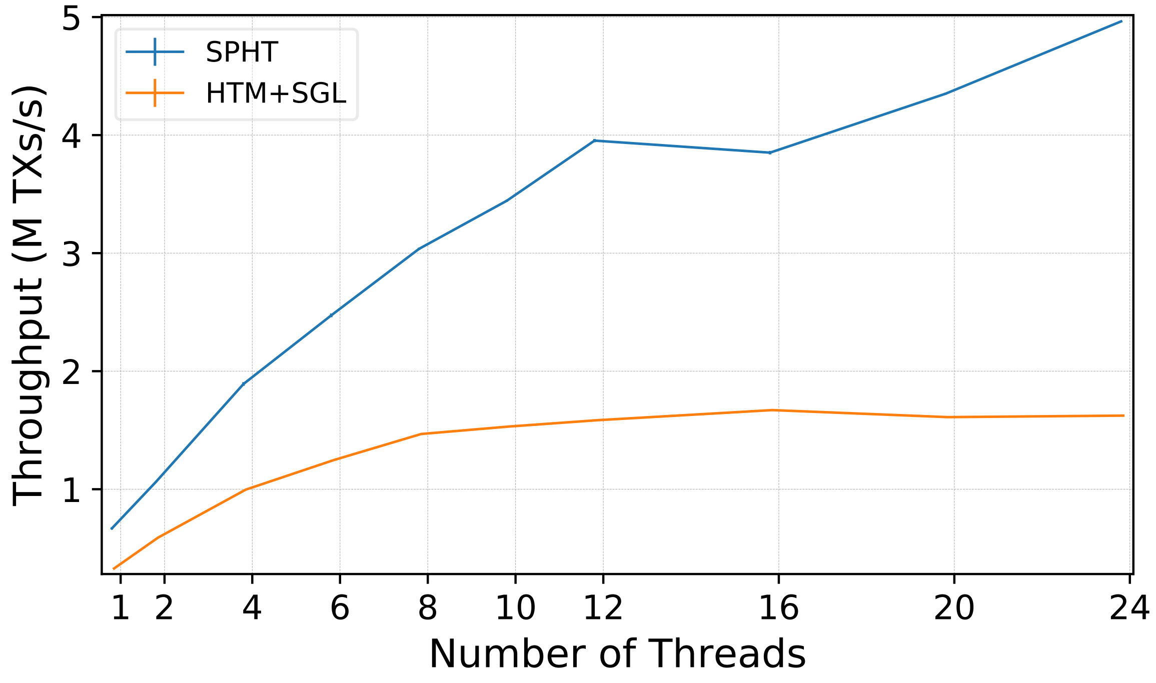

Figure 3.3: Throughput comparison of SPHT and HTM+SGL with read-intensive access pattern

transaction commits, SPHT makes use of a durable timestamped log commit marker to indicate the transaction is considered durable.

3.2.2 Wait for Preceding Transactions

One of the key ideas used in SPHT in order to overcome scalability limitations is to amortize the cost of ensuring immediate durability across multiple transaction commits. When multiple transactions are concurrently trying to commit, SPHT is able to ensure immediate durability for all of them through a single update of the durable log commit marker by writing the timestamp of the most recent durable transaction.

However, since SPHT allows threads to flush logs out of order, flushing the transaction log for a given thread is not enough for that transaction to be considered durable. At that point, there may still exist preceding transactions with lower timestamps that are not yet marked as durable, but whose changes may already have been observed by other threads. In order to solve this issue, each thread shares the timestamp of the most recent transaction along with whether the logs for that transaction have been persisted. Once the logs have been flushed the transaction enters a phase in which it scans the timestamps of the other threads and waits until all transactions with lower timestamps have been persisted. Only once all transactions with lower timestamps have finished flushing their logs can it be

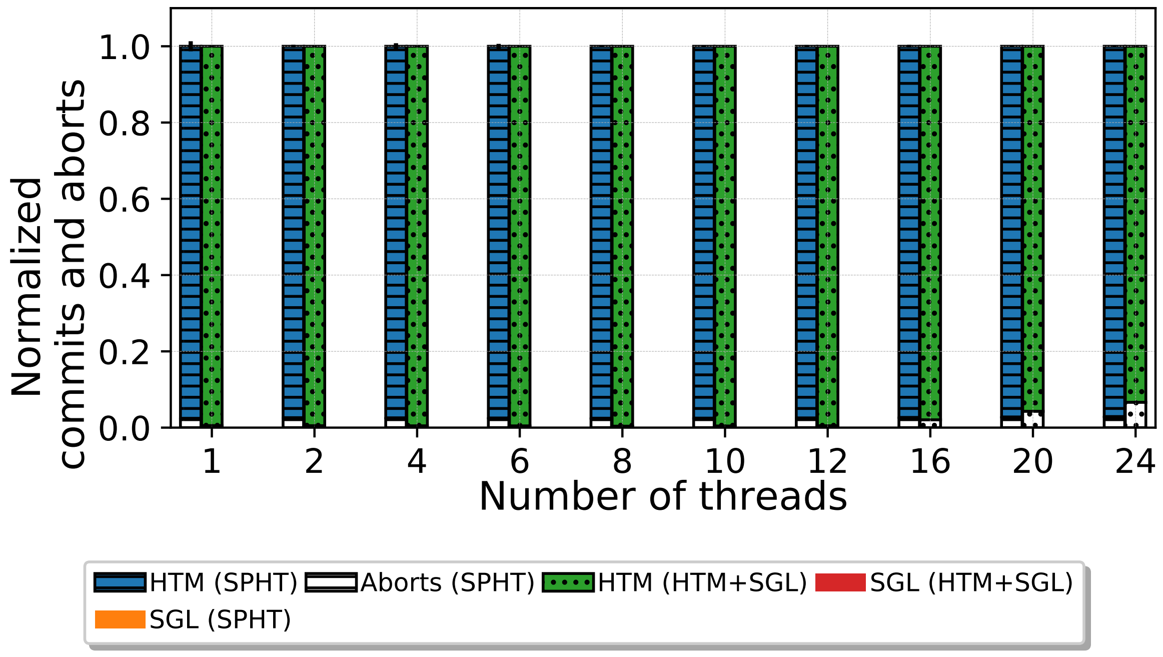

Figure 3.4: Breakdown of committed (via the HTM and SGL paths) and aborted transactions for SPHT and HTM+SGL with read-intensive access pattern

considered safe for a transaction to mark itself as durable.

It is also during this waiting phase that threads identify which transaction has the most recent timestamp and will be responsible for updating and flushing the log commit marker.

3.2.3 Flush Transaction Log

When flushing cached logs to persistent memory, SPHT needs to perform calculations to determine if it is safe to flush the cache without the possibility of generating partial log writes to PM. If the full log and commit marker fit in a single cache line, it is safe to flush that cache line. However, if they occupy more than a single cache line, it is necessary for SPHT to flush the earlier cache lines and ensure the consistency of cache pages before flushing the commit marker to persistent memory.

3.3 SPHT Simplified for eADR

The introduction of durable caches in eADR has significant implications for the durability of redo logs, which was the primary motivator behind the previously described mechanisms. Since logs can now be considered durable in cache, it is no longer necessary to ensure they have been flushed to persistent

Algorithm 3 Original SPHT

```
    Shared Volatile Variables
 1: ${}^{v}ts$[N], ${}^{v}marked$[N], ${}^{v}isUpd$[N]
    Persistent Variables
 2: ${}^{p}writeLog$[N], ${}^{p}marker$
    Thread Local Volatile Variables
 3: ${}^{v}ts'$, ${}^{v}skipCAS$

 4: function BEGINTX
 5:     ${}^{v}isUpd[myTid] \leftarrow$ FALSE
 6:     ${}^{v}skipCAS \leftarrow$ FALSE
 7:     UNSETPERSBIT(${}^{v}ts[myTid]$)                         ▷ Logs are not persistent
 8:     ${}^{v}ts[myTid] \leftarrow$ RDTSCP                      ▷ lower bound of final ts
 9:     HTM_BEGIN                                               ▷ begin hw tx
10: function WRITE(addr, val)
11:     logWrite(addr, val)                                     ▷ log to PM, no flush
12:     ∗addr← val                                              ▷ execute write
13: function COMMITTX
14:     ${}^{v}ts' \leftarrow$ RDTSCP                            ▷ store physical clock to local var.
15:     HTM_COMMIT                                              ▷ commit hw transaction
16:     ${}^{v}ts[myTid] \leftarrow ts'$                        ▷ Externalize the final timestamp
17:     if isReadOnly then                                      ▷ Read-only txs...
18:         SETPERSBIT(${}^{v}ts[myTid]$)                       ▷ ...unblock the others
19:         return                                              ▷ ...and return immediately
20:     ${}^{v}isUpd[myTid] \leftarrow$ TRUE                     ▷ Mark as update tx
21:     logCommit(${}^{p}writeLog[myTid], ts'$)                 ▷ Flush tx log.
22:     SETPERSBIT(${}^{v}ts[myTid]$)                           ▷ Signal logs are durable
23:     WAITPRECEDINGTXS
24:     UPDATEMARKER
25: function WAITPRECEDINGTXS
26:     for $t \in [0..N-1]$ do
27:                                                             ▷ Wait until prec. txs have flushed their logs
28:         while ${}^{v}ts[t] < {}^{v}ts[myTid] \wedge \neg$ISPERSBIT(${}^{v}ts[t]$) wait
29:                                                             ▷ If any update tx with large ts exists...
30:         if ${}^{v}ts[t] > {}^{v}ts[myTid] \wedge {}^{v}isUpd[t]$ then
31:             ${}^{v}skipCAS \leftarrow$ TRUE                  ▷ this tx can skip the CAS
32: function UPDATEMARKER
33:                                                             ▷ Is it needed to and am I responsible for updating ${}^{p}marker$?
34:     if ${}^{p}marker < {}^{v}ts[myTid] \wedge \neg{}^{v}skipCAS$ then
35:         val← ${}^{p}marker$
36:         while val < ${}^{v}ts[myTid]$ do
37:             val ← CAS(${}^{p}marker$, val, ${}^{v}ts[myTid]$)
38:         if (CAS was successful) then
39:             flush(${}^{p}marker$)
40:             ${}^{v}marked[myTid] \leftarrow {}^{v}ts[myTid]$       ▷ Signals ${}^{p}marker$ is flushed.
41:             return
42:     while TRUE do                                           ▷ Wait till flush of ${}^{p}marker$
43:         for $t \in [0..N-1]$ do                             ▷ ...is complete
44:             if ${}^{v}marked[t] \geq {}^{v}ts[myTid]$ then return
```

memory before the transaction itself can be considered durable. The transaction can now be considered durable as soon as it commits successfully, rendering the global log marker unnecessary.

Likewise, it is now possible to consider preceding transactions with an earlier timestamp to be durable without the need to share whether their logs have been flushed to persistent memory or not. Any transaction with an earlier timestamp will have successfully committed and written their logs either to persistent memory or to cache, which is now considered durable. This means it is no longer necessary for transactions with an earlier timestamp to wait for any preceding transactions.

Additionally, the process of flushing logs to persistent memory, along with determining which cache lines need to be flushed, is no longer necessary.

These changes allow the algorithm for SPHT-eADR to take advantage of the new possibilities introduced in eADR and to be significantly simplified in comparison to the original version of SPHT, as shown in Algorithm 4.

Algorithm 4 SPHT-eADR

```
    Persistent Variables
 1: $^{p}writeLog$[N]
    Thread Local Volatile Variables
 2: $^{v}isUpdate$

 3: function BEGINTX
 4:     HTM_BEGIN                                              ▷ Begin hw tx
 5: function WRITE(addr, val)
 6:     $^{v}isUpdate \leftarrow$ true
 7:     logWrite(addr, val)                                    ▷ Log to PM, no flush
 8:     *addr← val                                             ▷ Execute write
 9: function COMMITTX
10:     if isUpdate then
11:         logCommit($^{p}writeLog[myTid]$, RDTSCP)           ▷ No flush required
12:     HTM_COMMIT                                ▷ SGL commit needs a memory barrier
```

3.3.1 Maintaining Flushes

Even though flush operations are no longer required to ensure the persistence in an eADR environment, they may still be beneficial for performance by proactively removing cache lines containing data which no longer has temporal locality [7], as is the case with log entries for transactions that have already been committed. This is shown on Line 17 of Algorithm 5

Algorithm 5 SPHT-eADR with Flushes

```
    Persistent Variables
 1: $^{p}writeLog$[N]
    Thread Local Volatile Variables
 2: $^{v}isUpdate, {}^{v}txLogStart, {}^{v}txLogEnd$

 3: function BEGINTX
 4:     $^{v}txLogStart \leftarrow$ logNextPos()               ▷ Record starting log position
 5:     $^{v}txLogEnd \leftarrow$ txLogStart
 6:     HTM_BEGIN                                              ▷ Begin hw tx
 7:     *addr← val                                             ▷ Execute write
 8: function WRITE(addr, val)
 9:     $^{v}isUpdate \leftarrow$ true
10:     logWrite(addr, val)                                    ▷ Log to PM, no flush
11:     $^{v}txLogEnd \leftarrow$ logNextPos()                 ▷ Update current log position
12:     *addr← val                                             ▷ Execute write
13: function COMMITTX
14:     if isUpdate then
15:         logCommit($^{p}writeLog[myTid]$, RDTSCP)
16:     HTM_COMMIT                                ▷ SGL commit needs a memory barrier
17:     flushCache(txLogStart, txLogEnd)          ▷ Flush cache for updated log section
```

Given that changes lingering in cache can be considered persistent and that the version of SPHT optimized for eADR no longer needs to maintain a commit marker, it is possible to flush redo logs using an out-of-order operation like CLWB or CLFLUSHOPT without the need to issue a memory fence and wait for the flushes to finish.

3.4 Implementation of SPHT-eADR

SPHT-eADR exposes PM to the application via a persistent heap created by memory-mapping the persistent data stored in a PM-aware filesystem into the application address space [43] using the host Operating System (OS). Being based on SPHT, SPHT-eADR follows the same architecture with two main processes (Transaction Executers and Log Replayers) that were earlier described in Section 2.4.2. The transaction executor process memory-maps a persistent heap into its address space using Copy-on-Write provided by the OS which creates a shadow copy of the persistent heap. The process also spawns multiple worker threads that share access to this shadow copy. Changes to the shadow copy are not transmitted back to the persistent heap. Instead, worker threads track updates through private redo logs, implemented with a circular buffer, which contains an ordered sequence of transactions and timestamps. These logs can eventually be replayed in order to propagate changes back to the persistent heap. Transactions in SPHT-eADR utilize the underlying support for HTM and switch to a fallback Single Global Lock software-based commit mechanism when a transaction fails a pre-configured number of times. When this fallback mode is activated, all concurrent hardware-based transactions are immediately aborted.

4 Experimental Evaluation

Contents

This chapter presents the results of an experimental evaluation of SPHT-eADR, a new solution based on SPHT and optimized for an eADR environment (previously described in Section 3.3), and seeks to answer the question of whether shadow paging and software instrumentation techniques used by state-of-the-art solutions like SPHT can still be used to improve performance given the availability of eADR. The performance of SPHT-eADR was compared to the standard version of SPHT and an almost pure HTM mechanism with a software fallback path to ensure progress, as well as different versions of SPHT-eADR with preemptive flushing of logs with low temporal locality. These solutions were evaluated using synthetic benchmarks with no contention, STAMP, and TPC-C. This section is structured as follows. Section 4.1 describes the details about the testing platform and the benchmarks that were used. Section 4.2 tests two different approaches to flushing in SPHT-eADR. And finally, Section 4.3 presents and discusses the gathered experimental results gathered through synthetic benchmarks, STAMP, and TPC-C.

4.1 Experimental Settings

All experiments were conducted in a dual-socket system using a single Intel Xeon Gold 5317 3.00 GHz 3rd Generation Intel Xeon Scalable processor with 12 cores and 24 hardware threads, equipped with 128 GB of DRAM (4x 32 GB) and 512 GB of Intel Optane DC PM 200 (4x 128 GB) with interleaved access and configured in App Direct mode. These experiments evaluate the performance of:

- HTM+SGL: plain HTM with a software fallback using a single global lock;
- SPHT [5]: original version of SPHT developed for an ADR environment;
- SPHT-eADR: new solution based on SPHT and optimized for an eADR environment; see Section 3.3;
- SPHT-eADR-Flush: a version of SPHT-eADR with logic to flush the redo log after a successful commit; see Section 3.3.1.

All described solutions make use of HTM and fall back to SGL when a transaction fails after 10 retries. All results are the average of 10 runs.

The synthetic benchmark is configured with a 1 GB heap space which is split into private memory pools for each thread, ensuring that there are no conflicting transactions. Before beginning the execution of the benchmark, memory pages are pre-touched in order to simulate a long-running process. Each iteration of the test runs for 5 seconds and each transaction generates a pre-configured number of read and write operations to random memory addresses within the memory pool for each thread. This benchmark evaluates the performance of the various solutions in scenarios where every transaction is able to be executed concurrently without incurring conflicts in order to evaluate the scalability and possible bottlenecks for each solution.

STAMP [8] is a benchmark suite designed for transactional memory systems and includes transactional applications that, even though they were not originally designed with PM in mind, could still be able to benefit from crash-tolerance in a PM system. STAMP was previously used to test SPHT, along with several other related solutions [1,3,4,9] in the same field.

Although the STAMP benchmark suite includes 8 different benchmarks, this evaluation does not consider the Bayes application, as it is known to provide unstable performance results [5,44].

TPC-C [9] is a well-known benchmark that is widely used to evaluate database and transactional systems. The benchmark is composed of five transactions: three update transactions (New Order, Payment, and Delivery), and two read-only transactions (Status Order and Stock Level). This evaluation implemented three of these transactions, Payment, New Order, and Delivery. New Order and Delivery are transactions that contain item accesses dependent on other previous accesses.

4.2 Flushing Approach

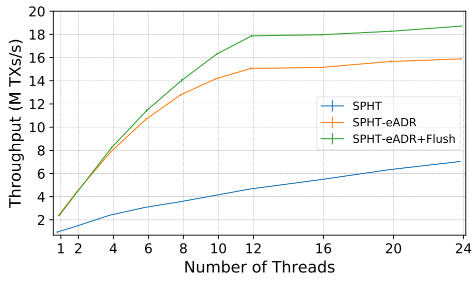

Figure 4.1: Throughput for synthetic benchmark with 5 writes and 5 reads using CLFLUSHOPT

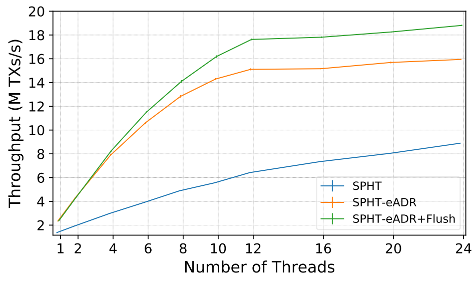

Figure 4.2: Throughput for synthetic benchmark with 5 writes and 5 reads using CLWB

This synthetic benchmark experiment compares the throughput and breakdown of committed and aborted transactions of the regular version of SPHT, SPHT-eADR without flushes, and SPHT-eADR with flushes using a balanced workload in which each transaction performs 5 read and 5 write operations. The test was repeated using both CLWB and CLFLUSHOPT operations in order to implement the log flushing phase taking place after transaction commit in these solutions. The CLFLUSHOPT instruction flushes data out of the CPU cache and invalidates it whereas CLWB flushes the data without invalidating the cache lines. This allows for evaluating not just the impact of preemptively flushing logs to PM but also determining whether cache invalidation has any negative effect. Figure 4.1 and Figure 4.2 show the throughput of the 3 solutions using CLFLUSHOPT and CLWB, respectively. SPHT-eADR and SPHT-eADR with flushes have similar throughput curves, scaling well up to the number of physical cores. Once Hyper-Threading is used, the curve flattens and throughput stays almost constant. The version with flushes performs noticeably better than the version without flushes, indicating that preemptively flushing data with low temporal locality does provide a performance boost. We argue that this increases the effectiveness of the caching layer by asking the hardware to flush log data that is unlikely to be reaccessed shortly thereafter. The original version of SPHT scales more linearly up to 24 threads but at much lower throughput levels. The results of the tests with CLWB and CLFLUSHOPT are similar, indicating that invalidating the cache lines for the redo logs, which have low temporal locality, does not have a negative effect on performance.

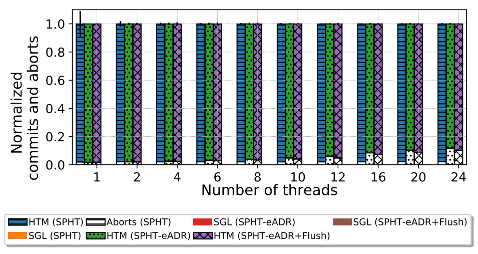

Figure 4.3: Breakdown of committed (via the HTM and SGL paths) and aborted transactions for synthetic benchmark with 5 writes and 5 reads using CLFLUSHOPT

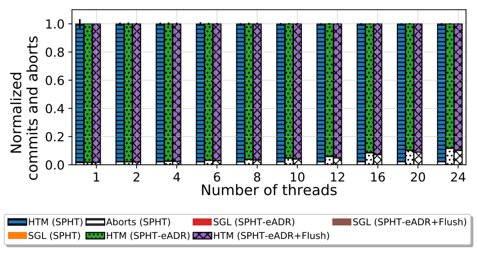

Figure 4.4: Breakdown of committed (via the HTM and SGL paths) and aborted transactions for synthetic benchmark with 5 writes and 5 reads using CLWB

Figure 4.3 and Figure 4.4 show the breakdown of committed and aborted transactions for the solutions, with committed transactions being split into HTM or SGL commit mechanisms. With this workload, the abort rate is very low, only growing a bit at higher thread counts. However, it is worth noting that even though the throughput of the original version of SPHT is lower, it does have a lower abort rate than SPHT-eADR and SPHT-eADR with flushes. The higher throughput of SPHT-eADR means that more write requests reach the write-pending queue of the PM module, generating more aborts.

4.3 Evaluating SPHT-eADR

4.3.1 Synthetic Benchmark

This experiment includes 6 variations of the synthetic (no contention) benchmark, covering a variety of scenarios encompassing all combinations of small or large transactions (10 or 50 memory accesses) with read-intensive, write-intensive, or mixed access patterns. The lack of conflicts in this case also allows for the evaluation of scalability and identification of possible bottlenecks for each solution.

SPHT-eADR has the best performance in read-intensive applications (see Figure 4.5 and Figure 4.6), reaching 2–3x the throughput performance of SPHT and 4x the throughput of HTM+SGL. SPHT-eADR with flush operations performs very close to SPHT-eADR, but flushing does not present an advantage in these applications. All versions of SPHT only generate log entries for write operations. As such,

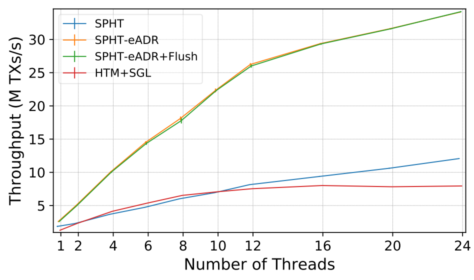

Figure 4.5: Throughput for synthetic benchmark with 1 write and 9 reads

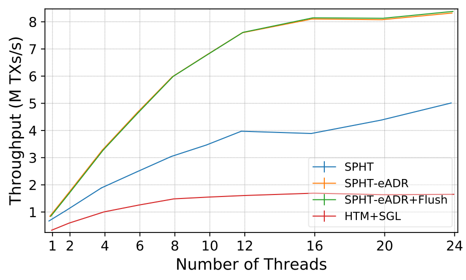

Figure 4.6: Throughput for synthetic benchmark with 5 writes and 45 reads

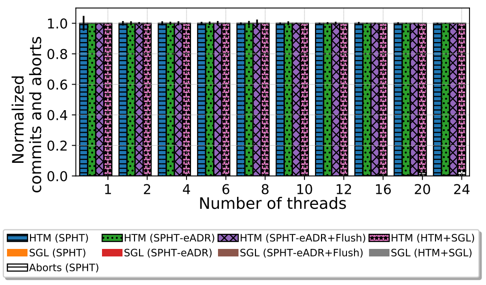

Figure 4.7: Breakdown of committed (via the HTM and SGL paths) and aborted transactions for synthetic benchmark with 1 write and 9 reads

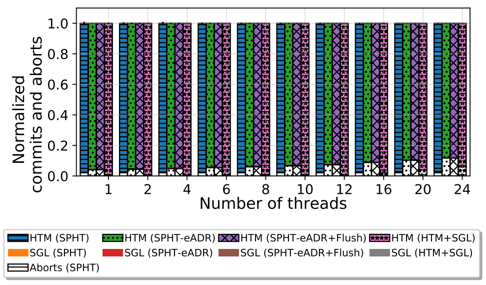

Figure 4.8: Breakdown of committed (via the HTM and SGL paths) and aborted transactions for synthetic benchmark with 5 writes and 45 reads

it is expectable for flushing the logs to have a small impact here given the small amount of memory generated by logs in cache.

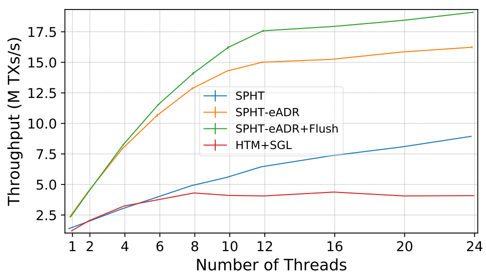

Figure 4.9: Throughput for synthetic benchmark with 5 writes and 5 reads

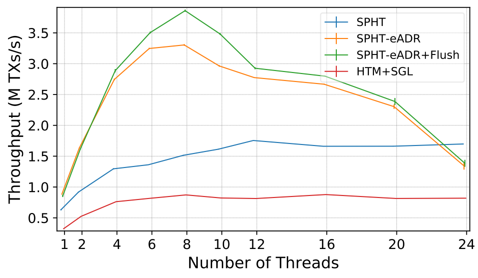

Figure 4.10: Throughput for synthetic benchmark with 25 writes and 25 reads

With a mixed access pattern (see Figure 4.9 and Figure 4.10) applications start to give SPHT-eADR with flushing an advantage, performing better than all other solutions. Write operations are noticeably slower though, causing the throughput for each application to reduce significantly compared to the read-intensive application performing the same number of operations. Again, SPHT-eADR and SPHT-eADR with flushes reach roughly 2x the performance of SPHT and 3–4x that of HTM+SGL.

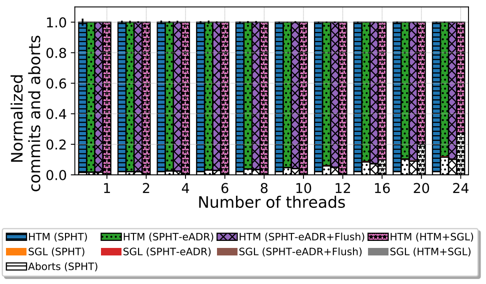

Figure 4.11: Breakdown of committed (via the HTM and SGL paths) and aborted transactions for synthetic benchmark with 5 writes and 5 reads

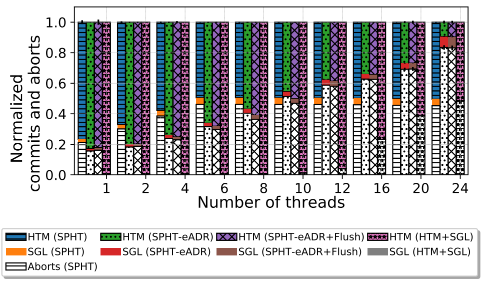

Figure 4.12: Breakdown of committed (via the HTM and SGL paths) and aborted transactions for synthetic benchmark with 25 writes and 25 reads

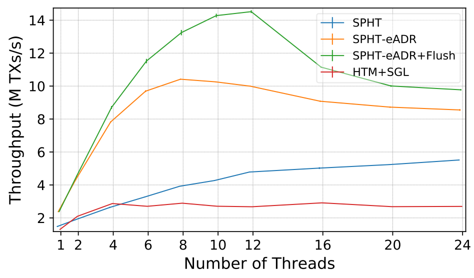

Figure 4.13: Throughput for synthetic benchmark with 9 writes and 1 read

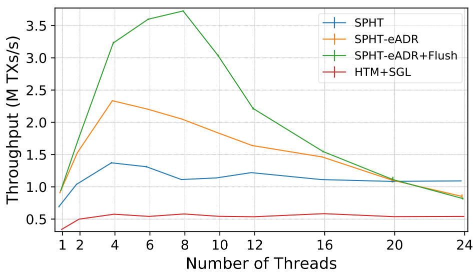

Figure 4.14: Throughput for synthetic benchmark with 45 writes and 5 reads

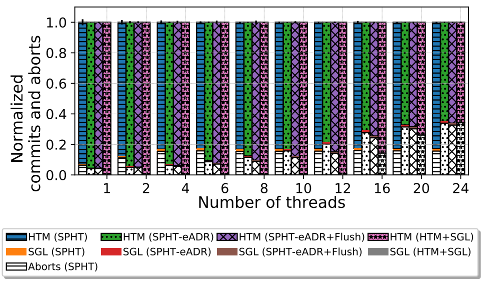

Figure 4.15: Breakdown of committed (via the HTM and SGL paths) and aborted transactions for synthetic benchmark with 9 writes and 1 read

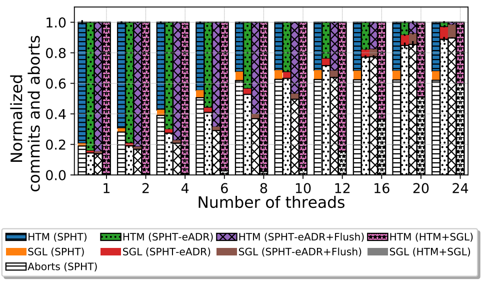

Figure 4.16: Breakdown of committed (via the HTM and SGL paths) and aborted transactions for synthetic benchmark with 45 writes and 5 reads

In contrast with read-intensive applications, write-intensive applications give SPHT-eADR with flushes a significant advantage in performance, albeit mostly at lower thread counts. Throughput for SPHT-eADR peaks at 8–12 threads, and declines noticeably after that, particularly in applications with a combination of large transactions and write-intensive access patterns.

The original version of SPHT has lower peak throughput but does not suffer from degraded performance with a high number of threads due to the waiting mechanism used in the commit phase which helps maintain a lower number of aborted transactions at higher thread counts.

Overall, taking these results as a whole, it is possible to see that HTM+SGL reaches a plateau early on with just a few threads and does not scale further, being limited by the higher latency of PM when compared to DRAM. The new versions of SPHT-eADR and SPHT-eADR with flushes scale better and reach much higher peak throughput with a lower number of cores. However, performance degrades with a higher number of threads, especially for large and write-intensive workloads.

4.3.2 STAMP Benchmark

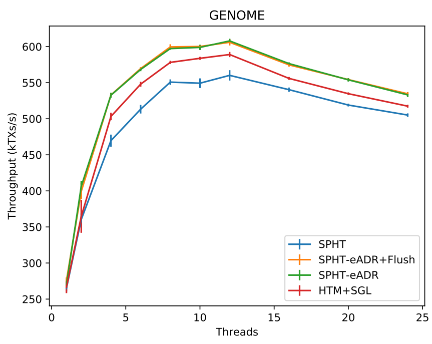

Figure 4.17: Throughput for GENOME benchmark

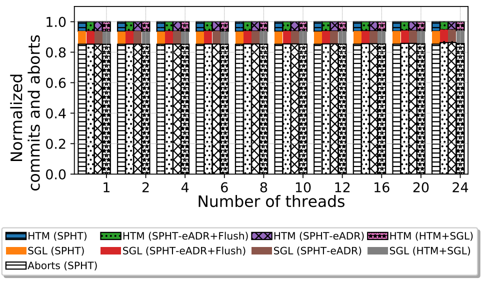

Figure 4.18: Breakdown of committed (via the HTM and SGL paths) and aborted transactions for GENOME benchmark

GENOME (see Figure 4.17) is a medium contention benchmark and not very favourable towards scalability, given that there is a high likelihood of generating conflicts between transactions, as can be seen on Figure 4.18 by the abort rate of over 80%. SPHT-eADR (with and without flushes) performed the best in this benchmark and reached peak throughput at 12 threads. The results for all solutions are quite closely correlated, which may be due to the high number of aborts causing most transactions to be committed via the SGL.

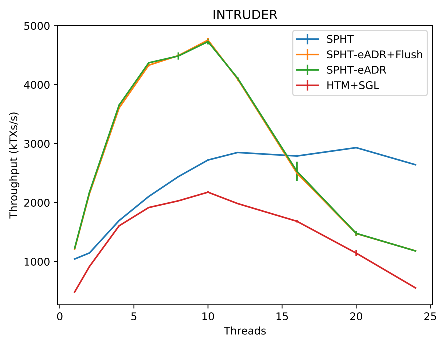

Figure 4.19: Throughput for INTRUDER benchmark

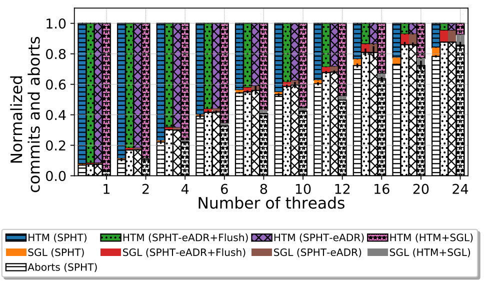

Figure 4.20: Breakdown of committed (via the HTM and SGL paths) and aborted transactions for INTRUDER benchmark

INTRUDER (see Figure 4.19) is also a contention-prone benchmark that is not favourable to scalability. SPHT-eADR (with and without flushes) reaches 1.5–2x higher peak throughput than other solutions, but degrades quickly as the number of threads increases due to a corresponding increase in the number of aborts, as can be seen on Figure 4.20. The original version of SPHT stays significantly lower in terms of maximum throughput but scales more gracefully to a large number of threads without degrading performance.

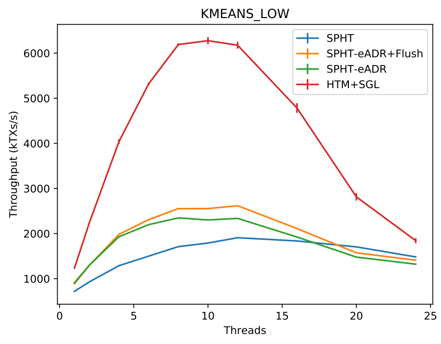

Figure 4.21: Throughput for KMEANS LOW benchmark

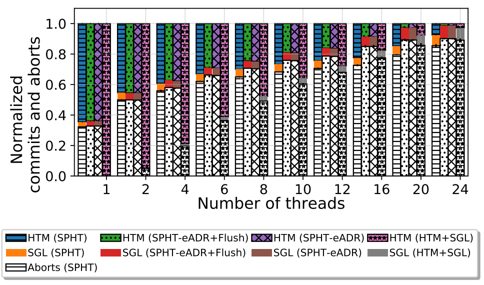

Figure 4.22: Breakdown of committed (via the HTM and SGL paths) and aborted transactions for KMEANS LOW benchmark

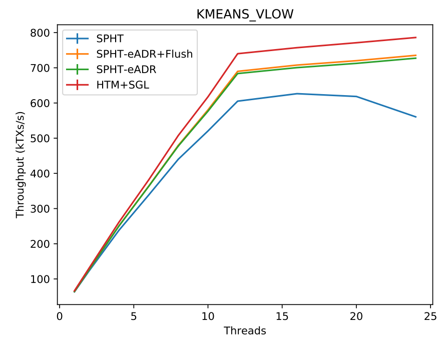

Figure 4.23: Throughput for KMEANS VLOW benchmark

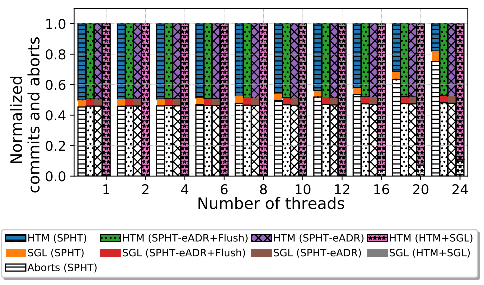

Figure 4.24: Breakdown of committed (via the HTM and SGL paths) and aborted transactions for KMEANS VLOW benchmark

HTM+SGL performs very favourably with KMEANS LOW (see Figure 4.21) and KMEANS VLOW (see Figure 4.23), both in terms of throughput and abort rate, achieving over 2x the throughput in KMEANS LOW when compared to all the versions of SPHT and SPHT-eADR. This can be explained by the fact that HTM+SGL has a much lower abort rate at lower thread counts and is able to commit via the HTM path most of the times. In KMEANS VLOW the slowdown is not as large as in KMEANS LOW, with all solutions reaching their peak throughput at 12 threads and stabilizing when additional HyperThreading threads are added. Looking at the committed and aborted transaction breakdown in Figure 4.22 and

Figure 4.24 it is possible to see that the number of aborted transactions increases with the number of threads in KMEANS LOW and stays stable in KMEANS VLOW, with the exception of the original version of SPHT that starts getting more aborted transactions at high thread counts. HTM+SGL is able to maintain a much lower abort rate in this case.

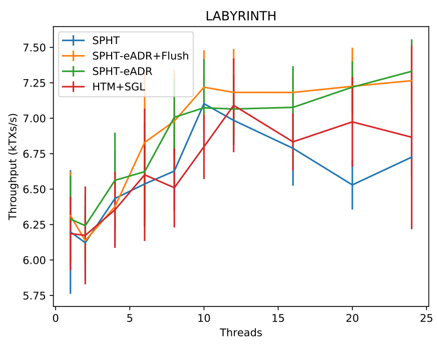

Figure 4.25: Throughput for LABYRINTH benchmark

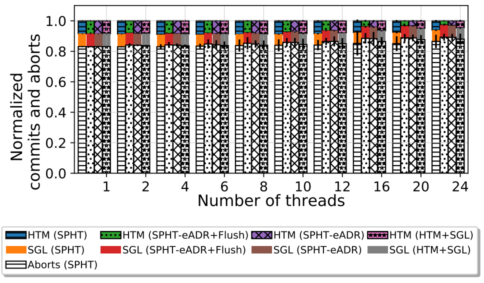

Figure 4.26: Breakdown of committed (via the HTM and SGL paths) and aborted transactions for LABYRINTH benchmark

LABYRINTH (see Figure 4.25) is a benchmark with large transaction sizes and medium contention, making it a benchmark that is not very well suited for HTM. It is difficult to take significant conclusions regarding the throughput of individual solutions from this benchmark given that there is very high variability in throughput over the 10 execution runs. However, Figure 4.26 does show that LABYRINTH has an abort rate of over 80% with all solutions.

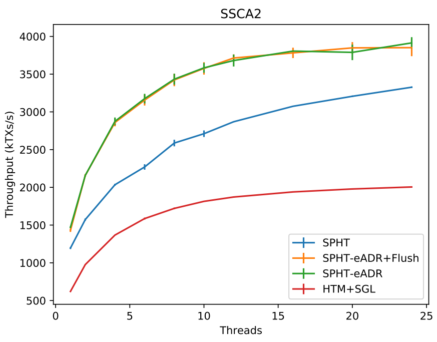

Figure 4.27: Throughput for SSCA2 benchmark

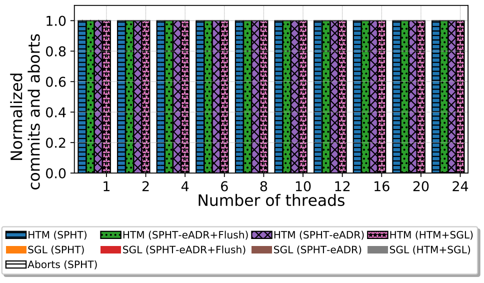

Figure 4.28: Breakdown of committed (via the HTM and SGL paths) and aborted transactions for SSCA2 benchmark

Both SSCA2 and VACATION LOW (see Figure 4.27 and Figure 4.29) are low-contention benchmarks that are favourable to HTM. This can be seen on Figure 4.28 which shows that SSCA2 has a 100% commit rate entirely through HTM, without falling back to SGL. All tested solutions scale smoothly up to 24 threads, but these are the benchmarks where both versions of SPHT-eADR show the best scalability,

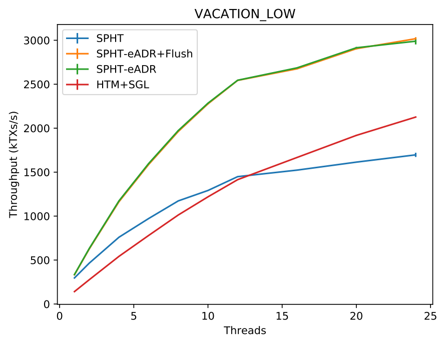

Figure 4.29: Throughput for VACATION LOW benchmark

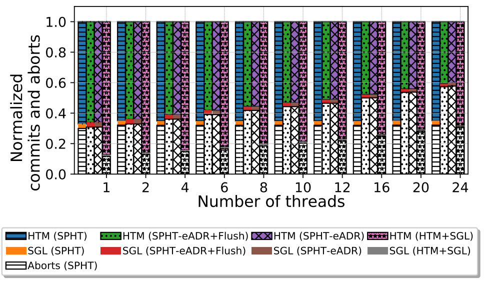

Figure 4.30: Breakdown of committed (via the HTM and SGL paths) and aborted transactions for VACATION LOW benchmark

reaching 2x the peak throughput of HTM+SGL and 1.2x of SPHT without any degradation in performance at higher thread counts.

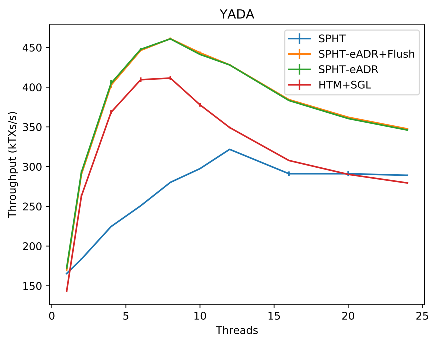

Figure 4.31: Throughput for YADA Benchmark

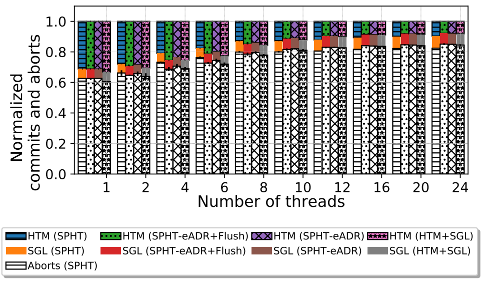

Figure 4.32: Breakdown of committed (via the HTM and SGL paths) and aborted transactions for YADA benchmark

YADA (see Figure 4.31) is another contention-prone benchmark that generates large transactions, providing an unfavourable running environment for HTM with a high percentage of aborted transactions, visible in Figure 4.32. Both versions of SPHT-eADR perform significantly better than SPHT, but peak throughput is reached at around 8 threads and performance degrades significantly after that. This is to be expected from a contention-prone benchmark like YADA.

4.3.3 TPC-C Benchmark

Figure 4.33 and Figure 4.34 show the results of TPC-C implemented with the three update transactions: New Order, Payment, and Delivery and configured with 32 warehouses, 95% payment, 3% delivery, and 2% new order transactions. Both versions of SPHT-eADR, with and without flushes, perform very similarly, scaling very well up to 12 threads with 1.5x the throughput of SPHT but degrading rapidly from thread 13 on, once HyperThreading is in use. The original version of SPHT scales very favourably in this test, however. It does not reach the same maximum throughput that SPHT-eADR is able to reach at 12 threads, but it continues scaling upwards even with HyperThreading due to the waiting mechanism used in the commit phase. Figure 4.34 shows that the number of aborted transactions increases with the number of threads for most solutions, with the exception of SPHT which generates a lower percentage of aborted transactions with higher thread counts.

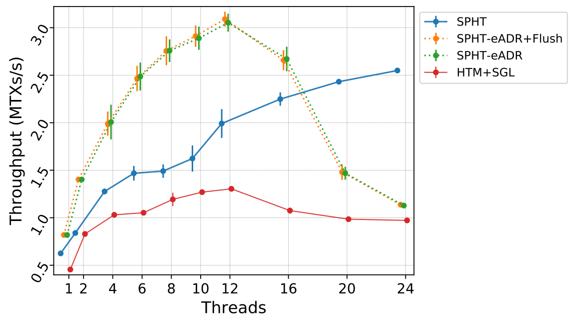

Figure 4.33: Throughput for TPC-C

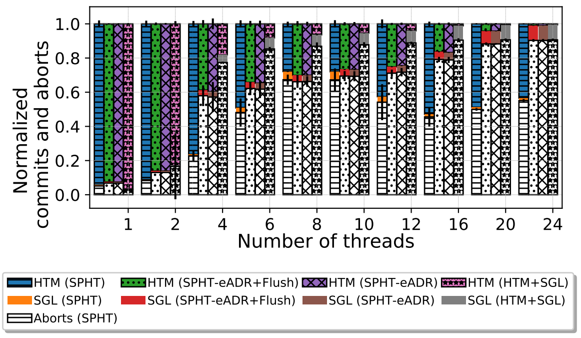

Figure 4.34: Breakdown of committed (via the HTM and SGL paths) and aborted transactions for TPC-C

5 Conclusion

Contents

5.1 Conclusions

The goal of this dissertation was to revisit state-of-the-art proposals for durable HTM based on DRAM shadow paging techniques and look at them through the lens of eADR, a new persistence domain for computer systems that offers the possibility of treating CPU caches as durable. Even though these techniques are no longer required to ensure durability in this new environment, they can still provide significant performance benefits.

Having revisited and studied these proposals, this dissertation introduced SPHT-eADR, a new solution for durable HTM optimized for systems with durable caches that makes use of DRAM shadow paging techniques.

This approach had not yet been studied in an eADR environment, but the experimental evaluation of SPHT-eADR shows that it significantly improves on the performance of previous state-of-the-art solutions by providing higher performance and fewer overheads, reaching 2–3x the throughput performance of SPHT and 4x the throughput of HTM+SGL in synthetic (no contention) benchmarks and 1.5x the throughput of SPHT and 2x the throughput of HTM+SGL at similar thread counts on TPC-C.

5.2 Future Work

One topic that was approached during the execution of this dissertation was the issue of support for large heap allocation in systems that make use of DRAM shadow paging. Most state-of-the-art systems are limited by the size of the DRAM pool available, causing them to either fail or drastically degrade performance once that limit is reached due to the cost of the operating system swapping memory in and out to disk. The main strategy that has been considered in the literature and is utilized by DudeTM [2] consists of paying the cost for restoring the content on page-in, when a page fault occurs. However, there are unexplored regions and alternative approaches that seem interesting and can potentially improve performance, such as shifting the cost to the page-out event instead, avoiding overheads on page-in events during execution.

Another future avenue of research would be the prevention of performance degradation of SPHT-eADR at higher thread counts, which could be addressed with the introduction of rate-limiting or some other form of back-off mechanism. There were preliminary experiments with the use of simple static back-off mechanisms but the results were not conclusive. One interesting path would be how to automatically determine the amount of rate-limiting or back-off time required to prevent performance degradation at higher thread counts without hurting performance at lower thread counts, borrowing ideas from previous literature in the area of self-tuning [45, 46].

Bibliography

[1] D. Castro, P. Romano, and J. Barreto, “Hardware Transactional Memory Meets Memory Persistency,” in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018, pp. 368–377.

[2] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren, “DudeTM: Building Durable Transactions with Decoupling for Persistent Memorya,” in *ASPLOS ’17: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, April 2017, pp. 329—343.

[3] E. Giles, K. Doshi, and P. Varman, “Continuous checkpointing of htm transactions in nvm,” in *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*. Association for Computing Machinery, 2017, p. 70–81.

[4] K. Genç, M. D. Bond, and G. H. Xu, “Crafty: efficient, HTM-compatible persistent transactions,” in *PLDI 2020: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2020, pp. 59–74.

[5] D. Castro, A. Baldassin, J. Barreto, and P. Romano, “SPHT: Scalable Persistent Hardware Transactions,” in *19th USENIX Conference on File and Storage Technologies (FAST 21)*, 2021, pp. 155–169.

[6] P. Zardoshti, M. Spear, A. Vosoughi, and G. Swart, “Understanding and Improving Persistent Transactions on Optane^{TM} DC Memory,” in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 348–357.

[7] S. Gugnani, A. Kashyap, and X. Lu, “Understanding the idiosyncrasies of real persistent memory,” *Proc. VLDB Endow.*, vol. 14, no. 4, p. 626–639, dec 2020. [Online]. Available: https://doi.org/10.14778/3436905.3436921

[8] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, “Stamp: Stanford transactional applications for multi-processing,” in *2008 IEEE International Symposium on Workload Characterization*, 2008, pp. 35–46.

[9] Transaction Processing Performance Council, "TPC-C Benchmark Revision 5.11.0," http://tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf, Last accessed 21 May 2021.

[10] Intel, "Affordably Accommodate the Next Wave of Data Demands," https://www.intel.com/content/dam/www/public/us/en/documents/brief/affordably-accommodate-the-next-wave-of-data-demands.pdf, Last accessed 21 May 2021.

[11] ——, "Achieve Greater Insight from Your Data with Intel Optane Persistent Memory," https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-persistent-memory-200-series-brief.pdf, Last accessed 21 May 2021.

[12] R. Ramakrishnan and J. Gehrke, *Database Management Systems*, 4th ed. McGraw-Hill Education - Europe, 2002.

[13] C. J. Rossbach, O. S. Hofmann, and E. Witchel, "Is transactional programming actually easier?" in *PPoPP '10: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, January 2010, pp. 47–56.

[14] Nuno Diegues and Paolo Romano and Luís Rodigues, "Virtues and Limitations of Commodity Hardware Transactional Memory," in *PACT '14: Proceedings of the 23rd international conference on Parallel architectures and compilation*, August 2017, pp. 3–14.

[15] R. Guerraoui and M. Kapałka, "Opacity: A Correctness Condition for Transactional Memory," 2007.

[16] D. Imbs and M. Raynal, "Virtual world consistency: A condition for STM systems (with a versatile protocol with invisible read operations)," *Theoretical Computer Science*, vol. 444, pp. 113–127, July 2012.

[17] P. Felber, C. Fetzer, and T. Riegel, "Dynamic performance tuning of word-based software transactional memory," in *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, February 2008, pp. 237–246.

[18] P. Felber, C. Fetzer, P. Marlier, and T. Riegel, "Time-Based Software Transactional Memory," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 12, pp. 1793–1807, 2010.

[19] Intel, "Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4," https://software.intel.com/content/dam/develop/external/us/en/documents-tps/325462-sdm-vol-1-2abcd-3abcd.pdf, Last accessed 21 May 2021.

[20] B. Hall, P. Bergner, A. S. Housfater, M. Kandasamy, T. Magno, A. Mericas, S. Munroe, M. Oliveira, B. Schmidt, W. Schmidt, B. K. Smith, J. Wang, S. Warrier, and D. Wendt, "Performance Optimization

and Tuning Techniques for IBM Power Systems Processors Including IBM POWER8," http://www.redbooks.ibm.com/redbooks/pdfs/sg248171.pdf, Last accessed 21 May 2021.

[21] C. S. Ananian, K. Asanović, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded Transactional Memory," in *11th International Symposium on High-Performance Computer Architecture*, 2005, pp. 316–327.

[22] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "LogTM: Log-based Transactional Memory," in *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.*, 2006, pp. 254–265.

[23] R. Rajwar, M. Herlihy, and K. Lai, "Virtualizing Transactional Memory," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 2, pp. 494–505, 2005.

[24] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum, "Hybrid transactional memory," in *ASPLOS XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, October 2006, pp. 336–346.

[25] A. Baldassin, J. a. Barreto, D. Castro, and P. Romano, "Persistent memory: A survey of programming support and implementations," *ACM Comput. Surv.*, vol. 54, no. 7, jul 2021. [Online]. Available: https://doi.org/10.1145/3465402

[26] S. Chen and Q. Jin, "Persistent b+-trees in non-volatile main memory," *Proc. VLDB Endow.*, vol. 8, no. 7, p. 786–797, feb 2015. [Online]. Available: https://doi.org/10.14778/2752939.2752947

[27] M. Liu, J. Xing, K. Chen, and Y. Wu, "Building scalable nvm-based b+tree with htm," in *Proceedings of the 48th International Conference on Parallel Processing*, ser. ICPP 2019. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3337821.3337827

[28] M. Nam, H. Cha, Y. ri Choi, S. H. Noh, and B. Nam, "Write-Optimized dynamic hashing for persistent memory," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 31–44. [Online]. Available: https://www.usenix.org/conference/fast19/presentation/nam

[29] F. Nawab, J. Izraelevitz, T. Kelly, C. B. M. III, D. R. Chakrabarti, and M. L. Scott, "Dalí: A Periodically Persistent Hash Map," in *31st International Symposium on Distributed Computing (DISC 2017)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), A. W. Richa, Ed., vol. 91. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, pp. 37:1–37:16. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2017/8014

[30] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, “Atlas: Leveraging locks for non-volatile memory consistency,” *SIGPLAN Not.*, vol. 49, no. 10, p. 433–452, oct 2014. [Online]. Available: https://doi.org/10.1145/2714064.2660224

[31] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison, “An Approach to Persistent Programming,” *The Computer Journal*, vol. 26, no. 4, pp. 360–365, 11 1983. [Online]. Available: https://doi.org/10.1093/comjnl/26.4.360

[32] J. Guerra, L. Marmol, D. Campello, C. Crespo, R. Rangaswami, and J. Wei, “Software persistent memory,” in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, Jun. 2012, pp. 319–331. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/guerra

[33] T. Kelly, “Persistent memory programming on conventional hardware: The persistent memory style of programming can dramatically simplify application software.” *Queue*, vol. 17, no. 4, p. 1–20, aug 2019. [Online]. Available: https://doi.org/10.1145/3358955.3358957

[34] L. Marmol, M. Chowdhury, and R. Rangaswami, “Libpm: Simplifying application usage of persistent memory,” *ACM Trans. Storage*, vol. 14, no. 4, dec 2018. [Online]. Available: https://doi.org/10.1145/3278141

[35] S. Park, T. Kelly, and K. Shen, “Failure-atomic msync(): A simple and efficient mechanism for preserving the integrity of durable data,” in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys ’13. New York, NY, USA: Association for Computing Machinery, 2013, p. 225–238. [Online]. Available: https://doi.org/10.1145/2465351.2465374

[36] H.-J. Boehm and D. R. Chakrabarti, “Persistence programming models for non-volatile memory,” *SIGPLAN Not.*, vol. 51, no. 11, p. 55–67, jun 2016. [Online]. Available: https://doi.org/10.1145/3241624.2926704

[37] V. Gogte, S. Diestelhorst, W. Wang, S. Narayanasamy, P. M. Chen, and T. F. Wenisch, “Persistency for synchronization-free regions,” *SIGPLAN Not.*, vol. 53, no. 4, p. 46–61, jun 2018. [Online]. Available: https://doi.org/10.1145/3296979.3192367

[38] A. Kolli, V. Gogte, A. Saidi, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, “Language-level persistency,” *SIGARCH Comput. Archit. News*, vol. 45, no. 2, p. 481–493, jun 2017. [Online]. Available: https://doi.org/10.1145/3140659.3080229

[39] S. Scargall, *Programming Persistent Memory: A Comprehensive Guide for Developers*, 1st ed. Apress, 2020.

[40] H. Volos, A. J. Tack, and M. M. Swift, “Mnemosyne: Lightweight Persistent Memory,” in *ASPLOS XVI: Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, March 2011, pp. 91–104.

[41] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, “NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories,” in *ASPLOS XVI: Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, March 2011, pp. 105–118.

[42] R. M. Krishnan, J. Kim, A. Mathew, X. wei Fu, A. Demeri, C. Min, and S. sun Kannan, “Durable transactional memory can scale with TimeStone,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 335–349.

[43] Storage Networking Industry Association (SNIA) Technical Position, “NVM Programming Model Version 1.2,” jun 2017.

[44] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Rivière, “Evaluation of amd’s advanced synchronization facility within a complete transactional memory stack,” in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys ’10. New York, NY, USA: Association for Computing Machinery, 2010, p. 27–40. [Online]. Available: https://doi.org/10.1145/1755913.1755918

[45] N. Diegues and P. Romano, “Self-tuning intel restricted transactional memory,” *Parallel Computing*, vol. 50, pp. 25–52, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819115001209

[46] D. Didona, P. Felber, D. Harmanci, P. Romano, and J. Schenker, “Identifying the optimal level of parallelism in transactional memory applications,” *Computing*, vol. 97, no. 9, p. 939–959, sep 2015. [Online]. Available: https://doi.org/10.1007/s00607-013-0376-3