UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

PERFORMANCE CHARACTERIZATION OF PAGE FAULT HANDLING IN MODERN PERSISTENT MEMORY SYSTEMS

ANDRÉ LIBÓRIO DE BARROS FERRAZ

Instituto de Geociências e Ciências Exatas
Rio Claro - SP
2025

UNIVERSIDADE ESTADUAL PAULISTA
"Júlio de Mesquita Filho"
Instituto de Geociências e Ciências Exatas
Câmpus de Rio Claro

ANDRÉ LIBÓRIO DE BARROS FERRAZ

PERFORMANCE CHARACTERIZATION OF PAGE FAULT HANDLING IN MODERN PERSISTENT MEMORY SYSTEMS

Dissertação de Mestrado apresentada ao Instituto de Geociências e Ciências Exatas do Câmpus de Rio Claro, da Universidade Estadual Paulista “Júlio de Mesquita Filho”, como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Alexandro José Baldassin

Rio Claro - SP
2025

F381p

Ferraz, André Libório de Barros
Performance characterization of page fault handling in modern persistent memory systems / André Libório de Barros Ferraz. -- Rio Claro, 2025
58 p. : il., tabs., fotos

Dissertação (mestrado) - Universidade Estadual Paulista (UNESP), Instituto de Geociências e Ciências Exatas, Rio Claro
Orientador: Alexandro José Baldassin

1. Ciência da computação. 2. Computação de alto desempenho. 3. Memória virtual. I. Título.

Sistema de geração automática de fichas catalográficas da Unesp. Dados fornecidos pelo autor(a).

POTENTIAL IMPACT OF THIS RESEARCH

This research aims to make a significant technical impact by establishing and proposing new approaches to technologies still in their infancy, thus contributing to technological innovation and, consequently, the pursuit of more performant and efficient systems.

IMPACTO POTENCIAL DESTA PESQUISA

Esta pesquisa busca trazer um impacto técnico significativo, por meio da fundamentação e proposta de novas abordagens para tecnologias ainda em sua infância. Colaborando assim, com a inovação tecnologica e consequentemente a busca por sistemas mais performantes e eficientes.

UNIVERSIDADE ESTADUAL PAULISTA
"Júlio de Mesquita Filho"
Instituto de Geociências e Ciências Exatas
Câmpus de Rio Claro

ANDRÉ LIBÓRIO DE BARROS FERRAZ

PERFORMANCE CHARACTERIZATION OF PAGE FAULT HANDLING IN MODERN PERSISTENT MEMORY SYSTEMS

Dissertação de Mestrado apresentada ao Instituto de Geociências e Ciências Exatas do Câmpus de Rio Claro, da Universidade Estadual Paulista “Júlio de Mesquita Filho”, como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação.

Comissão Examinadora

Prof. Dr. Alexandro José Baldassin
IGCE / UNESP/Rio Claro (SP)

Prof. Dr. Emilio Francesquini
CMCC / UFABC/Santo André (SP)

Prof. Dr. Orlando de Andrade Figueredo
IGCE / UNESP/Rio Claro (SP)

Conceito: Aprovado.

Rio Claro (SP), 07 de Julho de 2025.

I dedicate this thesis to my family and all those who helped me to grow, as a student, and especially, as a person.

ACKNOWLEDGEMENTS

First, I warmly thank my supervisor, Alexandro José Baldassin, for all the knowledge and opportunities he shared with me over the last few years that culminated in this work.

I also thank the effort of our colleagues in Lisbon, João Pedro Barreto, Paolo Romano, and Daniel Castro, for providing countless contributions to this work and being exceptional hosts during my time in Portugal.

This study was financed, in part, by the São Paulo Research Foundation (FAPESP), Brazil. Process Number 2022/11704-8. The opinions, hypotheses, conclusions, or recommendations expressed in this material are the responsibility of the authors and do not necessarily reflect the views of FAPESP.

A big thank you to my family for all the support, including those who unfortunately are no longer with us. And finally, but not less, I would like to thank all my friends and colleagues, old and new, who helped me be who I am today and, therefore, able to achieve this goal in my academic life; thank you.

Quand l'un tombe, on continue.

RESUMO

Memória persistente (PM) é uma tecnologia de memória não volátil, endereçável por bytes, que apresenta bom desempenho e tempos de resposta rápidos. No entanto, sua persistência apresenta muitos desafios adicionais ao software, onde as transações têm sido usadas como um mecanismo essencial.

Soluções de última geração empregam transações de hardware para gerenciar PM. Essas soluções geralmente utilizam DRAM como cache de PM, com uma abordagem que melhora a escalabilidade geral das transações e da recuperação em comparação com tentativas anteriores. No entanto, a maioria dos sistemas atuais não oferece suporte para casos em que a PM possui capacidades maiores que a DRAM. De fato, a maioria dos sistemas propostos não sequer analisa os impactos no desempenho de ter espaço de DRAM insuficiente.

Portanto, este estudo apresenta uma análise detalhada de sistemas de PM sob limitações de DRAM, bem como propõe uma estrutura de paginação flexível que não é restringida por limitações de hardware e kernel.

Os experimentos mostram que nossa estrutura de paginação apresenta melhorias significativas de desempenho em relação à troca em cenários de DRAM limitada, com melhorias de throughput de até 152% nas configurações testadas.

Palavras-chave: Memória persistente. Transações. Paginação.

ABSTRACT

Persistent memory (PM) is a non-volatile, byte-addressable, memory technology that has good performance and fast response times. However, its persistence presents many additional challenges to software, where transactions have been used as an essential mechanism.

State-of-the-art solutions employ hardware transactions to manage PM. These solutions usually make use of DRAM as a PM cache with an approach that improves overall transactions and recovery scalability compared to previous attempts. However, most current systems lack support for cases where PM has larger capacities than DRAM. In fact, most of the proposed systems do not even analyze the performance impacts of having insufficient DRAM space.

Therefore, this study presents a detailed analysis of PM systems under DRAM limitations as well as proposes a flexible paging framework that is not constrained by hardware and kernel limitations.

The experiments show that our paging framework has a significant performance improvements over swap in limited DRAM scenarios, with throughput improvements of up to 152% in tested configurations.

Keywords: Persistent memory. Transactions. Paging.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

ADR	Asynchronous DRAM Refresh
CoW	Copy-on-Write
DCPMM	Intel Optane DC Persistent Memory
DIMM	Dual In-line Memory Module
DRAM	Dynamic Random Access Memory
eADR	Extended Asynchronous DRAM Refresh
FASE	Failure-Atomic SEction
HTM	Hardware Transactional Memory
LRU	Least Recently Used
NVDIMM	Non-Volatile Dual In-line Memory Module
NVM	Non-Volatile Memory
PD	Persistence Domain
PM	Persistent Memory
SGL	Single Global Lock
SP	Shadow Paging
SPHT	Scalable Persistent Hardware Transactions
STM	Software Transactional Memory
WAL	Write-Ahead Logging

CONTENTS

1 INTRODUCTION

Non-Volatile Memory (NVM) was first introduced with core memory in the 1950s (ROSNER et al., 2018). This technology enabled the production and commercialization of low-cost stored-program computers. A notable implementation of this concept in the early days was seen in the Atlas computer, developed at the University of Manchester in 1962 (LAVINGTON, 2012). It offered a unique perspective where an address could refer to either the data stored in the main memory or persistent memory.

In its current state, Persistent Memory (PM) has several advantages over traditional DRAM, offering higher capacities, reduced costs, byte-addressable persistence, and low-latency durability (INTEL, 2022). The increased capacities at reduced costs are of particular interest to companies. This implies that software could leverage these larger memory sizes and depend less on memory paging. In a potential future where PM becomes mainstream, it may even replace main memory and traditional storage altogether.

Additional benefits of PM include byte-addressable persistence and low-latency durability. Byte-addressable persistence allows data granularity to be determined solely by cache line sizes, rather than the typical block sizes. Moreover, the inherent low latency can help achieve performance comparable to that of DRAM, all while simplifying data recovery, since a straightforward data flush from the processor can ensure data persistence. These attributes have the potential to improve power savings in systems, which is particularly beneficial for mobile devices due to the practical merging of sleep and hibernation modes (KATSARAGAKIS et al., 2022).

Although there are other competing memory technologies, PM has recently stood out due to Intel Optane DC PM (DCPMM), first released in 2019 in a partnership with Micron. It achieved good performance while presenting much denser DIMMs (here referred to as NVDIMMs) compared to the traditional DRAM available in the server market (INTEL, 2022). However, utilizing these devices is not straightforward when trying to extract their true potential due to numerous software challenges that can be presented. It should also be noted that although Intel has recently discontinued Optane DC devices, the industry has embraced the new Compute Express Link (CXL) standard (KWON et al., 2023), which also supports byte-addressable persistent memory.

When using a PM module, the initial decision revolves around its intended use: as main memory, replacing DRAM; as a main memory extension alongside DRAM; utilizing DRAM as PM cache; or using DRAM and PM separately at the same hierarchy level, which is discussed in this document. Although PM provides durability, the entire system does not. This is a result of data going through a volatile CPU cache, which makes the software

need extra care. This durability can be achieved by increasing the program's complexity and requiring a support mechanism from the programming language itself. Atomicity is another important aspect of PM systems. In the event of a failure, durability can only be guaranteed by completing the entire set of related operations. In this context, transactions are commonly employed to ensure atomicity.

1.1 OBJECTIVES

This is notably important for contemporary transactional PM systems, where DRAM is used as shadow memory to hold volatile copies of the updates processed by the transactions. Shadow memory in particular is the memory space used by a technique called Shadow Paging (SP), which addresses out-of-place updates. Here, the updates modify a private copy (shadow copy), not the PM itself, which happens only after the transaction finishes. This is the main method used by state-of-the-art transaction-based PM systems, since hardware transactions cannot change persistent data without aborting. Therefore, they operate on shadow memory and log their changes using the redo log, which are then committed to PM after the transaction commits.

Since DRAM is used for shadow copy, an issue related to the DRAM capacity arises: What if the DRAM size is much smaller than PM? Once there is no available space on DRAM, a typical operating system will start to evict and load pages (swapping), a scenario that could rapidly evolve to thrashing (where most of the time is spent paging) and therefore causing a significant impact on performance.

The main goal of this dissertation is to provide a thorough evaluation of the behavior of PM systems under DRAM limitations. As pointed out earlier, current systems assume that DRAM will never be an issue, and therefore little is known about the performance of the systems under this constraint. Our experimental results show that a standard swap implementation has significant impact on performance under limited DRAM situations; in the 2 workloads tested here, we noted a slowdown of 44.0x and 7.2x average across all thread configurations (1-24) with half the required DRAM capacity for each workload.

Additionally, we contribute to this analysis by presenting a user-level technique to simulate page handling based on intercepting OS page faults. We found that current frameworks either relied on specific kernel versions (BELAY et al., 2012a) or simulated the entire paging mechanism (LIU et al., 2017). Experiments showed that our paging framework had significant improvements over swap in memory-constrained scenarios, with our worst performing implementation improving upon the average 44.0x and 7.2x slowdowns presented by swap to 13.1x and 3.0x, respectively.

The work presented here has been presented and published in paper form at SSCAD

2024 (LIBÓRIO et al., 2024). Due to its relevance in the event, an extended version will be submitted for publication at CCPE in the SSCAD 2024 special issue.

The document is organized in the following way. Chapter 2 presents an overview of Persistent Memory and its current software implementations, involving data persistence techniques, as well as an overview of the structure used by current Persistent Memory systems, including their usual lack of support for DRAM-constrained scenarios. Chapter 3 presents our proposal for a paging framework, which tackles limited DRAM scenarios, containing its architectural details as well as considerations when integrating it to a state-of-the-art PM system, used for the experimental section. Chapter 4 corresponds to the experimental section, first presenting the benchmark and the configurations used, presenting the system, and a performance characterization of both swap and our paging framework implementations in memory-constrained scenarios. Finally, Chapter 5 presents a summary of this research and also points to relevant future research topics.

2 THEORETICAL FRAMEWORK

This chapter presents some relevant concepts used in this work. Section 2.1 presents an overview of PM technology and recent implementations and use cases, while Section 2.2 introduces transactions, a widespread technology used in PM systems, and their uses to achieve data consistency. Finally, Section 2.3 presents in some detail the inner workings of current PM systems, in particular the ones that leverage transactions and Shadow Paging, which are of utmost relevance. Finally, we discuss available solutions to the problem we are addressing.

2.1 PERSISTENT MEMORY

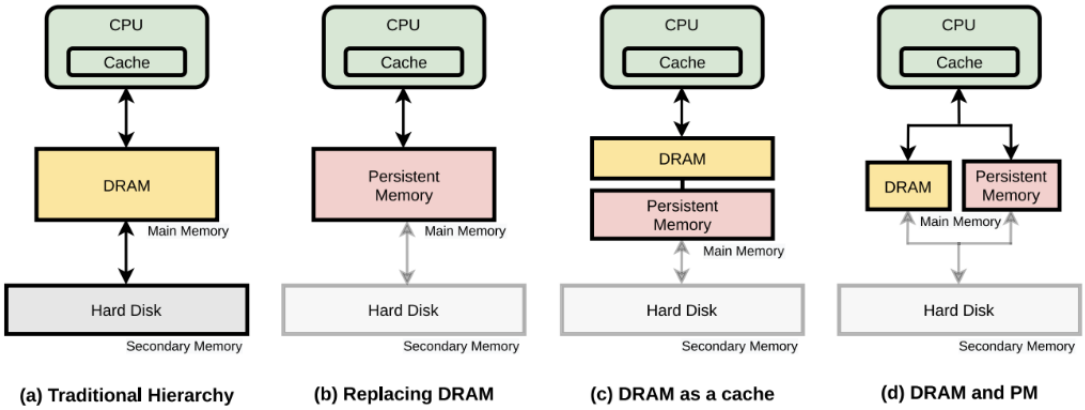
Persistent Memory (PM) stands as a byte-addressable memory technology characterized by its non-volatile and enduring nature. Similarly to SSDs and HDDs, PM can preserve the data when powered off, while having performance comparable to DRAM but at lower power consumption (PATIL et al., 2019).

Recent advances make use of CXL 3.0 (Compute Express Link) to expand persistence beyond CPUs, including communication with expansion cards such as GPUs (KWON et al., 2023). However, since CXL is commercially in its early stages, this work has been developed using the now discontinued Intel Optane DC Persistent Memory (DCPMM). This memory is supported by recent Intel server platforms via DIMM interface, and which, due to interfacing similarities, will be able to share the same code with the new CXL modules, future proofing this study (FRIDMAN et al., 2023). Here, this particular product will be referred to as NVDIMM.

Despite the potential of NVDIMMs to supplant existing DRAM DIMMs and conventional storage media such as SSDs and HDDs due to their large capacities (128GB-512GB per DIMM), the software side still has challenges in order to make this a viable implementation without compromising performance. Figure 1 illustrates the three main uses of PM in current systems: i) to use PM as main memory; ii) DRAM as a cache; iii) as a hybrid implementation, with PM and DRAM operating cooperatively at the same hierarchy level. The latter strategy involves loading volatile data onto DRAM and non-volatile onto PM. This eliminates the need for persistent data to be written back to an alternative non-volatile medium, thus enhancing overall system performance.

While the potential benefits of using NVDIMMs such as DCPMM are undoubtedly promising, programming them requires careful attention. One of the biggest concerns while adopting a non-volatile memory technology is the fact that the other memory structures

Figure 1 – Different memory hierarchy organizations: (a) a traditional two-level memory system, with both main (volatile) and secondary (block-based, non-volatile) memories; (b) PM replacing DRAM as main memory; (c) vertical integration: DRAM working as a cache to PM; and (d) horizontal integration: data can be placed in either DRAM or PM. Configurations (b), (c), and (d) could further provide a block-based backstore.



Source: (BALDASSIN et al., 2021).

inside the system are still volatile. In particular, the main memory (DRAM) and internal CPU caches (SRAM) are volatile and therefore susceptible to data loss.

In this context, the Persistence Domain (PD), often referred to as power-fail protected domain, comprehends a region of memory in which the computer system can ensure that the content will be preserved (GOGTE; KOLLI; WENISCH, 2022). This is an important part of the PM implementation, because the larger the domain, the simpler the supporting software. For instance, having persistent caches frees the software from having to add flush instructions to force data eviction from the caches.

Intel has created two main PD domains: i) Asynchronous DRAM Refresh (ADR) (RUDOFF, 2016) and; ii) enhanced ADR (eADR) (SCARGALL, 2020). The introduction of ADR marked a significant turning point in the expansion of PD. A notable outcome of ADR's implementation was the persistence of the Writing Pending Queue (WPQ) within the memory controller, even in the event of a power failure. This implies that the residual power supply could flush the WPQ when needed (BALDASSIN et al., 2021). Subsequently, eADR took this concept a step further, making also the processor caches persistent. However, it presented the caveat that the system's residual power is not enough to ensure proper operation, requiring an external power source, such as an external power supply or battery, which should increase costs and complexity.

Another alternative is to implement what has been called Whole System Persistence (WSP), which extends the Persistence Domain to encompass processor registers as well (NARAYANAN; HODSON, 2012). Using WSP assumes that only NVDIMMs are being used and the system relies on residual energy to flush data to PM in the case of a failure.

Table 1 presents a comparison between various memory technologies, including PCM (Phase-Change Memory), which serves as the foundation for DCPMM. The data presented show that PCM is vastly superior to NAND Flash, especially in read latency, where it is much closer to DRAM.

Although DRAM has an overall higher throughput compared to current PCM, endurance and read latency are on the same order of magnitude, so it is plausible to consider that a newer implementation like DCPMM can be even more competitive with DRAM while maintaining data persistence.

Table 1 – Access latency and endurance (in number of overwrites) of current DRAM, PCM and Flash memories. Prospective characteristics are based on demonstrated prototypes.

Technology	Read	Write	Endurance
DRAM	0.06 μs	0.006 μs	$> 10^{16}$
NAND Flash	25 μs	200 - 500 μs	10^4 - 100^5
PCM	0.115 μs	120 μs	10^6

Source:(VOLOS; TACK; SWIFT, 2011).

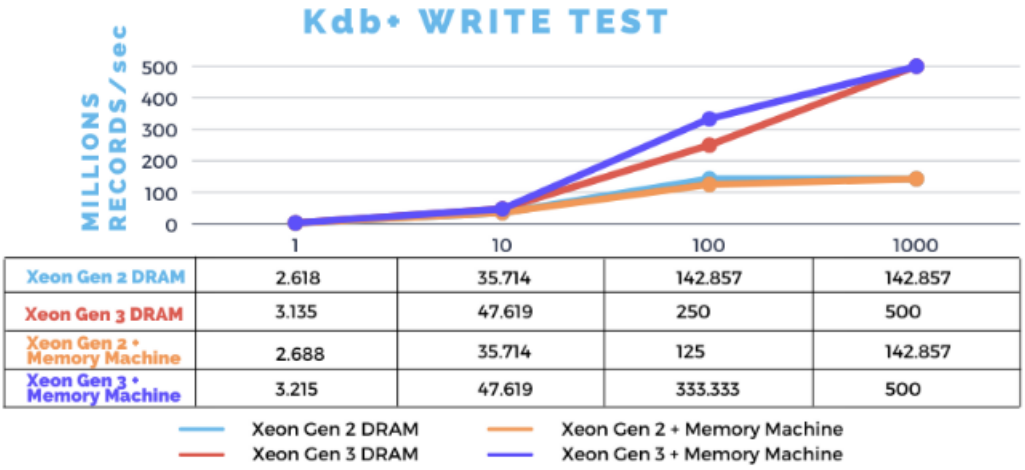
When considering DCPMM's utilization of both DDR4 and the newer DDR5 versions, the performance disparity compared to DRAM has been noticeably reduced compared to previous endeavors. Although certain workloads clearly benefit from the persistence offered by these devices, others do not present competitive performance compared to DRAM-only equipped systems.

Figure 2 provides information on various performance metrics, specifically within database scenarios, showcasing the capabilities of DCPMM on DDR4 platforms. The data highlight substantial performance improvements made by the use of PM in these systems. The Xeon Gen 2 system is equipped with 2x Intel Xeon Platinum 8270 (56c/112t), 192GB RAM DDR4, 1.5TB Intel Persistent Memory 100 Series (PMem), 1TB Sata SSD and CentOS 8.2.2004, while the Xeon Gen 3 system is equipped with 2x Intel Xeon Platinum 8380 (80c/160t), 512GB RAM DDR4, 2TB Intel Persistent Memory 200 Series (PMem), 1TB Sata SSD and CentOS 8.3.2011.

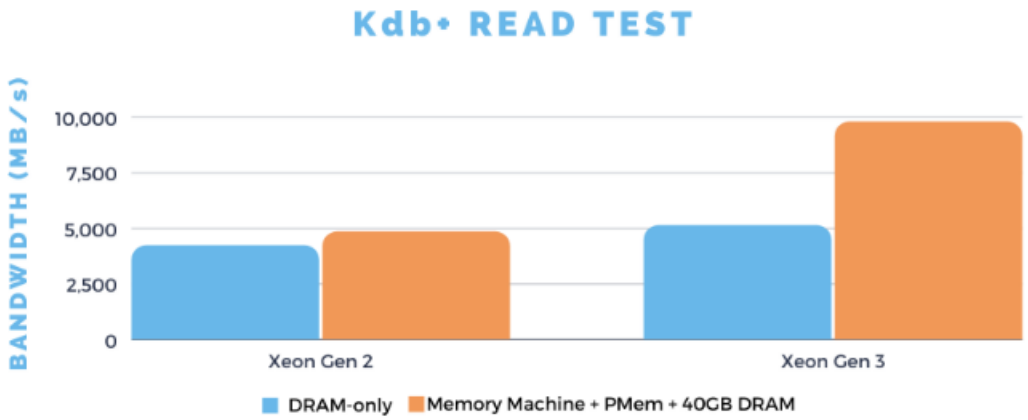
2.2 TRANSACTIONS

PM hardware is advanced enough to be considered as an alternative memory technology, but several challenges still persist when developing PM-based software. One of those is to achieve proper data persistence, meaning failure mechanisms need to be taken into account to ensure proper durability in the case of a system failure, and transactions are a prevalent approach to deal with this problem.

Figure 2 – Read and write benchmark of 1st and 2nd generation of Intel Optane PM DC, comparing DRAM performance with DRAM+DCPMM in each platform in a database workload using KX's kdb+.



Xeon Gen 2 DRAM	2.618	35.714	142.857	142.857
Xeon Gen 3 DRAM	3.135	47.619	250	500
Xeon Gen 2 + Memory Machine	2.688	35.714	125	142.857
Xeon Gen 3 + Memory Machine	3.215	47.619	333.333	500



Source:(BEELER, 2021).

A transaction comprises a series of instructions that, from an external process's perspective, seem indivisible and instantaneous. In practical terms, this implies that a transaction is either fully executed or completely aborted, reverting all the performed changes to a previous state. This concept is exemplified in Figure 2, which uses a bank transaction as an illustrative case.

In this scenario, the operations must be executed inside a transaction, because the decrement of the savings account and the subsequent increment of the checking account as well as the recording-keeping routines need to be set as an atomic operation. This concept of atomicity is pivotal, acting as a shield against possible failures that, in practical scenarios, could lead to discrepancies or even financial losses.

The usage of this technology can often be seen as related only to database systems, in order to ensure the ACID properties. But with the advent of parallel systems, the treatment of critical regions started to be even more important since it has a great impact on the

Table 2 – Simplified example of the inner workings of a transaction in the context of a banking transaction using database and hardware transactions.

Status	Database	Transaction
Start Transaction	BEGIN TRANSACTION	atomic {

Decrement Savings Account	UPDATE savings_account
	SET balance = balance - 500
	WHERE account = 3209
		int value1 = bank.get(3209);
		value1 -= 500
		bank.put(3209, value1);

Incrementing Checking Account	UPDATE checking account
	SET balance = balance + 500
	WHERE account = 3209
		int value2 = bank.get(3208);
		value2 += 500
		bank.put(3208, value2);

End Transaction	COMMIT	}

Source: Elaborated by the author.

overall performance of the system. The use of transactions as an abstraction for parallel programming first appeared in 1993 and the term Transactional Memory (TM) was coined by Herlihy and Moss (HERLIHY; MOSS, 1993).

A notable distinction between transactions employed in a TM system and those in conventional database systems lies in their interaction with data. In the context of a TM system, transactions operate on volatile data, which implies that the system typically does not enforce the durability property.

In this context, there are two main approaches to implementing TM: the software-based approach, commonly referred to as Software TM (STM); and the hardware-based approach, known as Hardware TM (HTM).

STM presents itself as a versatile option for most implementations, given its flexibility and compatibility with a wide range of systems. However, it often falls short in terms of performance when compared to HTM systems. The challenge with HTM lies in its dependency on processor support. Moreover, transactions executed within an HTM framework are approached as best-effort endeavors. In essence, the hardware does not guarantee that the hardware transaction will ever commit, which makes the software component extremely relevant.

Most current x86-based HTM implementations utilize Intel Transactional Synchronization Extensions (TSX), an extension to the x86 instruction set designed to provide transaction support (RAJWAR; DIXON, 2012). It should be noted that hardware transactions are not exclusive to x86; some IBM processors also provide support for this feature (JACOBI; SLEGEL; GREINER, 2012).

TM differs from database implementations in some key ways. Notably, in a database setup, data is conventionally stored on disks, while transactional memory operates by

accessing the faster main memory. As such, it leaves minimal computation time during the memory access interval, making it one of the justifications to favor HTM in these scenarios (HERLIHY; MOSS, 1993), (HALL et al., 2017).

Furthermore, in these systems, the data do not need to persist beyond the process execution, whereas, in a database system, it needs to be written back to the disk when the transaction is committed.

A typical transactional programming interface is shown in Figure 3. This kind of interface is supported by both STM and HTM and it is very intuitive. Using Figure 3 as an example, here *StartTx* starts the transaction, *CommitTx* attempts to commit the transaction; if it returns *true*, it was successful; otherwise the transaction is aborted. *AbortTx* is another call that always aborts the current transaction. Some low-level interfaces also require that the reads and writes to memory be versioned so that old and new data can be correctly managed.

Table 3 – Comparative example of concurrency control mechanisms.

Interface	Description
StartTx	Starts the transaction
CommitTx	Commits the transaction
AbortTx	Aborts the transaction

Source: Elaborated by the author.

Developing a TM interface requires careful recognition that, while the interface may appear straightforward, there exist a multitude of potential implementations. This is particularly evident when considering aspects such as concurrency control mechanisms, version management, and conflict control (HARRIS; LARUS; RAJWAR, 2010).

Concurrency control is one of the most important aspects, since it affects the synchronization that mediates concurrent access to data. To delve deeper into this concept, it is beneficial to explore the intricacies of what a conflict means in the context of program execution. A conflict occurs every time two transactions perform operations with the same data, either two concurrent writes or a write from one and a read from another.

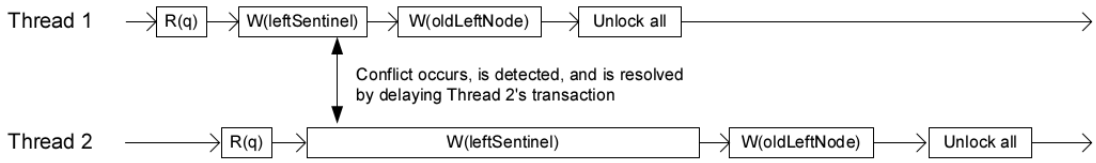
2.2.1 CONCURRENCY CONTROL

There are two main categories of concurrency control: pessimistic and optimistic. In a pessimistic approach, conflict resolution occurs as soon as a conflict is detected. This allows a transaction to prevent others from accessing the data or claiming exclusivity. In particular, in scenarios characterized by high conflict rates, this approach can significantly enhance

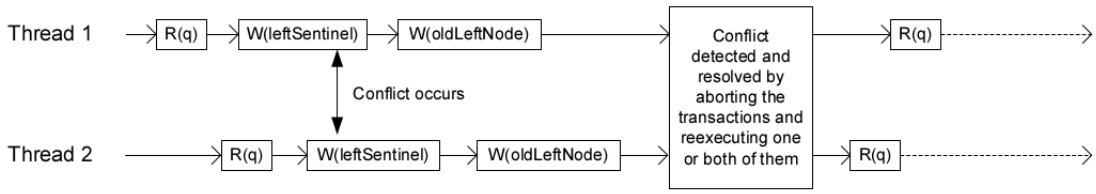
performance. However, it is essential to acknowledge that this mechanism can potentially lead to deadlocks. To mitigate this, the implementation of timeouts or dynamic deadlock detection techniques becomes crucial. Optimistic concurrency control allows conflict detection and resolution to happen after the conflict occurs. The objective is to resolve all conflicts before a transaction is committed. If a conflict is identified, the system will check its cause and resolve it. This approach is more suitable for systems where the conflict rates are rather small.

A more practical example of both can be observed in Figure 3, where two threads concur for writes in a single variable, and the respective conflict-solving mechanism is applied. In Figure 3a, a pessimistic setting is presented. Here, when a conflict occurs, thread 2's transaction is delayed until thread 1's transaction is finished. Figure 3b, on the other hand, shows an optimistic approach. In this instance, both transactions are executed simultaneously, and if a conflict is detected in the end, it is resolved, and one or both transactions are re-executed.

Figure 3 – Comparative example of concurrency control mechanisms.



(a) Pessimistic concurrency control.



(b) Optimistic concurrency control.

Source:(HARRIS; LARUS; RAJWAR, 2010).

2.2.2 VERSION MANAGEMENT

It aims to manage the tentative writes from concurrent transactions. The two main approaches are eager version management and lazy version management.

In eager version management, transactions directly modify data in memory, which means that it requires pessimistic concurrency control. This works by keeping an undo log for the overwritten values, which are written back in the case of a transaction abort.

On the other hand, lazy version management requires the updates to be delayed until the transaction commits. It also uses a log but, in this case, a redo log that is transaction-

private and contains its tentative writes, used by read instructions to consult logs that check for write updates. Upon committing, the transaction updates the locations from private copies; if it aborts, the redo log is simply discarded.

2.2.3 CONFLICT DETECTION

The final component is the conflict detection mechanism that diverges significantly based on the concurrency control approach used. Under pessimistic concurrency control, it is much simpler, as a lock can only be acquired when it is not conflicting with another thread. However, with optimistic concurrency control, there are three parameters that can determine the implementation used regarding conflict detection: granularity, time, and the kind of access that prompted it.

TM has been widely developed for use in parallel programming with volatile memory scenarios. More recently, it has also been utilized by new PM programming systems. In particular, a great challenge is how to make use of PM with HTM, since current HTM implementations do not support durability.

2.2.4 LOGGING MECHANISMS

In the event of system failure, a transactional system requires a logging mechanism to effectively manage versioning aspects. In the literature, the most well-studied approach is known as Write-Ahead Logging (WAL) (MOHAN; LEVINE, 1992), which requires two versions of the object being accessed and comes with two flavors, undo and redo logging, as elaborated earlier. Additional techniques employed in this context are Shadow Paging (SP) and Copy-on-Write (CoW) (CASTRO; ROMANO; BARRETO, 2019), (LIU et al., 2017). These mechanisms assume a pivotal role in the realm of persistent transactions and consequently merit further discussion within this section.

WAL implementations tailored for PM share many similarities with those utilized in database systems, providing failure atomicity and durability. In these implementations, updates are initially recorded and persisted in a log before being written to a specific PM memory address.

WAL mechanisms are divided into two main categories: undo and redo logging. In the context of undo logging, the original data is stored in the log before being updated directly in PM; should a failure arise before persistence is ensured, the changes are simply rolled back to the previous state. This approach has found application in various PM systems, such as Atlas (CHAKRABARTI; BOEHM; BHANDARI, 2014), NV-Heaps (COBURN et al., 2011) and SFR (GOGTE et al., 2018).

In contrast, redo logging uses a replay mechanism. It aims to write the updated version to the log, ensuring the log's persistence prior to applying the PM update. In the event of a failure occurring before the persistence of the redo log is confirmed, any new data is discarded, not requiring any changes to the PM data. If a failure arises during the replay, the redo log is replayed only upon recovery.

Despite the choice of the logging mechanism, both add overhead to the system and have the setbacks of wearing out the PM modules faster due to the increase in writes needed for each file to the device, a phenomenon called write amplification (WOO et al., 2021). These mechanisms also impact bandwidth, and consequently latency ((HU et al., 2017), (CORREIA; FELBER; RAMALHETE, 2018), (WU et al., 2020)).

Shadow Paging (SP) and Copy-on-Write (CoW) represent distinct approaches that address out-of-place updates without having the "write-twice problem" which is a concern inherent in logging mechanisms where both the new and current versions of data need to be written.

In the case of Shadow Paging, the updates do not directly modify the PM itself. Instead, a private copy of the current data is created, serving as a temporary place holder for the updates. This is an interesting approach, as it can be modified without any persistence concerns. When an update is about to finish, the original content is replaced by its updated copies. If a failure occurs before it finishes, only reclaiming the memory space allocated for the copy is required.

Copy-on-Write is a more restrictive version of SP. Both work with the same principles, but here, in favor of efficiency, only when a page is modified, a private copy is created, preventing other changes. However, both SP and CoW come with their own set of challenges. While they mitigate the write-twice issue present in WAL, they introduce a copy overhead, subsequently contributing to write amplification.

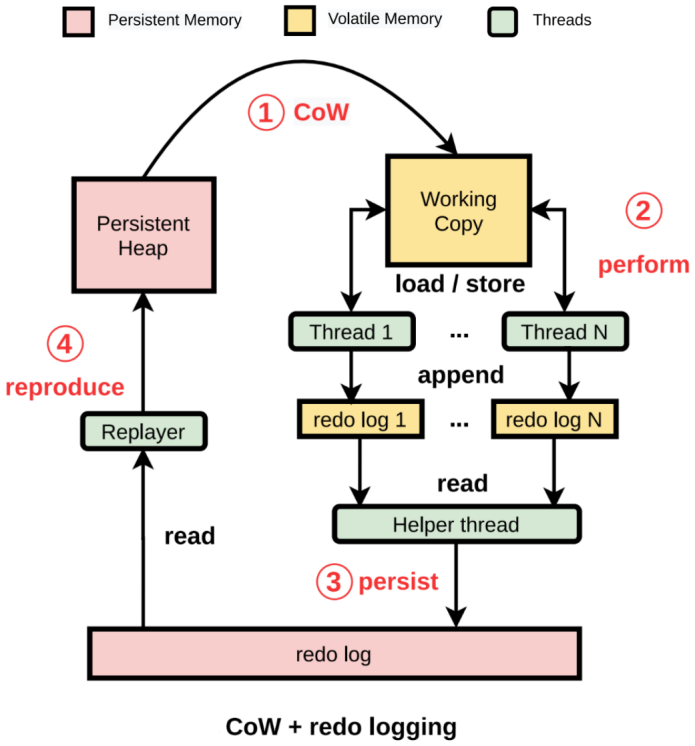
Modern transaction-based PM systems are focusing on hybrid designs that combine previous strategies. One of the most recent is the use of CoW + redo logging. This novel approach involves creating a working copy of persistent data within DRAM to utilize its fast performance with hardware transactions, saving the changes to a redo log in PM.

2.3 PM SYSTEMS

Transactions are a natural choice when implementing programming support for PM systems. With the availability of hardware transactions, new designs have been proposed to solve the issue of accessing persistent data ((LIU et al., 2017), (CASTRO; ROMANO; BARRETO, 2019), (GILES; DOSHI; VARMAN, 2017), (CASTRO et al., 2021)), since such an operation will abort the transactions in current processors. The key technique used by

current systems is to map the persistent heap into DRAM (using CoW, for instance) so that hardware transactions can operate normally. As transactions execute, they create a redo log that is later persistent and replayed. Figure 4 presents the overall architecture.

Figure 4 – Modern approach to implementing failure-atomic updates: CoW is used to create a working copy of the persistent heap in DRAM – updates are first appended to redo logs and then persisted and reproduced in the persistent heap by background threads.



Source: (BALDASSIN et al., 2021).

Typically, implementations of failure-atomic updates can be separated into four well-defined steps: CoW, perform, persist, and reproduce. First, a working copy of the persistent data, usually in DRAM, is created (1), this may leverage the OS CoW support or be executed using custom techniques, as observed in (WU et al., 2021b) and (GENÇ; BOND; XU, 2020). The next steps are the most unique as they do not require to be synchronous, that is, they are not required to be executed immediately after the previous one is concluded. The subsequent perform step (2) is where the load/store operations are properly executed in place within the working copy, while also being kept in a volatile redo log. The persist step (3) entails persisting the redo logs generated in the previous phase. This step guarantees that the modifications performed by the transaction are persistently committed, that is, in case of a failure the redo log can be used to rebuild the persistent state. The final step, reproduce (4), involves replaying the redo logs within the persistent heap. This approach prevents the log space from becoming full and creating execution bottlenecks. This replay phase can potentially be performed periodically by another thread in the background.

In the context of CoW + redo logging, an essential consideration revolves around the sequencing of volatile log persistence. This significance arises from the fact that the log is generated by a transaction and remains volatile after the commit (recall that a transaction cannot perform writes to PM without aborting). Therefore, other transactions may also require the persistence of their respective logs. As such, the PM system must ensure that the logs are replayed in the order in which their respective transactions are committed. A timestamp is usually employed to solve this issue, coming in two flavors: (i) physical timestamp, which is based on a hardware counter; and (ii) logical timestamp, based on a global variable that is incremented by the transactions. It is worth noting that hybrid approaches for logging are commonly used due to their high flexibility, i.e., having an asynchronous approach to the persist step removes costly fence instructions from a critical section, improving performance.

Despite the common framework based on CoW + redo logging, each recently developed PM system introduces its own distinctive characteristics aimed at addressing specific challenges.

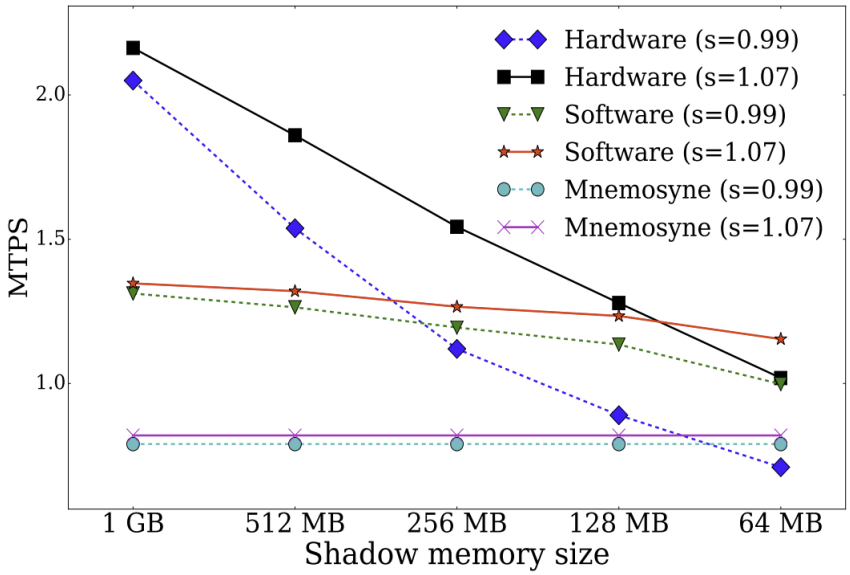
2.3.1 THE PROBLEM OF LIMITED DRAM

Note that the previously discussed mechanism used by PM systems requires the available DRAM space to be large enough to hold the working copy. Otherwise, the operating system swapping mechanism will be activated, which might inadvertently impact performance. Most of the current PM systems that implement this mechanism do not deal with this problem explicitly, thus assuming that memory will not be an issue. Therefore, the behavior under this scenario is not well understood in the literature, a fact that we intend to address in this dissertation.

The only work that quickly mentions the problem is DudeTM (LIU et al., 2017). The authors acknowledge the problem and propose an alternative to the operating system's default swapping mechanism. Instead of writing the data to the swap area in a page-out event, they simply discard the page. Note that this is correct because the changes that are performed on such a page are already recorded in the redo log and can later be replayed. However, when the same page is later required (i.e., a page-in event occurs), it can only be accessed after we are sure that the logs have already been replayed. To implement that, DudeTM adds a timestamp to each page and checks, during page-in, if the replayer have at least replayed the page (the replayer also keeps a timestamp of the most recently replayed page).

However, DudeTM provides very little experimental evidence of the usefulness of the proposed alternative paging mechanism. Figure 5 shows the performance evaluation conducted by the authors in their paper. They use a very simple microbenchmark based on a B+-tree key-value store, which is accessed following a Zipfian distribution (with parameters

Figure 5 – DudeTM's swap overhead analysis.



Source:(LIU et al., 2017).

$s = 0.99$ and $s = 1.07$). They use a 1GB NVM and a shadow RAM varying from 1GB (no paging required) to 64MB (X axis). The figure indicates that the throughput, measured in Millions of Transactions Per Second (MTPS), decreases as the size of shadow memory gets smaller. Their evaluation takes into account a hardware-based paging mechanism (more on that later) and a software-only implementation, as well as the Mnemosyne (VOLOS; TACK; SWIFT, 2011) system for comparison. It can be notice that the hardware approach is more efficient with larger shadow RAM. A flat line is seen with Mnemosyne, since it does not use the shadow-memory approach.

As can be seen, the only evaluation available is very simple and does not even present a comparison with the default operating system swapping mechanism. Furthermore, it is not clear how the system would perform under a more realistic benchmark, a situation we seek to address in this dissertation.

Another important point regards the infrastructure to conduct the paging experiments. Since most of the academic datasets are still small enough to fit in DRAM, some way to limit the amount of shadow RAM is required. DudeTM uses its own software-based paging implementation, as well as an alternative using the Dune library (BELAY et al., 2012a) – an approach that uses Intel's VT-X virtualization technique to enable the user space code to manage page tables. Dune requires modifications to the Linux operating system and has not been updated since 2017, which makes it impractical for contemporary use. Moreover, DudeTM's software-based implementation requires the instrumentation of all loads and stores, which will add additional overhead to PM systems based on hardware transactions. Therefore, this dissertation also presents an alternative approach to simulate paging based on intercepting operational system page faults (see Chapter 3). Another alternative would be

to use the `userfaultfd` library (USERFAULTFD, 2023), which allows the user space code to be notified of page faults in designated virtual address ranges. However, it does not provide an easy way to deal with page evictions. There was an attempt to extend `userfaultfd` with that feature (CALDWELL et al., 2017), but it is not part of the open source distribution and has the major inconvenience of being kernel specific.

3 PAGING FRAMEWORK

This chapter presents the paging framework that we have devised to run the experimental evaluation in Chapter 4. As discussed earlier, DudeTM's approach requires instrumenting both load and store instructions which, in turn, would add an overhead in the fast-path of hardware transactions. Therefore, the motivation for building this framework is to have an approach that avoids instrumenting this fast-path.

We start by presenting the overall architecture of our paging framework in Section 3.1. It allows for intercepting page faults and explicitly mapping pages using the `mmap` system call. We then discuss, in Section 3.2, how DudeTM's alternative paging mechanism can be implemented in this framework. To better illustrate how the framework can be integrated with current PM systems, we use SPHT (CASTRO et al., 2021), a modern PM system that makes use of hardware transactions.

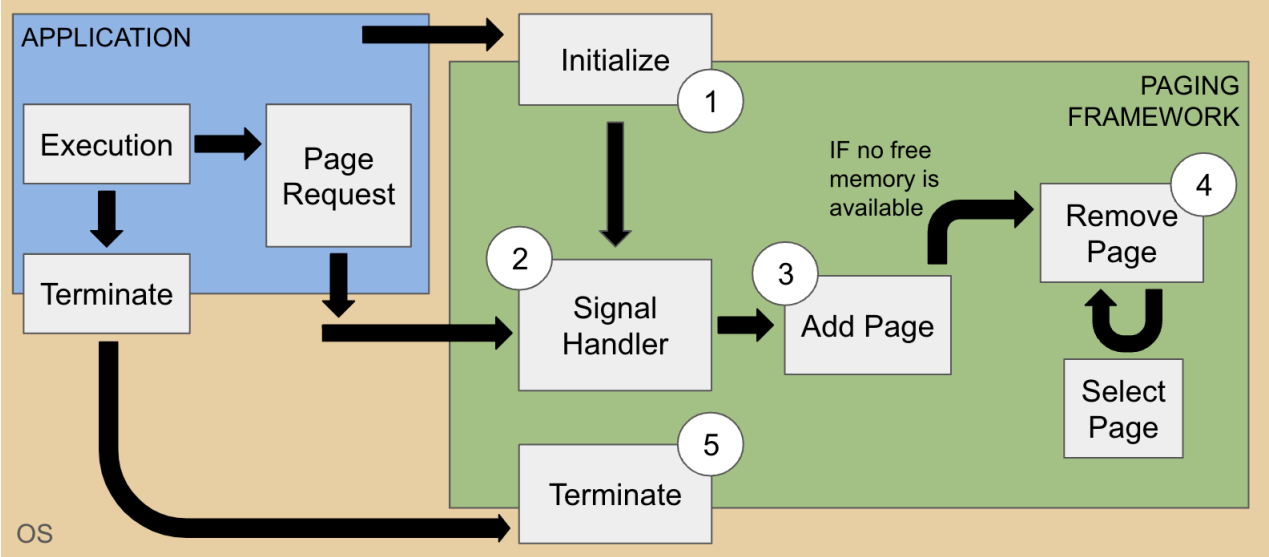
3.1 ARCHITECTURE

In order to implement the paging mechanism without specific hardware requirements like Dune (BELAY et al., 2012b) or that relied on specific Linux kernel versions, our implementation makes use of Linux signals and memory mapping system calls, with a combination of the SIGSEGV handler and the `mmap` and `munmap` functions. The key data structure used is a bitmap that determines, for each page in the system, whether it is currently mapped or not. Initially, all pages are unmapped and, as the operating system SIGSEGV signals are captured, the respective bit of that page in the bitmap is set and the page is mapped by means of a `mmap` system call.

The SIGSEGV handler works by capturing the memory address call sent by the OS to the main program when a fault occurs. We distinguish between *real faults* and *controlled faults*; the former refers to page faults caused by buggy references and should terminate the program, whereas the latter refers to faults to the persistent memory region controlled by the paging framework. A persistent memory region is initially set by the paging framework in order to correctly distinguish between real and controlled faults. Two parameters are used to control this aspect: *working copy size* (`wcs`) and *persistent heap size* (`phs`). The `phs` parameter refers to the total amount of physical memory available in the system. Its value can be set to the required amount of persistent memory for a given application (e.g., 512GB). The `wcs` parameter is usually set to a value smaller than `phs` in order to simulate scenarios where the amount of DRAM is limited. By controlling the `wcs` and `phs` parameters, we can simulate different scenarios to check the behavior of the underlying PM system.

Figure 6 presents a simplified overview of the paging framework. From the application's perspective, the framework consists of two main parts: an initialize (1) and a terminate (5) functions.

Figure 6 – Simplified overview of the Paging Framework.



Source: Elaborated by the author.

During initialization, the page table bitmap is allocated and reset, meaning that all pages are unmapped. The total number of pages in the system is controlled by the `phs` parameter. The maximum number of pages that can be mapped at once is given by the `wcs` parameter. These parameters are used in the initialization step and cannot be changed during execution time. As the application starts and memory addresses are referenced, the page faults are redirected by the operating system to our framework to perform the page insertion (3). It works by first checking whether there is a free page. If that is the case, it then marks the page as used in the bitmap and invokes the `mmap()` system call to map the respective page.

The framework uses two main thresholds to manage page eviction: one to control when pages should start to be evicted (`startth`) and another to control when to stop removing pages (`stopth`). When a page is added, we check if the number of used pages is greater than `startth`. If that is the case, old pages must start to be evicted in order to make room for new ones. Pages are evicted until the `stopth` limit is reached. For example, assume that `startth` is 90%, `stopth` is 85% and the system can support 100 mapped pages in total. Once 90 pages have been mapped, the next call to *add_page* will trigger the page removal procedure (4). An LRU-like algorithm is responsible for choosing the next page to be evicted. The page removal function keeps executing until 85 pages are available (thus evicting 5 pages in a row in this example).

Although not explicitly shown in Figure 6, there are two main methods by which the page removal procedure can be invoked. The first one is synchronous, that is, the procedure is executed directly during the page insertion operation (same thread). The other method is asynchronous: a specific thread is created to manage page removal. The `startth` threshold is checked during paging insertion and, if the limit has been reached, the page removal thread is awakened. This thread will continue to remove pages until the `stopth` threshold is reached. The main reason for having two different thresholds is to avoid constantly calling the page removal code when the percentage of used pages in the system is high. This is particularly important, for efficiency reasons, in the asynchronous case, as it requires synchronizing the thread that is doing insertion with the one doing removal (a typical producer/consumer synchronization).

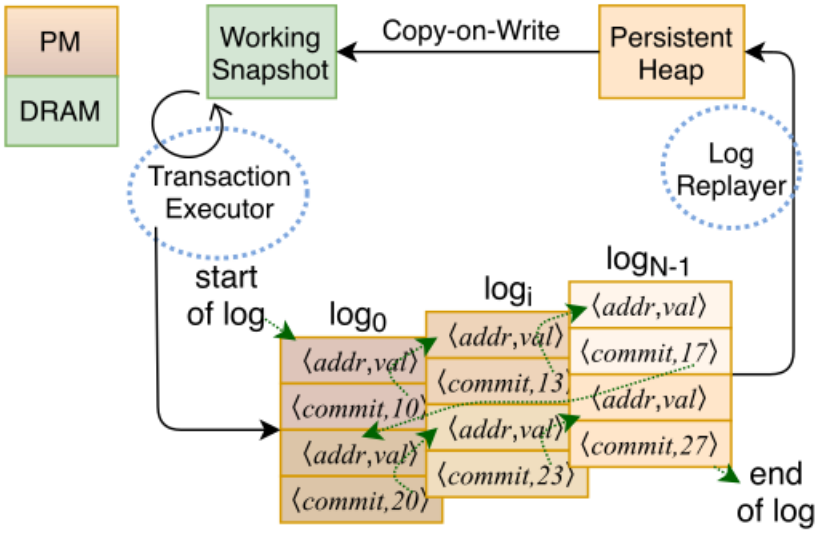
The advantage of having a separate thread performing page removal is a potential increase in parallelism, as the thread that triggered the removal action can resume execution. However, there is the overhead of synchronization. In the end, the choice will depend on the workload as we will discuss in the evaluation part (Chapter 4).

3.2 SPHT INTEGRATION

SPHT (Scalable Persistent Hardware Transactions) (CASTRO et al., 2021) is a state-of-the-art PM system, which can be considered an evolution of what was presented in NV-HTM (CASTRO; ROMANO; BARRETO, 2019), and is the system chosen to evaluate the paging behavior. Although still using HTM through commodity hardware as many other systems (e.g. (GENÇ; BOND; XU, 2020), (WU et al., 2021b), (GILES; DOSHI; VARMAN, 2017)), it made significant improvements to the commit protocol in order to improve system throughput, reporting a significant increase in performance over previous implementations.

The SPHT design follows the approach discussed previously in Section 2.3. It consists of two main parts: the Transaction Executor (TE) and the Log Replayer (LR), as illustrated in Figure 7. TE is responsible for executing transactions in multiple working threads and also uses the OS' CoW mechanism to map a persistent heap into the process's memory address space. Recall that while transactions are executed using the hardware support, they build a redo log. After the hardware transaction is logically committed, it is still necessary to persist the redo log in the correct execution order (among all the transactions in the systems), and only then can be assumed that the transaction is effectively committed. This may require a transaction to wait for older transactions to persist their redo logs before it can persist its own redo log. In the specific case of SPHT, a group commit protocol is used to reduce the time that the transactions wait for older transactions. Each persisted log ends with a *commit marker* which contains the timestamp of the transaction that generated it. The LR is responsible for applying the changes recorded in the redo logs (replay) in the persistent heap.

Figure 7 – Overview of SPHT architecture.



Source:(CASTRO et al., 2021).

It also keeps a *global commit marker* (GCM) with the timestamp of the last log replayed.

By default, SPHT does not manage any paging scenarios. As a result, if the shadow memory space is exhausted, the operating system's swapping mechanism will be triggered. In order to simulating a typical swap operation with the paging framework is straightforward. We reserve a swap area on the disk to copy the contents of the pages evicted. When a page fault occurs and the respective page must be mapped, we first check to see if that page is already in the swap area. If affirmative, we simply copy its contents back. Notice that using the swapping mechanism is safe with respect to consistency of the persistent heap. In case of a failure, the redo logs can be used to restore the system to a consistent state, regardless of the state of the swap area.

A potential problem with the swapping approach is the overhead of managing the swap area (moving data from/to the disk). As such, an alternative approach was proposed by DudeTM (LIU et al., 2017). It does not perform any action on page removal (the page is just discarded), since the modifications have already been recorded in the redo logs and will eventually be replayed. However, when the same page needs to be accessed, it is necessary to ensure that all updates to the persistent page have been replayed before remapping it. The way DudeTM solves this issue is by keeping a *touching ID* for each page, which is the ID of the last transaction that was written on the page. Before loading a persistent page into the working memory, it compares its touching ID and the ID of the transaction whose changes have been more recently replayed. If the touching ID is newer (bigger value), then it means that it was not replayed yet, and thus it needs to wait. This approach is beneficial when the waiting time required to map the page is minimal. It is important to note at this point that DudeTM has not evaluated this potential optimization with respect to a conventional

Algorithm 1: Simplified page management pseudo-code

```
   Input : new_page
1  // assume there is free memory on entrance
2  // wait replayer
3  while hashmap(new_page) > GCM do
4      sleep some microseconds;
5  mmap(new_page);
6  used_space += 4096;
7  if used_space > startth then
8      remove_page();
9      used_space -= 4096;
```

swapping scheme.

Since SPHT does not implement any optimization regarding page handling, we have adapted DudeTM's approach to evaluate its performance. For that, we have used a hashmap that stores, for each page in the system, a timestamp of the most recent transaction that has been written to it. Right before a transaction logically commits, we scan its redo log and update the timestamp of all modified pages in the hashmap. When a page fault signal is caught, we compare its timestamp with the most recently replayed page (the GCM, kept by the RL) and wait, if necessary. This approach is named `OpenHash` in the rest of this document. Not that the only change required in the SPHT code is the addition of the hashmap and the corresponding page updates before the transaction logically commits.

Algorithm 1 presents a simplified version of the changes required in the `add_page` procedure of the paging framework in order to implement `OpenHash`. The first thing that needs to be checked is whether the timestamp of the new page is greater than the LR's global commit marker (GCM) (Line 3). If that is the case, then the new page contents have not yet been replayed and we need to wait until the condition is false. After that criterion is met, the page is finally mapped (Line 5), the amount of available space is increased (Line 6) and, if the `startth` threshold is reached (Line 7), it calls the procedure to remove pages (Line 8) in order to make some room for newer ones. The behavior of the page removal code depends on whether the synchronous or asynchronous version is used. In the former case, the same thread will execute the operation. In the latter case, a background thread responsible for removing the pages is awakened, and the execution will proceed immediately.

3.2.1 PERFORMANCE AND CORRECTNESS CONSIDERATIONS

Compared to a conventional swap mechanism, `OpenHash` avoids the overhead of moving pages to the swap area. However, it has two main critical points that need to perform well in order for this approach to pay off. Firstly, the hashmap that stores the pages' timestamp

must be updated sufficiently fast. Recall that these updates are performed as the last step of the transaction before it logically commits. One potential problem here is that, since the updates are done inside a hardware transaction, it increases the transaction write set size. Therefore, it might cause the transaction to abort and retry if too many updates are performed.

These updates must be done inside the transaction and not afterward since this would avoid increasing the transaction write set. This subtle issue is not discussed by DudeTM at all. In fact, after looking at their source code, we realized that they indeed update the timestamps after the transaction is persisted. Note that this is incorrect for the following reason. Consider a transaction that has been completed and its redo logs persisted. However, the timestamps of the modified pages have not yet been updated, and neither the logs have been replayed. Assume now that one of these pages is evicted and then remapped because a new transaction is accessing it. The timestamp of this page is outdated, and therefore the transaction will observe stale data (the condition in Line 3 of Algorithm 1 will fail). The other critical point for `OpenHash` to be effective is the waiting time while LR replays the changes to the needed page (again, Line 3 of Algorithm 1). If LR is too slow, it might prevent transactions from running optimally, resulting in performance loss.

The hashmap implementation used by `OpenHash` uses closed addressing to resolve collisions. This implies building a linked list with a key–value pair for each page and timestamp. As more pages are mapped to the same bucket, the size of this linked list increases. Using an open addressing scheme could potentially avoid allocating new memory for the linked lists, but the number of memory addresses touched (and, therefore, the transaction's read set size requirement) might still be an issue. Therefore, we devised an alternative implementation, named `ImpHash`, which forces each bucket in the hashmap to have a single timestamp. Note that this allows page aliasing but, since we are always storing the most updated timestamp, it is still correct. The drawback is that it might force a transaction to wait longer than necessary in Line 3 of Algorithm 1.

4 EXPERIMENTAL ANALYSIS

In this chapter, we characterize the behavior of different paging mechanisms in scenarios with shadow memory restrictions. We start by describing the benchmark and workloads used for performance characterization (Section 4.1), followed by the experimental settings (Section 4.2) and finally the experimental results (Section 4.3).

4.1 TPC-C

In order to assess the performance and effectiveness of the different paging schemes, it is important to use a relevant benchmark. The TPC-C (COUNCIL, 2018) is an On-Line Transaction Processing (OLTP) benchmark commonly employed for analyzing PM systems (GU et al., 2019; WU et al., 2021a) which presents realistic workloads.

TPC-C simulates a complex industry scenario that can manage, sell, and distribute a product or a service, each represented by its many available transactions. Those services are executed by the many warehouses the user specifies. TPC-C presents the distinct advantage of being built from the ground up with transactions in mind, so it does not have any critical information being lost upon replay, allowing our implementation to work properly.

The benchmark has transactions that are intended to simulate operations in a product distribution logistics center, and for that it uses up to 5 different transactions to form a workload, including: Delivery, New Order, Order Status, Payment, and Stock Level (SPECIFICATION, 1994).

- **Delivery** transactions consist of processing a batch of 10 orders not yet delivered. First, it processes the order, updates the costumer's balance, and lastly removes the order from the new-order list;
- **New Order** transactions consist of entering a complete order from a single database transaction. It works by creating an order header and ordering a variable number of items (average 10 items);
- **Order Status** transactions query the status of a customer's last order. It does that by finding the costumers last order and checking the delivery date for each order item (average of 10 items);
- **Payment** transactions update the customer's balance and reflects the payment on the district and warehouse sales statistics. It selects a costumer based on its number or

selects a costumer based on their last name. The payment value is then randomly selected and the database updated;

- **Stock Level** transaction determines the number of items sold whose stock level is below a certain threshold of the last 20 orders. It first examines the next available order number, examines all items on the last 20 orders, and, for each unique item, examines whether the stock available is below the threshold.

Different workloads can be generated by changing the percentage of operations performed in the warehouses. We chose to use two broad categories: `light-focus` and `heavy-focus`.

- **`light-focus`** is comprised of 50% Order Status, 25% Payment and 25% Delivery transactions, mid-weight read-only, light-weight read-write, and relaxed read-write respectively;
- **`heavy-focus`** is taken from the TPC-C specification, consisting of 45% New Order and 43% Payment, while incorporating only 4% Stock Level, 4% Delivery, and 4% Order Status transactions, with New Order being a mid-weight read-write transaction, Payment being light-weight read-write, and Stock Level being a heavy read-only transaction.

Whereas the transactions in the `light-focus` workload predominantly perform lighter read operations, the `heavy-focus` configuration is more read-write intensive. Each warehouse can serve 3.000 costumers and can stock 100.000 items sold. In our experiments a total of 32 warehouses were used.

4.2 EXPERIMENTAL SETTINGS

We selected the SPHT system with our paging framework incorporated to evaluate performance under different stress scenarios using the TPCC workload. In particular, the following configurations are used:

1. **`OpenHash`** - on a page-out event, nothing happens, the page is simply discarded, and none of its changes are written to the persistent heap. On a page-in event, it first makes sure its contents reflect the most recent updates by checking whether the log-replayer has updated them. A hashmap (500,000 entries) is used to annotate, for each page, the timestamp of the last transaction that wrote to it;

2. **ImpHash** - an optimization of `OpenHash` that allows mapping different pages to the same bucket in the hashmap (instead of chaining them through a linked-list). It speeds up insertion, but can cause slowdown during page-in events because it can force a thread to wait for the log replayer longer than it should;

3. **Swap** - this is the default operating system behavior: on a page-out event, the page is written to a swap area (SSD) and recovered in case the page is needed again.

We control the level of memory utilization using the total persistent heap size (`phs`) and the working copy size (`wcs`) parameters. In particular, `phs` is kept constant to a value that is always larger than `wcs`. We then vary `wcs` to decreasing sizes in order to cause a more controlled number of page faults. For each workload, we first find, by trial and error, the minimum value for `wcs` that will not cause page faults: this is the 100% memory usage baseline. We reduce `wcs` to smaller values in order to force scenarios in which working copy memory is more restricted, forcing page faults to occur. In the experiments, values ranging from 100% to 20% are used.

The thresholds for determining when to start (`startth`) and stop (`stopth`) removing pages were set to 97% and 95%, respectively. We performed some sensitivity analysis and found that those numbers presented a good trade-off. Initiating page removal before the system reaches 100% allows for more parallelism. Likewise, removing a good amount of pages in batches avoids having to call the page removal code frequently. We tested both synchronous and asynchronous page removal and found that the former provided better overall results. Having a separate thread for doing asynchronous removal seemed to cause more overhead due to the extra conditional synchronization required between the worker threads and the page removal thread. In practice, we found out that the best scenario for the asynchronous case (having the pager thread working in parallel to the worker threads) almost never really materialized since the worker threads would cause a page fault and needed to wait for the page removal thread anyway.

All experiments were performed with a system equipped with an Intel(R) Xeon(R) Gold 5317 CPU (total of 24 logical cores), 256 GB RAM in eight-channel mode, 512 GB of Intel Optane PM DC (Intel Persistent Memory 200 Series) and Ubuntu Server 20.04. The number of threads used in the experiments ranges from 1 to 24. After the system is initialized and the TPCC workload is configured and loaded, it first unmaps all pages before running the workloads. A warm-up phase is first executed to allow the system to enter a steady state. Only then do we collect the performance metrics for the next 10 seconds. For each workload, number of threads, and a given memory restriction, we measure the throughput (transactions executed per second) of the system. The data presented here is the result of 3 runs for each configuration, from which an average value is obtained and presented together with the resultant standard deviation in the plots. Note that since the execution time

is relatively long (10 seconds), we opted for only doing 3 runs, which would be equivalent to running the experiments 15x in 2-second windows (both presenting a total of 30 seconds).

4.3 PERFORMANCE CHARACTERIZATION

In this section, we discuss the performance of the different methods for handling page faults, namely `OpenHash`, `ImpHash`, and `Swap`. In order to understand the different tradeoffs involved, we seek to answer the following questions:

- How much overhead is added by `OpenHash` and `ImpHash`? (Section 4.3.1).
- How much overhead is caused by paging? (Section 4.3.2).
- How much improvement do `OpenHash` and `ImpHash` offer? (Section 4.3.3).

The following sections discuss each of these questions.

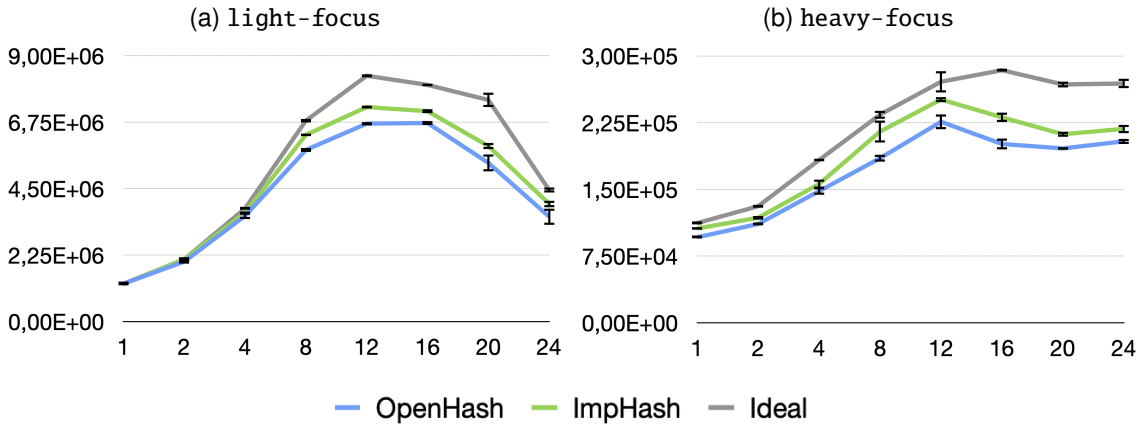
4.3.1 OVERHEAD OF `OpenHash` AND `ImpHash`

There are two main potential sources of overhead for `OpenHash` and `ImpHash`: i) the additional computation required to annotate each accessed page with the corresponding timestamp before a transaction commits (inserting the timestamps into the hashmap); ii) the time required to wait for a page to be consistent during a page-in event (consistency wait).

In order to characterize the first overhead scenario, we run the two paging schemes in a situation in which no page faults ever occur and obtained the throughput shown in Figure 8, for both (a) `light-focus` and (b) `heavy-focus` workloads. Since there are no page faults being caused, `OpenHash` and `ImpHash` are free of the second overhead source (consistency wait). The ideal case (gray line) presents the best performance since there are no page faults and no other overheads exist. Therefore, the difference in throughput displayed in the figure is the overhead caused by inserting the timestamps in the hashmap.

For both workloads, we can see that `OpenHash` and `ImpHash` are slower than the ideal case, as expected. We can also observe that the optimization of only storing the most recent timestamp in `ImpHash` provides better throughput compared to `OpenHash`, which stores all timestamps of pages that map to the same bucket in the hashmap via a linked list. Having a faster hashmap insert operation has two main benefits. First, it reduces the transaction length and may avoid conflicts with other transactions. Second, it reduces the write footprint and can potentially avoid capacity aborts. Capacity aborts are caused when the number of writes made by a transaction is greater than the hardware capacity to store them.

Figure 8 – TPC-C throughput (x1.000.000) without page faults.



When analyzing the `light-focus` workload, `OpenHash` can be 40% slower and `ImpHash` 26% slower than the ideal case with 20 threads in the worst-case scenario. Note that below 8 threads, the difference is not representative (around 5%). The performance difference between `OpenHash` and `ImpHash` is 10% on average. When considering the `heavy-focus` workload and comparing the two alternatives with the ideal scenario, `OpenHash` is 42% slower with 16 threads and `ImpHash` is 26% slower with 20 threads. Therefore, the worst cases for both workloads are very similar. In contrast to `light-focus`, however, the performance differences when looking at the case with 1, 2, and 4 threads are more noticeably pronounced with `heavy-focus`. Recall that in this workload the transactions are more write intensive, which causes more pages to be written, and therefore the number of hashmap insertions is bigger, thus adding more overhead even in the cases with a small number of threads. As more threads are added, the likelihood of conflicts between transactions tends to increase. Note that the hashmap insertion required by `OpenHash` and `ImpHash` increases the length of the transactions, increasing even more the likelihood of conflicts. This helps to understand the overhead added by both approaches.

As for the second overhead source, the consistency wait, Tables 4 and 5 show the percentage of waiting time (lines 3-4 of Algorithm 1) relative to the entire time required to add a new page (Algorithm 1). For each table, we show the overhead as the number of threads increases and the fraction of the working copy memory is reduced, as a percentage of the total available memory (from 100% to 50%).

The overhead depends on the number of pages written by a transaction, which determines the number of hashmap insertions, and the hash scheme employed, `OpenHash` or `ImpHash`. Ideally, there should be no consistency wait overhead in the 100% configuration since a page is never removed (no page-outs) and, as such, all pages are new (no need to wait for it to become consistent). Although this is true for `OpenHash`, Tables 4 and 5 show a different scenario for `ImpHash` in both workloads. What is happening here is that `ImpHash` allows aliases, which means that when a page is first mapped, the hashmap can

Table 4 – Percentage of time waiting for a consistent page in the page-in event for both `OpenHash` (OH) and `ImpHash` (IH). The results are shown for different working copy fractions (from 100% to 50%) using the `light-focus` workload.

THR	MEMORY PERCENTAGE											
	100 %		90 %		80 %		70 %		60 %		50 %	
	OH	IH	OH	IH	OH	IH	OH	IH	OH	IH	OH	IH
1	0.0%	1.8%	1.2%	2.2%	1.2%	1.6%	0.9%	0.9%	0.6%	0.8%	0.5%	0.7%
2	0.0%	0.4%	1.3%	1.6%	0.9%	1.0%	0.6%	0.8%	0.5%	0.5%	0.4%	0.5%
4	0.0%	1.7%	1.3%	1.6%	0.9%	1.1%	0.6%	0.7%	0.5%	0.6%	0.4%	0.4%
8	0.0%	1.3%	1.2%	1.1%	0.9%	1.1%	0.6%	0.8%	0.4%	0.6%	0.4%	0.5%
12	0.0%	0.3%	1.1%	1.4%	0.8%	1.1%	0.6%	0.8%	0.4%	0.5%	0.5%	0.5%
16	0.0%	1.0%	1.0%	1.2%	0.7%	0.9%	0.5%	0.7%	0.5%	0.6%	0.3%	0.4%
20	0.0%	0.4%	0.8%	1.0%	0.7%	0.9%	0.5%	0.7%	0.4%	0.6%	0.4%	0.4%
24	0.0%	0.5%	0.4%	0.7%	0.7%	0.8%	0.5%	0.6%	0.4%	0.4%	0.3%	0.4%

Table 5 – Percentage of time waiting for a consistent page in the page-in event for both `OpenHash` (OH) and `ImpHash` (IH). The results are shown for different working copy fractions (from 100% to 50%) using the `heavy-focus` workload.

THR	MEMORY PERCENTAGE											
	100 %		90 %		80 %		70 %		60 %		50 %	
	OH	IH	OH	IH	OH	IH	OH	IH	OH	IH	OH	IH
1	0.0%	4.5%	0.7%	2.4%	1.3%	1.4%	0.9%	1.1%	0.8%	0.9%	0.7%	0.9%
2	0.0%	4.1%	0.9%	1.6%	0.9%	1.1%	0.7%	0.9%	0.6%	0.6%	0.5%	0.6%
4	0.0%	3.8%	0.8%	1.4%	1.0%	1.0%	0.8%	0.8%	0.6%	0.7%	0.6%	0.6%
8	0.0%	4.2%	0.5%	1.1%	0.8%	1.0%	0.6%	0.9%	0.7%	0.8%	0.6%	0.7%
12	0.0%	4.2%	0.7%	1.2%	0.8%	1.2%	0.8%	0.9%	0.7%	0.8%	0.6%	0.6%
16	0.0%	3.5%	0.0%	2.4%	0.7%	1.0%	0.7%	0.8%	0.6%	0.7%	0.6%	0.6%
20	0.0%	2.7%	0.0%	3.2%	0.4%	1.0%	0.7%	0.8%	0.6%	0.6%	0.5%	0.5%
24	0.0%	2.4%	0.0%	3.2%	0.0%	1.5%	0.6%	0.7%	0.5%	0.6%	0.5%	0.5%

return a timestamp of a previously mapped page that occupies the same hashmap bucket. It does not cause any errors but can force the thread to waste some time until the page becomes consistent. This overhead varies from 0.4% (20 threads) to 1.8% (1 thread) for the `light-focus` workload and 2.4% (24 threads) to 4.5% (1 thread) for `heavy-focus`. The overhead is slightly higher for `heavy-focus` because it performs more writes, thus increasing the likelihood of hashmap aliasing (a larger number of pages are inserted into the hashmap).

As the working memory fraction is reduced, page faults start to happen more frequently. This, in turn, tends to increase the total time required to map a new page since the page removal code (line 8 of Algorithm 1) is invoked more frequently, thus diluting the relevance of the waiting time required to check the page consistency. Overall, the tables show that the consistency wait overhead is very small, and therefore the optimization provided by `ImpHash` is worth having, since it addresses the main source of the paging overhead: the hashmap insertion.

4.3.2 HOW MUCH OVERHEAD IS CAUSED BY PAGING?

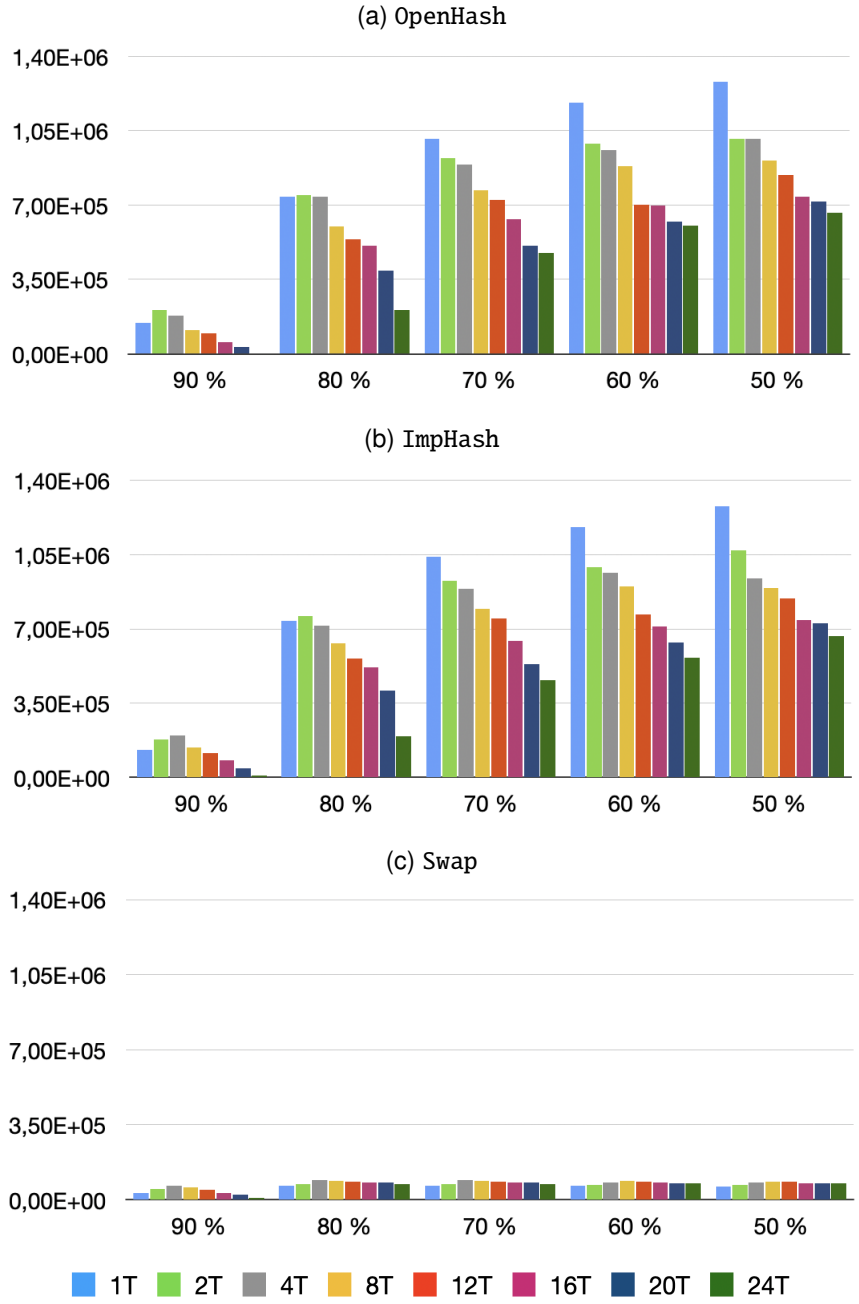
When evaluating the paging behavior, ideally we would like to control the number of page faults and check how the system performs under different scenarios. However, there is no simple way to directly control this parameter. As such, we use the working copy size (`wcs`) as a proxy and proportionally vary its size to force different levels of page faults. The idea is that the smaller the `wcs`, the bigger the number of page faults generated. Figures 9 and 10 show the absolute number of page outs as the working copy size decreases, from 90% to 50%.

As can be seen, using the `wcs` as a proxy to indirectly control the number of page faults is valid. For both `light-focus` and `heavy-focus` workloads, we can see that there is an important increase in the level of page faults generated up to 50%, when this number starts to flatten. Although the general behavior is similar for `light-focus` and `heavy-focus`, notice that `light-focus` tends to generate more page faults in absolute numbers since it is more read intensive and its throughput is also higher (discussed later). Also note that page faults increase more significantly with memory reduction in `light-focus` than in `heavy-focus`. This is also due to the way both workloads were constructed: whereas `light-focus` tends to have more reads and displays a higher throughput, `OpenHash` transactions perform more writes and are longer, favoring more aborts and thus presenting a comparatively lower throughput.

Also note that `OpenHash` and `ImpHash` have very similar behavior. The number of page faults tends to decrease with more threads due to the increased parallelism level. Notice that the scale of the plots in each figure is the same for all paging schemes to highlight the contrast in the number of page faults caused by `Swap`. When a page fault occurs, the transactions have to acquire a Single Global Lock (SGL), which serializes the execution. The amount of time that `Swap` keeps the lock acquired is much longer compared to `OpenHash` and `ImpHash`, since it needs to copy the page to the swap area in the occurrence of each fault, while `OpenHash` and `ImpHash` simply discard the page. As a result, `Swap` tends to execute fewer transactions and touch a smaller range of pages, thus producing fewer page faults.

The performance of the three paging schemes is directly related to the number of page faults generated. Figure 11 shows the overall throughput for `light-focus` workload considering different percentages for the working copy size. Firstly, notice that for this workload, the throughput tends to get worse after 16 threads. This is due to the fact that transactions are more read intensive and the increased level of parallelism tends to rapidly increase the number of page faults generated (as shown in Figure 9) which, in turn, decreases throughput. It can be seen that, for the three paging schemes, the throughput rapidly decreases from 90% to 80% and also from 80% to 70%. The system practically does not scale anymore from 50% downwards. This decrease is much more pronounced for `Swap`.

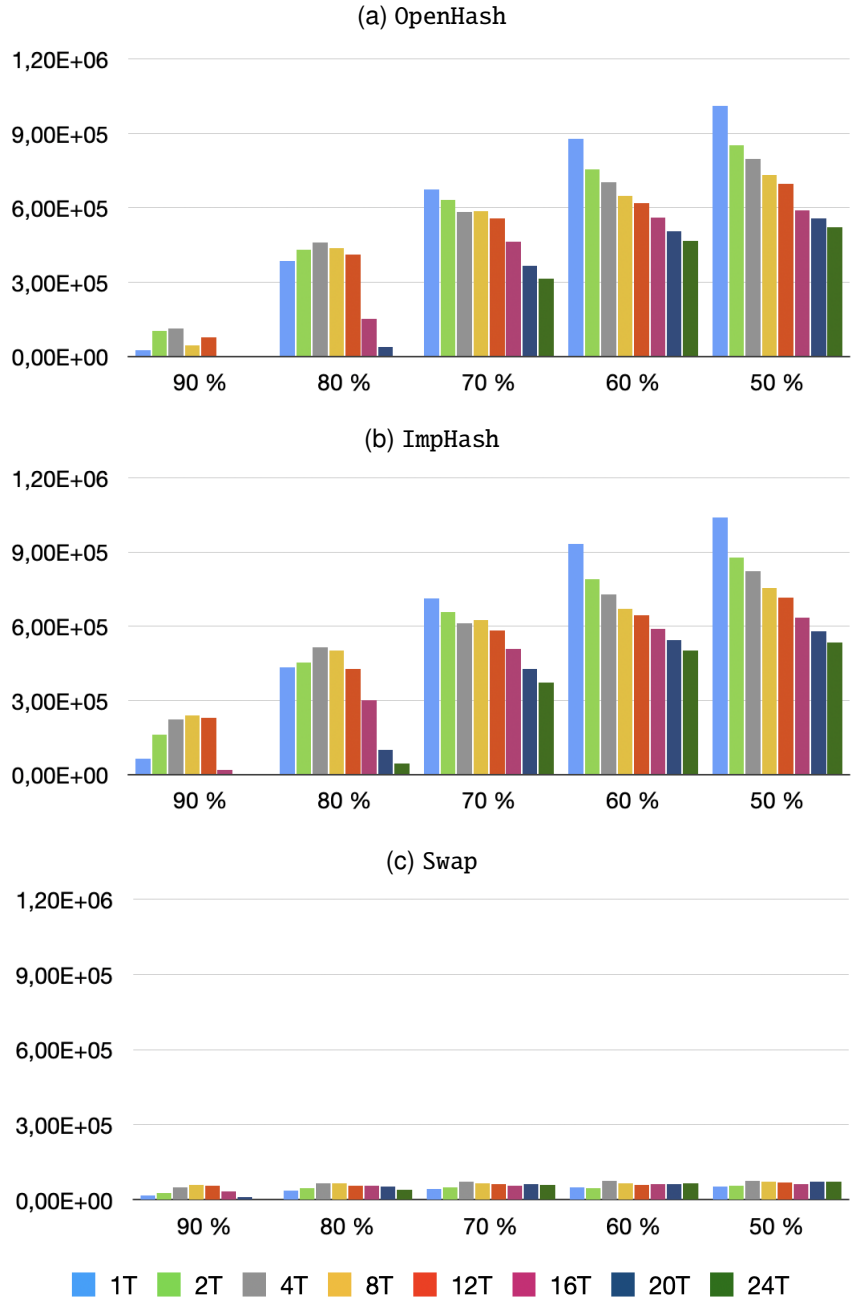
Figure 9 – Number of page outs for each evaluated memory scenario for each number of threads used in the `light-focus` workload.



Considering the results with 12 threads, the performance loss with 90% is around 10% for `OpenHash`, 13% for `ImpHash`, and 43% for `Swap`. When we consider the loss with memory percentage of 80% compared to 90%, they are 50%, 49%, and 76%, respectively.

A different behavior is presented with the `heavy-focus` workload, as shown in Figure 12. Compared to `light-focus`, here the absolute throughput numbers are an order of magnitude lower, due to the more write-oriented nature of the transactions. Moreover, there is no steep performance drop after 16 threads, although the performance tends to get flatter. Recall from the previous page fault analysis (Figure 10) that the absolute number of

Figure 10 – Number of page outs for each evaluated memory scenario for each number of threads used in the `heavy-focus` workload.



page faults here does not increase as quickly.

Looking at Figure 12, we can also observe that the performance drop with increasing memory pressure is smoother. For `OpenHash` and `ImpHash`, a more noticeable loss can be seen from 70% downwards. Considering 24 threads, the loss at 70% compared to 100% is 12% with `OpenHash`, 14% with `ImpHash`, and 37% with `Swap`. Performing the same comparison with a memory pressure of 50%, we have a loss of 57%, 61%, and 79%, respectively. We found that the `heavy-focus` workload reaches the same level of practically no scaling anymore when the working copy size is reduced to 20% of the total memory size.

Figure 11 – TPC-C throughput (x1.000.000) for the `light-focus` workload with the three paging schemes.

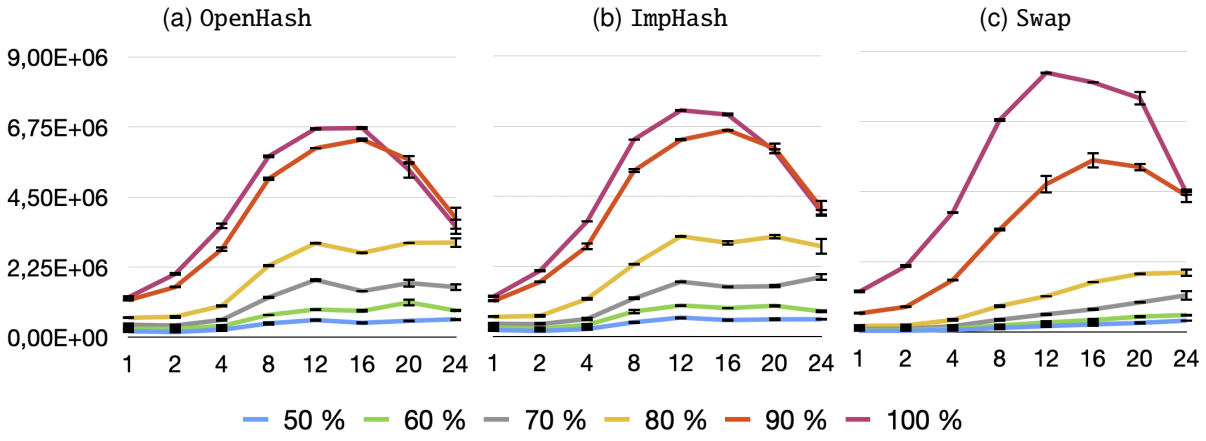
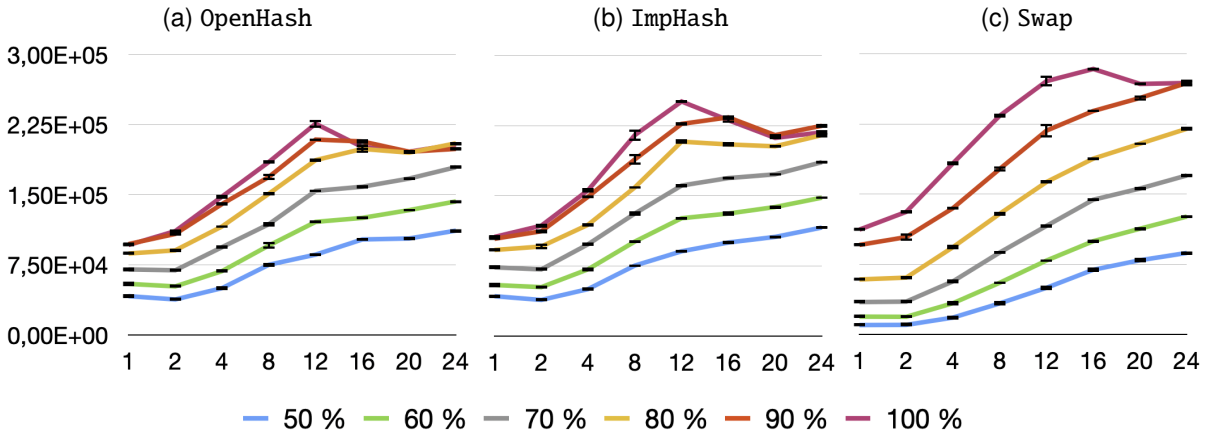


Figure 12 – TPC-C throughput (x1.000.000) for the `heavy-focus` workload with the three paging schemes.



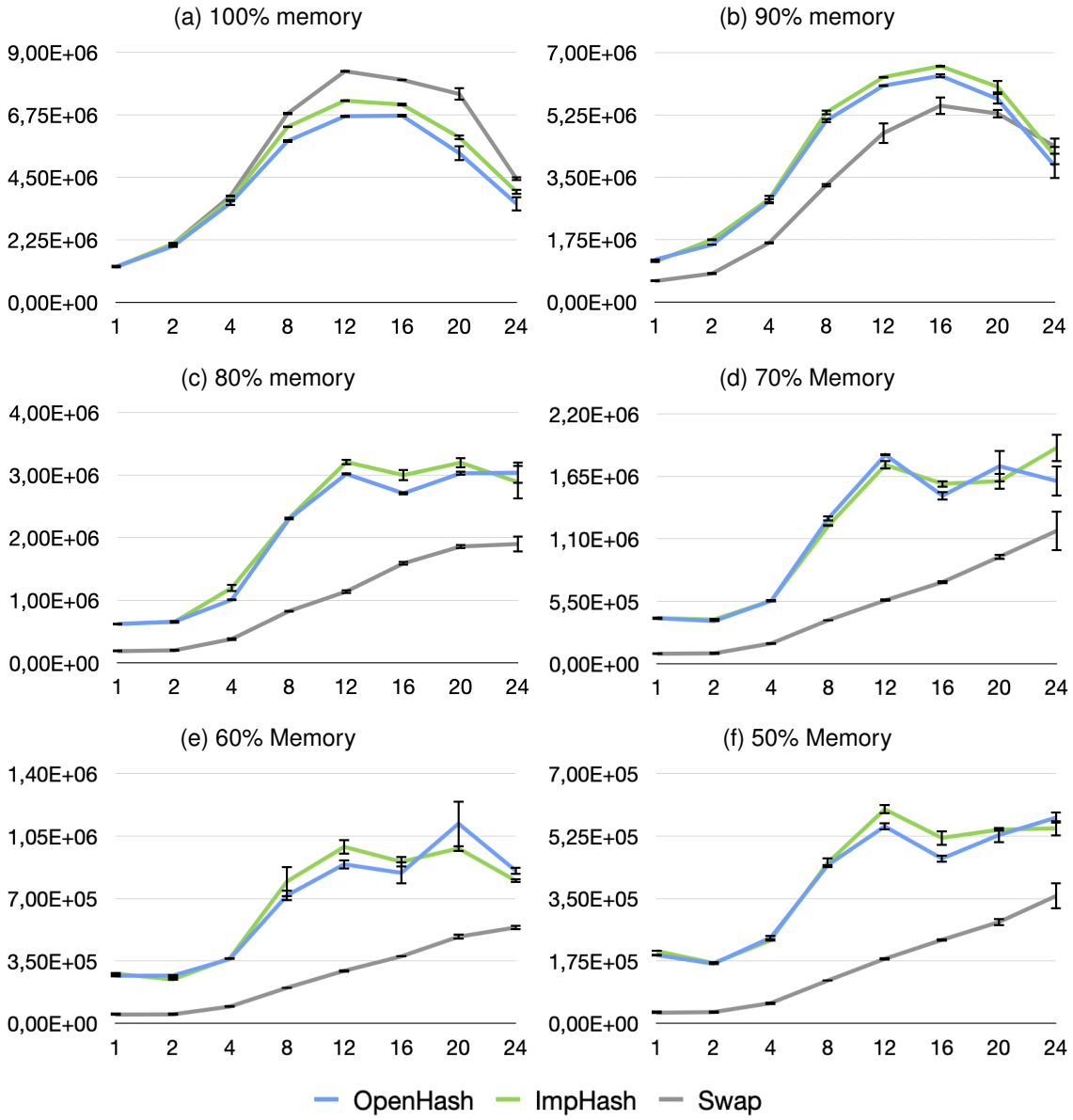
Overall, we can conclude from this section that the overhead caused by paging depends on the workload, and it can reach a point of no performance scalability around 50% of the total memory size for read-intensive workloads and 20% for more write-intensive ones. In both scenarios, the alternative paging schemes proposed by `OpenHash` and `ImpHash` tend to improve performance, as discussed in the next section.

4.3.3 HOW MUCH IMPROVEMENT DO `OpenHash` AND `ImpHash` OFFER?

This section seeks to characterize the performance improvements brought by the alternative paging schemes: `OpenHash` and `ImpHash`. In order to do that, we directly compare, for each workload and different working copy sizes, the throughput provided by them compared to a traditional swap-based scheme (`Swap`).

Figure 13 shows the results for the `light-focus` workload. In the scenario with no page faults (100% memory), as already discussed, alternative paging schemes suffer

Figure 13 – TPC-C throughput (x1.000.000) results for the `light-focus` workload with different memory percentages.



from the hashmap overhead. As the working copy size is reduced, we can notice that both alternative approaches perform better than `Swap`. This difference tends to increase as the size of the memory is decreased because the cost paid by `Swap` for the page faults (moving data to/from the swap area) is higher than the other approaches (basically, inserting the page timestamps into the hashmap). The plots also show that when the memory percentage decreases, particularly from 70% downwards, there is a higher performance variability with `OpenHash` and `ImpHash` with higher thread counts. This is because the increased number of page faults causes more transaction aborts and the measured throughput tends to oscillate more.

In order to better visualize the benefits of the alternative paging schemes, we compute

the slowdown of the three paging schemes as the working copy size is decreased relative to the version without any page faults. The slowdown is calculated for each thread number using the following formula: $sd_{i,p} = throughput_{i,100}/throughput_{i,p}$, where $sd_{i,p}$ is the slowdown with i threads at memory level $p\%$, and $throughput_{i,p}$ is the throughput at the same i thread count and memory level $p\%$. For example, $sd_{24,50}$ refers to the slowdown when using 24 threads with a memory size of 50% of the amount of memory that does not cause page faults.

The slowdown numbers are shown in Table 6. We also show the average slowdown numbers for the three paging schemes at different memory levels in the last row (AVG). As discussed earlier, when there are no page faults, both `OpenHash` and `ImpHash` present an extra overhead of managing the hashmap, particularly for higher thread counts. When page faults start to occur more frequently, the advantages of both `OpenHash` and `ImpHash` are more evident. At a memory level of 80%, the `Swap` scheme has an average slowdown of 6.8x, whereas `OpenHash` and `ImpHash` can reduce that to 2.7x and 2.6x, respectively. At the very extreme with 50%, the `Swap` approach reaches a slowdown of 44x versus 13.1x and 12.7x of `OpenHash` and `ImpHash`, an improvement of around 3.4x and 3.5x, respectively.

Although `ImpHash` consistently performs better than `OpenHash` in most scenarios, this difference is not very significant. In the best situation, at the 100% memory level, its slowdown is 1.1x against 1.2x of `OpenHash`, an improvement of 1.09x, or 9%. Recall that in this scenario, at a 100% memory level, there are no page faults, and therefore `ImpHash` does not suffer any overhead due to the consistency wait.

Table 6 – Performance slowdown with the `light-focus` workload for the different paging schemes (`OpenHash` (OH), `ImpHash` (IH), and `Swap` (S)) per working copy memory percentage.

THR	MEMORY PERCENTAGE																	
	100 %			90 %			80 %			70 %			60 %			50 %		
	OH	IH	S	OH	IH	S	OH	IH	S	OH	IH	S	OH	IH	S	OH	IH	S
1	1.0	1.0	1.0	1.1	1.1	2.2	2.1	2.1	6.8	3.2	3.2	14.6	4.8	4.6	25.8	6.7	6.3	42.7
2	1.0	1.0	1.0	1.3	1.2	2.6	3.2	3.2	10.5	5.6	5.4	23.0	7.9	8.6	41.5	12.6	12.5	66.4
4	1.1	1.0	1.0	1.4	1.3	2.3	3.8	3.2	10.1	6.9	6.9	21.4	10.6	10.5	40.5	15.9	16.4	67.3
8	1.2	1.1	1.0	1.3	1.3	2.1	3.0	2.9	8.2	5.3	5.6	17.7	9.5	8.5	34.0	15.3	15.1	56.7
12	1.2	1.1	1.0	1.4	1.3	1.8	2.8	2.6	7.3	4.5	4.7	14.9	9.3	8.4	28.2	15.1	13.9	46.0
16	1.2	1.1	1.0	1.3	1.2	1.5	3.0	2.7	5.0	5.4	5.1	11.1	9.5	8.8	21.3	17.3	15.4	34.2
20	1.4	1.3	1.0	1.3	1.2	1.4	2.5	2.3	4.0	4.3	4.7	8.0	6.7	7.6	15.4	14.2	13.8	26.4
24	1.3	1.1	1.0	1.2	1.1	1.0	1.5	1.5	2.3	2.8	2.3	3.8	5.2	5.6	8.3	7.7	8.1	12.5
AVG	1.2	1.1	1.0	1.3	1.2	1.9	2.7	2.6	6.8	4.8	4.7	14.3	7.9	7.8	26.9	13.1	12.7	44.0

Figure 14 shows the throughput results for `heavy-focus`. Notice that for this workload, the variance is much lower, since there is no abrupt increase in the number of page faults in the scenarios with higher thread counts. Like in the `light-focus` case, the benefits of alternative paging approaches are more evident when the size of the working copy memory decreases, particularly from 80% downward.

Table 7 shows the slowdowns achieved for each configuration. Here again we can see that the difference between `Swap` and the alternative schemes is not as high as with the `light-focus` workload. The biggest difference is at 50% memory level, when the average slowdown for `Swap` is 7.2x, versus 3.0x and 2.9x for `OpenHash` and `ImpHash`, respectively. The improvement therefore is 2.4x for `OpenHash` and 2.5x for `ImpHash`. The more discrete improvements with `heavy-focus` are explained by the fact that this workload is more write intensive, which overall results in lower throughput numbers and a higher percentage of the time is spent in SGL mode.

An important observation is that with a higher thread count, the difference between `Swap` and the alternative approaches tends to decrease. In these scenarios, the overhead of managing the hashmap is higher, as more transactions are adding the page timestamps to the hashmap, increasing the abort rate and the overhead.

Figure 14 – TPC-C throughput (x1.000.000) results for the `heavy-focus` workload with different memory percentages.

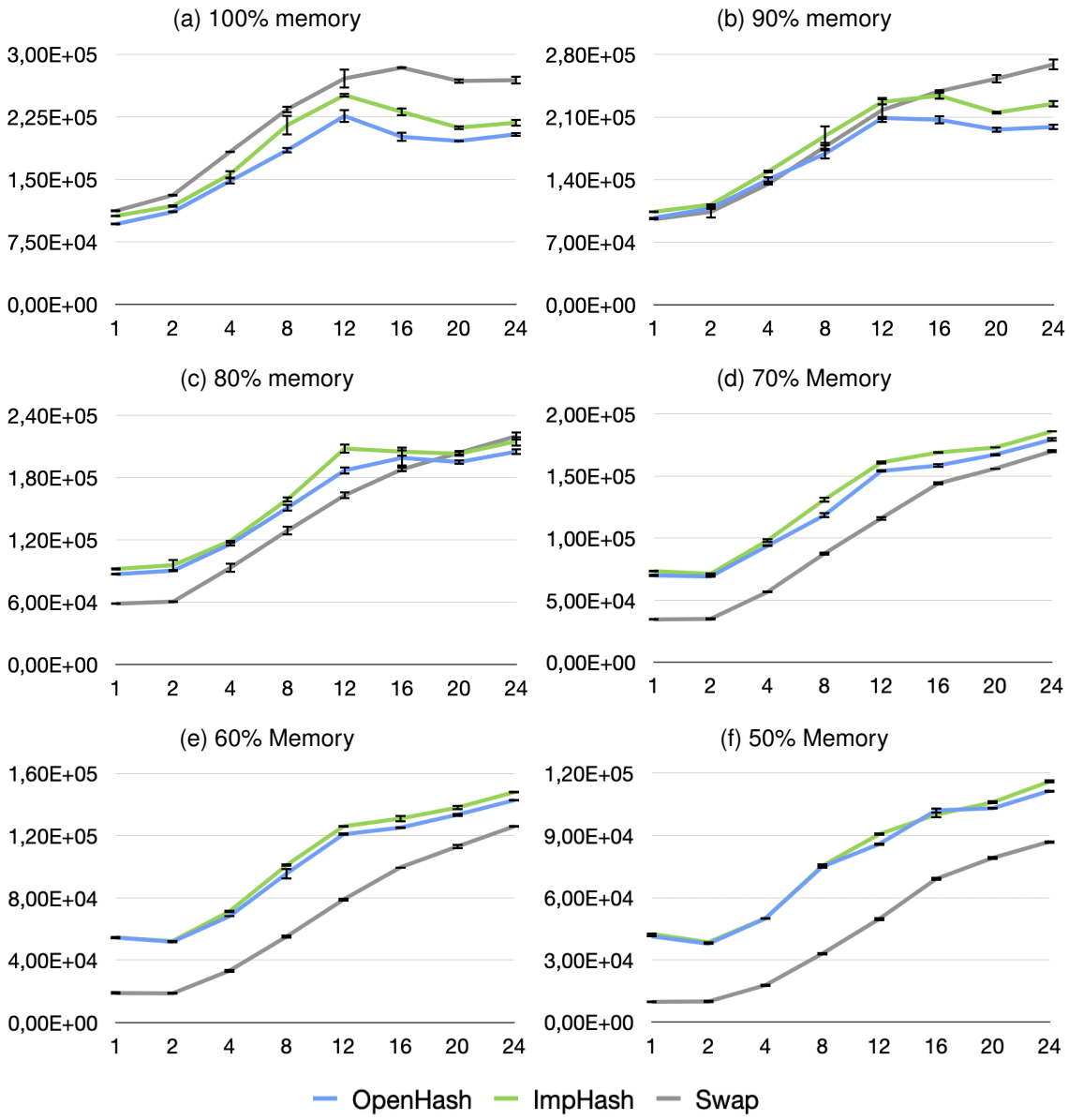


Table 7 – Performance slowdown with the `heavy-focus` workload for the different paging schemes (`OpenHash` (OH), `ImpHash` (IH), and `Swap` (S)) per working copy memory percentage.

THR	MEMORY PERCENTAGE																	
	100 %			90 %			80 %			70 %			60 %			50 %		
	OH	IH	S	OH	IH	S	OH	IH	S	OH	IH	S	OH	IH	S	OH	IH	S
1	1.2	1.1	1.0	1.2	1.1	1.2	1.3	1.2	1.9	1.6	1.5	3.3	2.1	2.1	6.0	2.7	2.6	11.4
2	1.2	1.1	1.0	1.2	1.2	1.3	1.4	1.4	2.2	1.9	1.8	3.8	2.5	2.5	7.0	3.5	3.4	13.1
4	1.2	1.2	1.0	1.3	1.2	1.4	1.6	1.5	2.0	2.0	1.9	3.2	2.7	2.6	5.5	3.7	3.7	10.3
8	1.3	1.1	1.0	1.4	1.2	1.3	1.5	1.5	1.8	2.0	1.8	2.7	2.4	2.3	4.2	3.1	3.1	7.1
12	1.2	1.1	1.0	1.3	1.2	1.2	1.4	1.3	1.7	1.8	1.7	2.3	2.2	2.2	3.4	3.2	3.0	5.5
16	1.4	1.2	1.0	1.4	1.2	1.2	1.4	1.4	1.5	1.8	1.7	2.0	2.3	2.2	2.9	2.8	2.8	4.1
20	1.4	1.3	1.0	1.4	1.2	1.1	1.4	1.3	1.3	1.6	1.5	1.7	2.0	1.9	2.4	2.6	2.5	3.4
24	1.3	1.2	1.0	1.4	1.2	1.0	1.3	1.3	1.2	1.5	1.4	1.6	1.9	1.8	2.1	2.4	2.3	3.1
AVG	1.3	1.2	1.0	1.3	1.2	1.2	1.4	1.4	1.7	1.8	1.7	2.6	2.3	2.2	4.2	3.0	2.9	7.2

5 CONCLUSION

Persistent memory (PM) is an emerging technology with promising features and performance. A major topic that has been neglected in the literature so far is paging support, that is, PM systems in situations where PM is much larger than DRAM.

This work presented two main contributions involving scenarios where paging is required: i) presented a detailed analysis of a state-of-the-art PM system, SPHT (CASTRO et al., 2021), in limited DRAM situations; and ii) proposed a flexible paging framework that does not require specific kernel and hardware requirements, relying solely on Linux signals and memory mapping system calls, which is not a common practice in the literature.

The results show that our implementation, when compared to the standard Linux OS swap mechanism, has its worst relative throughput performance when no paging is required (100% DRAM), with `OpenHash` (our worst performing implementation) reaching 40% slower with 20 threads in `light-focus` workload and 42% slower with 16 threads in `heavy-focus` workload, and `ImpHash` following suit, with a slight 10% performance advantage on average versus `OpenHash`.

Differently, when comparing situations with lower DRAM capacities with the optimal configuration (where no page outs occur - 100% memory) we can observe an increasingly higher performance degradation for `Swap` as the memory capacity is reduced compared to `OpenHash` and especially `ImpHash`. This behavior leads to `OpenHash` and `ImpHash` becoming the better performing implementations already with 90% memory, a difference which only increases further, the lower the memory capacity used in the cases here presented. When comparing the 50% memory results with 100% memory `Swap` implementation, it is possible to observe in the `light-focus` workload a staggering average performance slowdown of 44.0x for `Swap`, and much more reasonable slowdowns of 13.1x and 12.7x for `OpenHash` and `ImpHash`, respectively. This difference is also seen with the `heavy-focus` workload, with `Swap` having an average slowdown of 7.2x, while `OpenHash` and `ImpHash` have 3.0x and 2.9x, respectively. It should be noted that this small performance disparity between `OpenHash` and `ImpHash` is due to the simpler hashmap structure used in `ImpHash`, which is more optimal for the workloads tested, resulting in a small but consistent performance advantage.

In summary, relying on OS swap is not ideal, showing significant performance degradation across all DRAM constrained scenarios. The lack of portable paging frameworks led to the creation of our own implementation; our results showed that the use of our paging framework implementation is widely advantageous in DRAM-constrained situations in PM systems.

5.1 FUTURE WORKS

Our paging framework presented good overall performance over standard swap, but is not exempt from further optimizations. Our two main considerations are: i) further hashmap optimizations, continuing the work done with `ImpHash`, with more intelligent storage solutions, and ii) improvements to aborted hardware transactions, which here resort to the use of Single Global Lock (SGL) in order to maintain data consistency, implying in serialized workloads and, therefore, worse performance.

This work analyzed the performance of swap and our paging framework, but a broader comparison would add to this work. Other test scenarios of interest would be: 1) swap implementation using persistent memory instead of standard SSD; and 2) DudeTM on the same machine as the other experiments.

BIBLIOGRAPHY

BALDASSIN, A.; BARRETO, J.; CASTRO, D.; ROMANO, P. Persistent memory: A survey of programming support and implementations. *ACM Computing Surveys (CSUR)*, ACM New York, NY, USA, v. 54, n. 7, p. 1–37, 2021.

BEELER, B. *Intel Optane Persistent Memory 200 Series Review (MemVerge)*. 2021. [Online; Access in December, 19 2022]. Available in: <https://www.storagereview.com/review/intel-optane-persistent-memory-200-series-review-memverge>.

BELAY, A.; BITTAU, A.; MASHTIZADEH, A.; TEREI, D.; MAZIèRES, D.; KOZYRAKIS, C. Dune: safe user-level access to privileged cpu features. In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. [S.l.]: USENIX Association, 2012. p. 335–348. ISBN 9781931971966.

BELAY, A.; BITTAU, A.; MASHTIZADEH, A.; TEREI, D.; MAZIÈRES, D.; KOZYRAKIS, C. Dune: Safe user-level access to privileged {CPU} features. In: *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. [S.l.: s.n.], 2012. p. 335–348.

CALDWELL, B.; IM, Y.; HA, S.; HAN, R.; KELLER, E. *FluidMem: Memory as a Service for the Datacenter*. 2017. Disponível em: <https://arxiv.org/abs/1707.07780>.

CASTRO, D.; BALDASSIN, A.; BARRETO, J.; ROMANO, P. {SPHT}: Scalable persistent hardware transactions. In: *19th USENIX Conference on File and Storage Technologies (FAST 21)*. [S.l.: s.n.], 2021. p. 155–169.

CASTRO, D.; ROMANO, P.; BARRETO, J. Hardware transactional memory meets memory persistency. *Journal of Parallel and Distributed Computing*, Elsevier, v. 130, p. 63–79, 2019.

CHAKRABARTI, D. R.; BOEHM, H.-J.; BHANDARI, K. Atlas: Leveraging locks for non-volatile memory consistency. *ACM SIGPLAN Notices*, ACM New York, NY, USA, v. 49, n. 10, p. 433–452, 2014.

COBURN, J.; CAULFIELD, A. M.; AKEL, A.; GRUPP, L. M.; GUPTA, R. K.; JHALA, R.; SWANSON, S. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. *ACM SIGARCH Computer Architecture News*, ACM New York, NY, USA, v. 39, n. 1, p. 105–118, 2011.

CORREIA, A.; FELBER, P.; RAMALHETE, P. Romulus: Efficient algorithms for persistent transactional memory. In: *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*. [S.l.: s.n.], 2018. p. 271–282.

COUNCIL, T. P. P. *TPC-C Benchmark Revision 5.11. 0.(2018)*. 2018.

FRIDMAN, Y.; DESAI, S. M.; SINGH, N.; WILLHALM, T.; OREN, G. Cxl memory as persistent memory for disaggregated hpc: A practical approach. In: *Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*. New York, NY, USA: Association for Computing Machinery, 2023. (SC-W '23), p. 983–994. ISBN 9798400707858. Disponível em: <https://doi.org/10.1145/3624062.3624175>.

GENÇ, K.; BOND, M. D.; XU, G. H. Crafty: Efficient, htm-compatible persistent transactions. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. [S.l.: s.n.], 2020. p. 59–74.

GILES, E.; DOSHI, K.; VARMAN, P. Continuous checkpointing of htm transactions in nvm. *ACM SIGPLAN Notices*, ACM New York, NY, USA, v. 52, n. 9, p. 70–81, 2017.

GOGTE, V.; DIESTELHORST, S.; WANG, W.; NARAYANASAMY, S.; CHEN, P. M.; WENISCH, T. F. Persistency for synchronization-free regions. *ACM SIGPLAN Notices*, ACM New York, NY, USA, v. 53, n. 4, p. 46–61, 2018.

GOGTE, V.; KOLLI, A.; WENISCH, T. F. Data persistence. In: *A Primer on Memory Persistency*. [S.l.]: Springer, 2022. p. 13–20.

GU, J.; YU, Q.; WANG, X.; WANG, Z.; ZANG, B.; GUAN, H.; CHEN, H. Pisces: A scalable and efficient persistent transactional memory. In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: [s.n.], 2019. p. 913–928. ISBN 978-1-939133-03-8.

HALL, B.; BERGNER, P.; HOUSFATER, A. S.; KANDASAMY, M.; MAGNO, T.; MERICAS, A.; MUNROE, S.; OLIVEIRA, M.; SCHMIDT, B.; SCHMIDT, W. et al. *Performance optimization and tuning techniques for IBM Power Systems processors including IBM POWER8*. [S.l.]: IBM Redbooks, 2017.

HARRIS, T.; LARUS, J.; RAJWAR, R. Transactional memory. *Synthesis Lectures on Computer Architecture*, Morgan & Claypool Publishers, v. 5, n. 1, p. 1–263, 2010.

HERLIHY, M.; MOSS, J. E. B. Transactional memory: Architectural support for lock-free data structures. In: *Proceedings of the 20th annual international symposium on Computer architecture*. [S.l.: s.n.], 1993. p. 289–300.

HU, Q.; REN, J.; BADAM, A.; SHU, J.; MOSCIBRODA, T. {Log-Structured}{Non-Volatile} main memory. In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. [S.l.: s.n.], 2017. p. 703–717.

INTEL. *Memory Tiering: A New Approach to Solving Modern Data Challenges*. 2022. [Online; Access in January, 9 2023]. Available in: <https://www.intel.com/content/dam/www/central-libraries/us/en/documents/2022-04/optane-memory-tiering-white-paper.pdf>.

JACOBI, C.; SLEGEL, T.; GREINER, D. Transactional memory architecture and implementation for ibm system z. In: *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. [S.l.: s.n.], 2012. p. 25–36.

KATSARAGAKIS, M.; BALOUKAS, C.; PAPADOPOULOS, L.; KANTERE, V.; CATTHOOR, F.; SOUDRIS, D. Energy consumption evaluation of optane dc persistent memory for indexing data structures. In: *2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. [S.l.: s.n.], 2022. p. 75–84.

KWON, M.; JANG, J.; CHOI, H.; LEE, S.; JUNG, M. Training resilience with persistent memory pooling using cxl technology. In: IEEE. *Heterogeneous and Composable Memory Workshop at HPCA, 2023*. [S.l.], 2023.

LAVINGTON, S. The atlas story. In: *Atlas Symposium,*. [S.l.: s.n.], 2012. v. 6.

LIBÓRIO, A.; BALDASSIN, A.; CASTRO, D.; ROMANO, P.; BARRETO, J. A thorough analysis of page fault handling in persistent memory systems. In: SBC. *Simpósio em Sistemas Computacionais de Alto Desempenho (SSCAD)*. [S.l.], 2024. p. 85–96.

LIU, M.; ZHANG, M.; CHEN, K.; QIAN, X.; WU, Y.; ZHENG, W.; REN, J. Dudetm: Building durable transactions with decoupling for persistent memory. *ACM SIGPLAN Notices*, ACM New York, NY, USA, v. 52, n. 4, p. 329–343, 2017.

MOHAN, C.; LEVINE, F. Aries/im: an efficient and high concurrency index management method using write-ahead logging. *ACM Sigmod Record*, ACM New York, NY, USA, v. 21, n. 2, p. 371–380, 1992.

NARAYANAN, D.; HODSON, O. Whole-system persistence. In: *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*. [S.l.: s.n.], 2012. p. 401–410.

PATIL, O.; IONKOV, L.; LEE, J.; MUELLER, F.; LANG, M. Performance characterization of a dram-nvm hybrid memory architecture for hpc applications using intel optane dc persistent memory modules. In: *Proceedings of the International Symposium on Memory Systems*. New York, NY, USA: Association for Computing Machinery, 2019. (MEMSYS '19), p. 288–303. ISBN 9781450372060. Available in: <https://doi.org/10.1145/3357526.3357541>.

RAJWAR, R.; DIXON, M. Intel transactional synchronization extensions. In: *Intel Developer Forum San Francisco*. [S.l.: s.n.], 2012. v. 2012.

ROSNER, D. K.; SHOREY, S.; CRAFT, B. R.; REMICK, H. Making core memory: Design inquiry into gendered legacies of engineering and craftwork. In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. New York, NY, USA: Association for Computing Machinery, 2018. (CHI '18), p. 1–13. ISBN 9781450356206. Available in: <https://doi.org/10.1145/3173574.3174105>.

RUDOFF, A. *Deprecating the PCOMMIT Instruction*. 2016. [Online; Access in January, 9 2023]. Available in: <https://www.intel.com/content/www/us/en/developer/articles/technical/deprecate-pcommit-instruction.html>.

SCARGALL, S. *Programming persistent memory: A comprehensive guide for developers*. [S.l.]: Springer Nature, 2020.

SPECIFICATION, S. Tpc benchmark™ c. 1994.

USERFAULTFD. 2023. [Online; Access in October, 11 2023]. Available in: <https://www.kernel.org/doc/html/next/admin-guide/mm/userfaultfd.html>.

VOLOS, H.; TACK, A. J.; SWIFT, M. M. Mnemosyne: Lightweight persistent memory. *ACM SIGARCH Computer Architecture News*, ACM New York, NY, USA, v. 39, n. 1, p. 91–104, 2011.

WOO, Y.; KIM, T.; JUNG, S.; SEO, E. Analysis and optimization of persistent memory index structures' write amplification. *IEEE Access*, v. 9, p. 167687–167698, 2021.

WU, K.; REN, J.; PENG, I.; LI, D. ArchTM: Architecture-Aware, high performance transaction for persistent memory. In: *19th USENIX Conference on File and Storage Technologies (FAST 21)*. [S.l.: s.n.], 2021. p. 141–153. ISBN 978-1-939133-20-5.

WU, K.; REN, J.; PENG, I. B.; LI, D. Archtm: Architecture-aware, high performance transaction for persistent memory. In: *FAST*. [S.l.: s.n.], 2021. v. 21, p. 141–153.

WU, Z.; LU, K.; NISBET, A.; ZHANG, W.; LUJÁN, M. Pmthreads: Persistent memory threads harnessing versioned shadow copies. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. [S.l.: s.n.], 2020. p. 623–637.