Concurrency and Computation: Practice and Experience

WILEY

SPECIAL ISSUE PAPER OPEN ACCESS

An In-Depth Characterization of Page Fault Handling in Modern Persistent Memory Systems

André Libório[1] | Alexandro Baldassin[1] | Daniel Castro[2] | João Barreto[2] | Paolo Romano[2]

[1]DEMAC-IGCE, Universidade Estadual Paulista (UNESP), São Paulo, Brazil | [2]INESC-ID, Instituto Superior Técnico, Lisboa, Portugal

Correspondence: Alexandro Baldassin (alexandro.baldassin@unesp.br)

Received: 30 June 2025 | **Revised:** 19 November 2025 | **Accepted:** 10 March 2026

Keywords: page handling | persistent memory | transactional memory

ABSTRACT

Recent advancements in Persistent Memory (PM) technologies have enabled the integration of such devices directly into the processor's memory hierarchy, allowing them to be accessed via standard load/store instructions. These developments have revived interest in the design and implementation of systems capable of effectively supporting PM. A prominent approach adopted by several PM programming systems involves leveraging DRAM as a shadow memory to enable the use of modern hardware transactional mechanisms. While this technique offers performance benefits, it presents a critical limitation: when the available DRAM capacity is significantly smaller than that of the PM device, system performance may deteriorate due to excessive paging. Despite its practical implications, this issue remains underexplored in the literature. This article presents, to the best of our knowledge, the first comprehensive performance evaluation of PM systems under constrained DRAM availability. We begin by introducing a user-level page management framework that underpins our experimental methodology. Subsequently, we conduct a comparative analysis between traditional swap-based paging mechanisms and more advanced approaches that leverage the redo logs mechanisms of PM systems. Using the TPC-C suite as a representative benchmark, our experimental results demonstrate that specialized paging strategies can significantly mitigate performance degradation caused by excessive paging. In particular, we observe a decrease in performance loss of up to 3.5× in read-dominant workloads and up to 2.5× in write-intensive ones.

1 | Introduction

Persistent Memory (PM) represents a class of byte-addressable non-volatile memory technologies that retain data across power cycles, similar to storage-class devices such as SSDs and HDDs [1, 2]. However, PM differentiates itself by offering latency and throughput characteristics closer to DRAM, coupled with lower energy consumption [3]. The relevance of PM has grown significantly with the introduction of Intel's Optane DC Persistent Memory Modules (DCPMM), launched in 2019 in collaboration with Micron [4]. These modules deliver high-performance memory with substantially greater density than traditional DRAM solutions available in the server market [5–7]. Although Intel has since discontinued the Optane product line, the broader ecosystem has embraced emerging standards such as Compute Express Link (CXL) [8, 9], which includes provisions for byte-addressable persistent memory, signaling continued industry interest in this memory paradigm.

Despite its potential, programming persistent memory remains challenging due to a range of software-level concerns [10]. One fundamental difficulty stems from the volatility of processor caches: data written by applications may remain transiently cached and not yet durable in PM. Consequently, in the event of a crash (e.g., power failure), this discrepancy can lead to consistency violations. To ensure persistence, programmers

must explicitly invoke cache-flushing and fencing instructions to force the eviction of modified cache lines to the underlying PM. Even with these mechanisms, inconsistencies may still arise. For instance, in the case of inserting a node into a persistent linked list, a crash occurring between updating the previous node's pointer and linking the new node to its successor may leave the data structure in a corrupted state.

To mitigate these challenges, transactions have emerged as a central abstraction for programming with PM [11, 12]. A growing body of research has proposed systems that support transactional semantics on PM, including approaches such as those in DudeTM [13], cc-HTM [14], and SPHT [15]. These systems commonly incorporate two core design principles to enhance efficiency: (i) the use of hardware transactional memory (HTM) for concurrency control; and (ii) the use of DRAM as a shadow memory, combined with redo logging to persist updates asynchronously. A key assumption in these systems is that DRAM is as large as PM so that the shadow memory technique can be efficiently utilized, which might not always be the case. When DRAM capacity falls short relative to PM, a paging mechanism becomes necessary to manage memory residency between shadow and persistent pages. Among the existing systems, only DudeTM [13] has addressed this issue, and even then, its analysis was limited to constrained and synthetic benchmarks.

In order to shed some light on this issue, we present in this article a detailed and systematic analysis of the overheads and operational behavior of paging mechanisms in state-of-the-art PM systems. To this end, we design and implement a flexible user-level page handling framework aimed at evaluating system performance across varying configurations of shadow and persistent memory sizes, thereby inducing distinct paging behaviors for comprehensive analysis. We then integrate it into SPHT (Scalable Persistent Hardware Transactions) [15], a state-of-the-art PM system, to perform an in-depth characterization of how different page fault handling mechanisms affect PM system performance. While SPHT is used as our reference implementation, the proposed page handling mechanism is generic and can be incorporated into other PM systems with minimal adaptation.

In addition to evaluating a conventional swap-based page management strategy, we explore alternative approaches tailored for PM systems that leverage redo-log mechanisms [13]. In these approaches, page-out events do not trigger any immediate actions. Instead, when a previously evicted page is paged back into DRAM, the PM system ensures consistency by verifying whether the corresponding redo-log entries have been successfully replayed, thereby restoring the page to a coherent state. Using the TPC-C benchmark suite [16], our experimental evaluation demonstrates that alternative paging strategies can significantly mitigate performance degradation compared to traditional swap-based approaches. Specifically, we observe performance improvements ranging from 2.5× to 3.5×, depending on the read or write-dominant nature of the workload. To the best of our knowledge, this is the first study to systematically analyze and compare distinct page fault handling strategies in persistent memory systems. Although preliminary results were presented in our earlier work [17], this article provides a substantially more complete description of the proposed system. In particular, it introduces a conventional swap-based paging mechanism as a baseline and conducts a more comprehensive evaluation encompassing multiple paging strategies, PM-aware benchmarks, and a wider range of memory constraint scenarios. The source code and scripts used in the experiments are publicly available.[1]

In a nutshell, the main contributions of this article are:

- We implement a user-level page management mechanism for PM systems that works seamlessly on standard environments, unlike prior solutions requiring specialized hardware or custom kernels (Section 3);
- We analyze paging overheads in PM systems with shadow memory and hardware transactions, showing that alternatives to swap-based paging improve performance by up to 2.5× for write-intensive and 3.5× for read-intensive workloads (Section 4).

The remainder of this article is organized as follows: Section 2 reviews background and related work; Section 3 presents our user-level paging mechanism and its integration into SPHT; Section 4 analyzes paging scheme behavior; and Section 5 concludes this article.

2 | Background and Related Work

A majority of PM systems in the literature adopt transactions as their fundamental programming abstraction [10]. The transactional model, defined by the ACID properties (Atomicity, Consistency, Isolation, and Durability) [18], naturally aligns with the challenges of PM programming: it enables recovery from system crashes by rolling back incomplete operations and ensures that committed changes are durable. Early PM systems were largely inspired by research on Software Transactional Memory (STM) [19], which implements transactional semantics entirely in software through mechanisms such as data versioning and conflict detection. Representative systems from this category include Mnemosyne [11] and NV-Heaps [12].

With the advent of Hardware Transactional Memory (HTM) support in commercial processors by Intel and IBM during the 2010s [20, 21], new PM systems emerged that leverage hardware acceleration for transaction execution [13–15, 22, 23]. However, a key limitation of HTM is the lack of durability guarantees. As a result, these systems must incorporate additional software mechanisms to ensure data persistence following transaction completion. A common architectural feature of such systems is the use of DRAM as a *shadow memory* (also referred to as *working copy* in this article). Persistent pages are mapped to DRAM, and transactions execute on this volatile copy. To guarantee durability, each transaction maintains a redo log that records modifications, which are subsequently applied to the underlying persistent memory. Figure 1 shows the general architecture of a PM system that uses DRAM as shadow memory.

The execution model adopted by many PM systems relies on the maintenance of a working copy of persistent data, typically established through an operating system-supported copy-on-write (CoW) mechanism ①. This strategy is particularly important in systems leveraging HTM, as direct writes to persistent

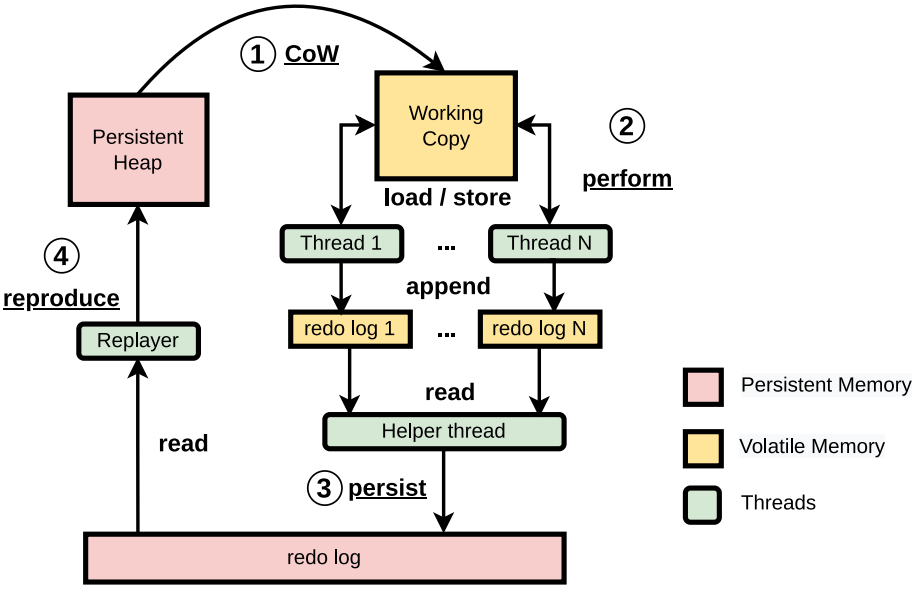

FIGURE 1 | General PM system with shadow memory [10]: Copy-on-Write creates a DRAM working copy of the persistent heap ①. Updates are logged ② and persisted ③ by working threads, and later reproduced ④ by background threads.

memory can trigger transaction aborts. Following the creation of the working copy, execution proceeds through three main stages – perform, persist, and reproduce – each responsible for progressively transitioning modifications from volatile to durable state. In the first stage, referred to as **perform** ②, modifications are performed directly on the working copy (volatile). Alongside these updates, each transaction maintains a volatile redo log, which is typically local to the thread in which it executes. This indirection ensures that transactional changes do not immediately affect the persistent state, preserving consistency in the event of a crash or abort. Once a hardware transaction successfully commits, altering the working copy in DRAM, the **persist** stage ③ is triggered. Recall that hardware transactions cannot change persistent data, and therefore it is in this step that the volatile redo logs are durably written to a dedicated region in persistent memory by helper threads. The job of the helper threads can be performed either synchronously (by the worker threads themselves) or asynchronously (by dedicated background threads), with different performance tradeoffs [24, 25]. Once the persist step is completed, the transaction's updates become persistent and recoverable, ensuring durability guarantees. The third and final stage, known as **reproduce** ④, involves applying the persisted redo logs to the persistent heap. This step can be executed asynchronously and is carried out by one or more replayer threads, which integrate the logged modifications into the global persistent heap.

2.1 | The Problem of Limited DRAM

It is important to note that the shadow copy mechanism employed by PM systems assumes that the available DRAM is sufficiently large to accommodate the process' working set. When this assumption does not hold, the operating system's default swapping mechanism is invoked, potentially leading to significant performance degradation. Despite its practical importance, this limitation is often overlooked by most existing PM systems implementing such execution models, which implicitly assume that memory capacity will not be a bottleneck. As a result, the system behavior under DRAM-constrained scenarios remains poorly understood in the literature, a gap this article aims to address.

To date, the only work that briefly acknowledges this issue is DudeTM [13]. The authors recognize the challenges posed by memory pressure and propose a custom alternative to the operating system's default swapping mechanism. Rather than writing evicted pages to the swap area during a page-out event, DudeTM discards the page entirely. This approach is semantically correct because any modifications to the pages are already captured in the redo logs and can be safely replayed to recover the lost data. However, if the discarded page is accessed again, the system must ensure that the associated redo logs have been fully replayed. To guarantee this, DudeTM associates a timestamp with each page and compares it to the progress of the replayer thread, which maintains a record of the most recently replayed timestamp. The motivation for this approach is to avoid the overheads of traditional swapping, while still ensuring correctness.

However, DudeTM provides limited experimental evidence to support the effectiveness of its proposed alternative paging mechanism. The evaluation is restricted to a simple microbenchmark consisting of a B+-tree key-value store accessed under a Zipfian distribution, with experiments conducted on a system equipped with 1GB of persistent memory (simulated) and shadow memory sizes varying between 1GB and 64MB. This narrow scope offers only a superficial assessment of the alternative paging mechanism behavior. Notably, the study does not include a comparison against the default operating system paging strategy, leaving the relative advantages of the proposed solution unquantified. Furthermore, it remains unclear how the system would perform under more complex or realistic workloads. In prior work [17], we presented preliminary results using a more representative benchmark to evaluate paging behavior in PM systems.

However, that study had several important limitations. First, it did not include an evaluation of a conventional swap-like paging mechanism, thereby omitting a critical baseline for comparison. Second, although the benchmark employed was more realistic than that used by DudeTM, it was not specifically designed with PM systems in mind, leading to performance anomalies that may have affected the validity of the results. Finally, the analysis did not explore system performance under varying degrees of memory pressure, which constrained the generalizability of the findings. In the present work, we address these shortcomings through a more comprehensive evaluation that incorporates multiple paging strategies, realistic and PM-aware benchmarks, and diverse memory constraint scenarios.

An additional challenge in evaluating paging mechanisms lies in the experimental infrastructure. Most academic datasets remain relatively small and can be entirely accommodated in DRAM, making it necessary to artificially constrain the amount of shadow memory to induce paging behavior. DudeTM addresses this by implementing its own software-based paging mechanism, and by offering an alternative approach built upon the Dune library [26], a framework that leverages Intel VT-x virtualization to allow user-space applications to manage page tables directly. However, Dune requires modifications to the Linux kernel and has not received updates since 2017, rendering it impractical for modern systems and software environments. Moreover, DudeTM's software-based approach necessitates the instrumentation of all memory loads and stores, which introduces non-trivial overhead, particularly problematic for systems relying on hardware transactions that suffer from limited memory capacity.

An alternative approach would involve the use of the `userfaultfd` interface [27], which enables user-space programs to handle page faults on specific virtual address ranges. However, `userfaultfd` lacks native support for page eviction handling. Although there have been efforts to extend it for this purpose, such as the work in FluidMem [28], these extensions are not part of the mainline kernel and are tied to specific kernel versions, limiting their portability and practical adoption. To address these limitations, this article also proposes an alternative technique for analyzing paging by transparently exploiting the page fault mechanism via user-level signal handlers, as discussed next.

3 | User-Level Paging Framework

As previously discussed, DudeTM's paging framework requires instrumentation of both load and store instructions, which introduces additional overhead along the critical path of hardware transactional execution. This limitation motivates the development of our own paging framework, designed specifically to avoid interference with the fast path of hardware transactions by eliminating the need for such instrumentation. We begin by presenting the overall architecture of our framework in Section 3.1, which enables the interception of page faults and explicit page management via the `mmap` system call. Subsequently, in Section 3.2, we describe how DudeTM's alternative paging strategy can be realized within this architecture. To demonstrate the practical applicability of our approach, we integrate the framework with SPHT [15], a state-of-the-art PM system that employs hardware transactional memory. Finally, Section 3.3 discusses some performance and correctness arguments, proposing a variant paging mechanism.

3.1 | Architecture

To design the paging framework without relying on specialized hardware support, such as Dune [26], or kernel-specific features, we developed a user-level solution based on standard Linux facilities such as signal handling and memory mapping system calls. Our framework combines the use of the `SIGSEGV` signal handler with the `mmap` and `munmap` system calls to manage virtual memory mappings dynamically. At the core of the implementation is a bitmap data structure that tracks the mapping status of each page within the managed memory region. Initially, all pages are unmapped. As page faults occur, intercepted through the `SIGSEGV` handler, the corresponding bit in the bitmap is updated, and the faulting page is explicitly mapped via a call to `mmap`.

The `SIGSEGV` handler operates by capturing the faulting memory address delivered by the operating system upon a segmentation fault. A key aspect of our design is the distinction between *real faults*, caused by erroneous memory accesses that should terminate execution, and *controlled faults*, which result from accesses to unmapped pages in the persistent memory region and are handled by the paging framework. To support this differentiation, the framework initializes and monitors a designated persistent memory region during setup. Two configurable parameters govern the memory environment: the working copy size (`wcs`) and the persistent heap size (`phs`). The `phs` parameter defines the total size of the persistent memory region, corresponding to the target application's persistent memory requirements (e.g., 512GB in our case). In contrast, the `wcs` parameter specifies the amount of DRAM allocated to serve as the working copy. Typically, `wcs` is set to a value smaller than `phs` to emulate scenarios in which DRAM is a constrained resource. By varying `wcs` and `phs`, the framework enables controlled experimentation across a wide range of memory pressure conditions, allowing us to evaluate the behavior and performance of PM systems under realistic and stress-tested configurations.

Figure 2 presents a simplified overview of the paging framework. During the initialization phase ①, the framework allocates and resets the page table bitmap, effectively marking all pages as unmapped. The total number of pages managed by the system is determined by the `phs` parameter, which defines the size of the persistent heap. The `wcs` parameter specifies the maximum number of pages that can be simultaneously mapped and thus governs the capacity of the shadow memory. These parameters are fixed at initialization time and cannot be modified during program execution.

As the application begins and memory addresses are accessed, resulting page faults are intercepted by the operating system and redirected to our framework ②, which handles page insertion ③. Upon receiving a fault, the framework checks whether there is available capacity within the shadow memory. If so, the corresponding page is marked as active in the bitmap, and the memory is mapped using the `mmap` system call. Page eviction ④

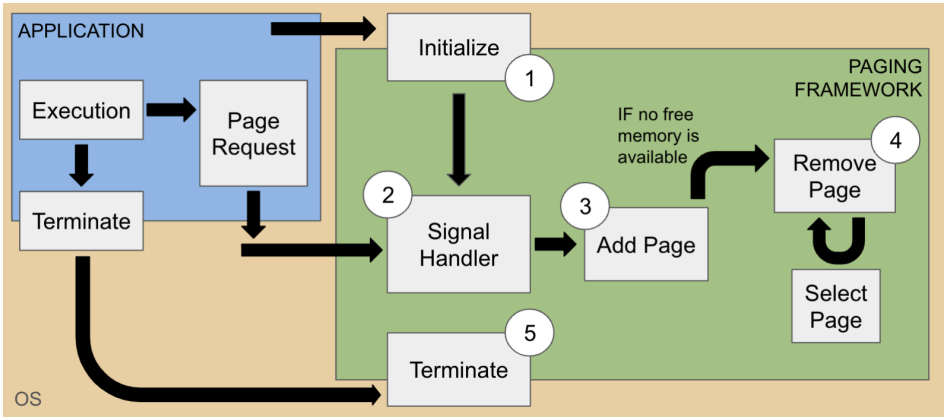

FIGURE 2 | Simplified paging framework overview. The persistent heap size (`phs`) and working copy size (`wcs`) are set at initialization ①. On a page fault ②, the page is mapped via `mmap` ③. If accesses exceed `wcs`, an LRU-like algorithm selects a page for removal ④. At termination, memory is freed and the signal handler is unregistered ⑤.

is managed based on two threshold parameters: `startth` and `stopth`. The `startth` threshold defines the utilization level at which page eviction should begin, while `stopth` determines the level at which eviction should cease. For instance, if the system supports 100 concurrently mapped pages, and `startth` and `stopth` are set to 90% and 85%, respectively, then once 90 pages are in use, the next page insertion will also initiate the eviction process. Pages are evicted using an LRU-like algorithm until the number of mapped pages falls to or below 85, ensuring enough room for new pages.

Although not explicitly depicted in Figure 2, the framework supports two modes for invoking the page eviction routine. In the *synchronous mode*, eviction is performed inline with the page insertion logic, using the same thread. In contrast, the *asynchronous mode* delegates eviction to a dedicated thread: when the `startth` threshold is reached, the insertion logic signals the eviction thread, which proceeds to reclaim pages until the number of mapped pages falls below the `stopth` threshold. The rationale for maintaining two distinct thresholds is to minimize unnecessary invocations of the eviction logic, particularly under high memory pressure. This is especially important in asynchronous mode, where synchronization between the worker threads and the eviction thread (a typical producer-consumer pattern) may introduce non-trivial overhead. The primary advantage of using a separate eviction thread is increased parallelism: the thread handling the page fault can resume execution immediately after triggering the eviction process. However, this approach incurs synchronization costs. Ultimately, the most effective strategy depends on the workload characteristics, a trade-off we analyze in Section 4.

3.2 | SPHT Integration

SPHT [15] adopts the general architecture common to modern HTM-based PM systems, as outlined in Section 2. It is composed of two main components: the Transaction Executor (TE) and the Log Replayer (LR). The TE is responsible for executing transactions across multiple worker threads. It utilizes the operating system's copy-on-write (CoW) mechanism to map the persistent heap into the process's virtual address space, thereby enabling transactional updates to DRAM before being persisted. The main feature of SPHT is the use of a group commit protocol, which aggregates transactions into batches to reduce commit latency and improve throughput. Each redo log is concluded with a commit marker that encodes the timestamp of the originating transaction. The LR is responsible for replaying the persisted redo logs into the persistent heap, ensuring that the system state remains consistent. To coordinate this process, it maintains a record of the timestamp of the most recently replayed transaction, or simply the *Most Recently Replayed Transaction Timestamp* (MRRTT). SPHT also guarantees *immediate durability* [24], ensuring that data changed by a transaction are durable once committed. It does this by executing the persist stage synchronously (as discussed in Section 2). Finally, as with all major transaction-based PM systems, SPHT acquires a Single Global Lock (SGL) whenever a transaction cannot be successfully committed in hardware (e.g., due to capacity limitations or due to operations that are illegal within the scope of a transaction), effectively serializing the transactions.

By default, SPHT does not incorporate any mechanism for explicitly managing paging. Consequently, when the shadow memory space is exhausted, the operating system's default swapping mechanism is triggered. Simulating this behavior within our paging framework is straightforward: we reserve a designated swap area on disk to store the contents of evicted pages. Upon a subsequent page fault, the framework checks whether the faulting page exists in the swap area. If so, its contents are simply restored into memory. It is important to note that employing such a swap-based mechanism does not compromise the consistency of the persistent heap. In the event of a system failure, recovery is guaranteed by replaying the persisted redo logs, irrespective of the state of the swap area. We note that the user-level implementation of the swap-based mechanism may overestimate the absolute cost of kernel-level swapping, as it incurs additional user-kernel transitions and system call overheads. Nonetheless, this design provides a controlled and repeatable environment in which all paging strategies are evaluated under identical conditions, ensuring that performance comparisons remain meaningful in relative terms.

ALGORITHM 1 | Simplified page management pseudo-code.

```
1: procedure ADD_PAGE(newpage)                          ▷ Invoked on a page fault
2:    while hashmap.getTimestamp(newpage) > MRRTT do    ▷ Wait until page is coherent
3:       Sleep some microseconds
4:    end while
5:    mmap(newpage)                                     ▷ Map the page to the address space
6:    used_space += 4096
7:    if used_space> startth then
8:       remove_page()                                  ▷ Invoke page eviction code
9:    end if
10: end procedure
```

Despite its correctness, the swap-based approach can introduce substantial overhead due to disk I/O incurred during page-in and page-out operations. To mitigate this, DudeTM [13] proposed an alternative paging strategy as discussed previously: evicted pages are simply discarded rather than written to disk, under the assumption that all modifications have been recorded in redo logs and will be applied during recovery. However, to safely remap such pages upon future access, it is necessary to ensure that their contents have already been replayed to the persistent heap. DudeTM addresses this requirement by associating a *touching ID* with each page, representing the timestamp of the last transaction that modified it. Before remapping a page, its touching ID is compared against the MRRTT, that is, the timestamp of the most recently replayed transaction. If the touching ID is greater than MRRTT, replay is incomplete and the remapping must be delayed until the condition is satisfied. This mechanism proves effective in scenarios where the waiting time for replay is minimal; however, DudeTM does not offer a quantitative comparison between this technique and the traditional swap-based approach.

Since SPHT does not include any optimization for page handling, we have adapted DudeTM's mechanism for evaluation. Specifically, we introduce a hashmap that records, for each page, the timestamp of the latest transaction that modified it. Prior to the logical commit of a transaction, we scan its redo log and update the corresponding page entries in the hashmap accordingly. When a page fault is intercepted, the framework consults the hashmap to retrieve the page's timestamp and compares it against the MRRTT maintained by the Log Replayer. If the page's timestamp exceeds the MRRTT, the thread blocks until the condition is no longer true, that is, until all prior modifications have been safely replayed. We refer to this approach as `OpenHash` throughout the remainder of this document.

This integration requires minimal changes to SPHT's original design. Specifically, it involves augmenting the transaction commit path to include the hashmap updates. The logic for handling a page fault is summarized in Algorithm 1. The procedure begins by verifying whether the timestamp of the faulting page exceeds the MRRTT (Line 2); if so, it waits until the condition is false. Once the page becomes safe to remap, it is allocated via `mmap()` (Line 5), the count of available mapped pages is incremented (Line 6), and the `startth` threshold is checked (Line 7). If the threshold is exceeded, the page eviction routine is triggered (Line 8). The nature of the eviction (synchronous or asynchronous) determines whether the calling thread or a background thread performs the removal.

3.3 | Performance and Correctness Considerations

Compared to a conventional swap-based paging mechanism, `OpenHash` eliminates the overhead associated with transferring pages to and from the swap area. However, the effectiveness of this approach hinges on two critical performance factors. First, the hashmap that records the most recent transaction that modified each page must be updated efficiently. These updates occur as the final step of a transaction prior to its logical commit. A key concern is that these modifications are executed within the context of the hardware transaction itself, thereby enlarging the transaction's write set. This increase in write footprint can elevate the risk of transaction aborts due to capacity overflows or conflicts, resulting in retries and degraded performance. Crucially, these updates must be performed within the hardware transaction to ensure correctness. Post-commit updates would reduce the transaction's write set but at the cost of potential consistency violations. This subtle correctness issue is not addressed in DudeTM. Indeed, inspection of DudeTM's available source code reveals that timestamp updates are carried out only after the redo logs are persisted, which is problematic. To illustrate why, consider a transaction whose redo logs have been persisted, but whose timestamp updates have not yet been recorded. If a page modified by this transaction is evicted and later remapped, for example, accessed by a new transaction, the paging mechanism will consult an outdated timestamp. Since the logs have not yet been replayed, the new transaction may proceed with stale data, violating consistency guarantees (i.e., the condition in Line 2 of Algorithm 1 fails to block the remapping). The second key factor for `OpenHash`'s performance is the latency associated with waiting for the Log Replayer to apply the required updates to persistent memory. If the replay process is slow, whether due to limited parallelism, backlog, or I/O contention, then transactions will be forced to stall during page remapping, potentially offsetting the benefits of bypassing the swap area.

The hashmap implementation in `OpenHash` adopts closed addressing [29] (i.e., open hashing) to resolve key collisions, storing key-value pairs (page address and timestamp) in linked lists associated with each bucket. While straightforward, this scheme may introduce pointer chasing and memory allocation overhead as more pages collide in a single bucket. Although open addressing (i.e., closed hashing) could alleviate memory allocation by avoiding dynamic list construction, it still increases the number of memory locations accessed, potentially enlarging the transaction's read set and making it more prone to aborts. To

address this issue, we developed an alternative strategy called `ImpHash`. This variant enforces a single timestamp per hashmap bucket, effectively allowing page aliasing. While multiple pages may map to the same bucket, only the most recent timestamp is retained. This ensures correctness, as the mechanism remains conservative: transactions will only proceed once the MRRTT surpasses the stored timestamp, regardless of whether it was associated with the exact page. The downside of this design is reduced precision, as it may lead to transactions waiting longer than strictly necessary (i.e., false positives in Line 2 of Algorithm 1).

4 | Experimental Analysis

In order to understand the tradeoffs of the different approaches to page handling in PM systems, we seek to answer the following questions in this section:

- How much overhead is caused by paging? (Section 4.2).
- How much overhead is added by `OpenHash` and `ImpHash`? (Section 4.3).
- How much improvement do `OpenHash` and `ImpHash` offer? (Section 4.4).

4.1 | Experimental Setting

All experiments were conducted on a server equipped with an Intel(R) Xeon(R) Gold 5317 processor (24 physical cores), 256GB of DRAM, and 512GB of Intel Optane DC Persistent Memory (Intel Persistent Memory 200 Series), running Ubuntu Server 20.04. The number of application threads varied between 1 and 24. For each experiment, after system initialization and workload configuration, all persistent pages were explicitly unmapped prior to workload execution. A warm-up phase was executed to ensure the system reached a steady state. Performance metrics were collected over a subsequent 10-s measurement window. For each combination of workload, thread count, and memory constraint, we measured system throughput in terms of transactions per second. Each configuration was executed three times, and we report the average performance along with the standard deviation in the plots. Given the relatively long duration of each run (10 s), we determined that three repetitions provided a reliable estimate of behavior. This corresponds to the statistical stability of executing 15 repetitions over 2-s windows, yielding a comparable total measurement time of 30 s.

In order to assess the performance and effectiveness of the different paging schemes, it is important to use a relevant benchmark. The TPC-C [16] is an On-Line Transaction Processing (OLTP) benchmark commonly employed for analyzing PM systems [30, 31] 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. The TPC-C has the distinct advantage of being built from the ground up with transactions in mind, perfectly fitting the requirements of transaction-based PM systems. The benchmark has transactions that are intended to execute 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 [32].

Different workloads can be generated by changing the percentage of operations performed in the warehouses. We chose to use two broad categories:

1. `read-dominant` 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;
2. `write-intensive` 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 `read-dominant` workload predominantly perform lighter read operations, the `write-intensive` configuration is more read-write intensive. We have used the default parameters defined in the TPC-C specification [16] for the initial population per warehouse, including the number of customers (30,000), districts (10), and items (100,000). In our experiments, a total of 32 warehouses were used. This number determines the overall dataset size and memory pressure: larger configurations increase paging activity, but the relative performance trends among the evaluated mechanisms remain consistent.

The following paging mechanisms are evaluated:

1. `OpenHash`: In this scheme, no action is taken during a page-out event: the page is simply discarded, and its modifications are not immediately applied to the persistent heap. On a page-in event, the system ensures that the page reflects the latest committed state by checking whether the log replayer has already replayed all associated updates. To achieve this, a hashmap with 500,000 entries is maintained to track, for each page, the timestamp of the last transaction that modified it;
2. `ImpHash`: This is an optimized variant of `OpenHash` that improves insertion performance by allowing multiple pages to map to the same hashmap bucket, avoiding the need for linked-list chaining (as used in closed addressing). While this approach reduces overhead during timestamp updates, it introduces a potential performance penalty during page-in events, as it may force unnecessary waiting when unrelated pages share a bucket with a more recently modified page;
3. `Swap`: This scheme emulates the default behavior of the operating system's virtual memory subsystem. During a page-out event, the page contents are written to a dedicated swap area (e.g., SSD). Upon a page-in event, the page is restored from swap storage. This mechanism guarantees correctness via full state restoration but introduces significant I/O overhead.

Memory utilization levels are controlled via two parameters: the total persistent heap size (`phs`) and the working copy size (`wcs`). In all experiments, `phs` is held constant at a value larger than `wcs`, ensuring that the persistent memory footprint remains stable while allowing us to vary the size of the DRAM-based shadow memory. To systematically induce page faults, we progressively decrease `wcs`, thereby increasing memory pressure on the working copy region. For each workload, we first empirically determine the minimum value of `wcs` that avoids any page faults. This configuration serves as the 100% memory usage baseline (in the order of some gigabytes). Subsequently, we reduce `wcs` to simulate constrained DRAM scenarios, effectively triggering page evictions and faults in a controlled manner. In the experimental evaluation, we consider `wcs` values ranging from 100% down to 50% of the baseline, enabling a detailed analysis of system behavior under varying memory pressure. To ease the analysis of the results and enhance the reproducibility of our results, we also assume isolated execution, in which memory pressure is induced only within the user-level managed region.

The thresholds used to trigger the start (`startth`) and stop (`stopth`) of page eviction were empirically set to 97% and 95% of the working copy capacity, respectively. A sensitivity analysis revealed that this configuration offered a favorable balance between responsiveness and overhead. In a preliminary study, we also evaluated both synchronous and asynchronous page eviction strategies. Our findings indicate that the synchronous approach consistently yielded better performance. Although asynchronous eviction theoretically allows the eviction thread to operate concurrently with the application threads, this advantage was rarely observed in practice. In most cases, application threads encountering page faults were forced to wait on the eviction thread due to the required synchronization. The added conditional signaling and coordination overhead often outweighed any potential gains from parallelism. Therefore, the experimental results shown in the rest of this section are for the synchronous page eviction.

4.2 | How Much Overhead is Caused by Paging?

In evaluating paging behavior, an ideal experimental setup would allow for direct control over the number of page faults to systematically assess system performance under varying memory pressure. However, directly regulating page fault frequency is not straightforward, as it depends on application access patterns and OS-level memory management. To address this, we adopt the working copy size (`wcs`) as a proxy control parameter. By proportionally varying `wcs`, we indirectly influence the frequency of page faults: as `wcs` decreases, the available DRAM for shadow memory becomes more constrained, thereby increasing the likelihood and frequency of page faults. This approach enables us to approximate and study the system's behavior across a spectrum of memory-limited scenarios. Figure 3 shows the absolute number of page-out events as `wcs` decreases, from 90% to 50%, for each of the evaluated paging strategies and workloads.

The results confirm that using `wcs` as a proxy for indirectly controlling the number of page faults is a valid and effective strategy. For both `read-dominant` and `write-intensive` scenarios, a significant increase in page fault frequency is observed as `wcs` decreases, particularly up to the 50% threshold, beyond which the growth rate of faults begins to plateau. While the overall trend is consistent across both workloads, notable differences emerge in the absolute number of faults. Specifically, `read-dominant` consistently generates a higher number of page faults compared to `write-intensive`. This behavior is attributed to the read-intensive nature of `read-dominant`, which not only increases its sensitivity to page availability but also results in higher throughput. In contrast, `write-intensive` generates a relatively higher number of write operations and longer transactions, characteristics that tend to increase contention and abort rates in `OpenHash` and`ImpHash`. Consequently, its overall throughput is lower, which in turn dampens the number of page faults relative to `read-dominant` for equivalent memory constraints. It is also worth noting that, for each workload, `OpenHash` and `ImpHash` exhibit very similar behavior in terms

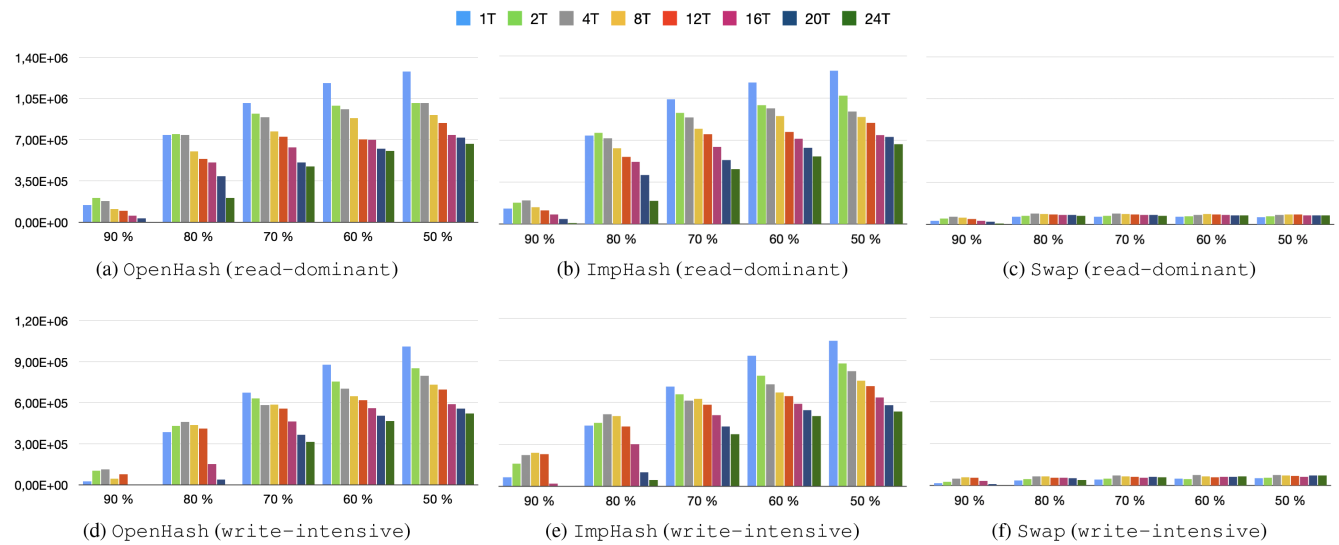

FIGURE 3 | Number of page faults for different degrees of memory pressure and thread count. Results are for the `read-dominant` (top) and `write-intensive` (bottom) workloads with the three studied paging schemes.

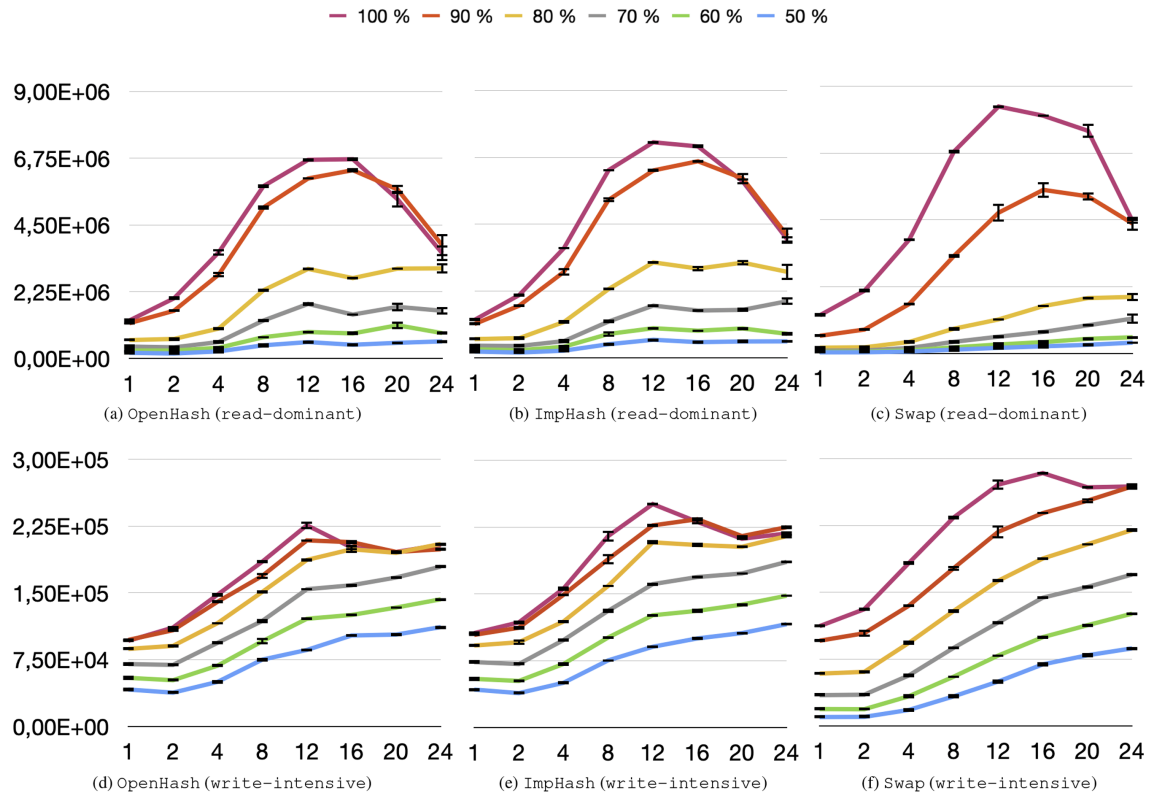

FIGURE 4 | TPC-C throughput for both `read-dominant` (top) and `write-intensive` (bottom) workloads with the three studied paging schemes and different degrees of memory pressure.

of page fault frequency. As the number of threads increases, the total number of page faults generally decreases, likely due to improved parallelism and more efficient page reuse across concurrent transactions.

Note that the y-axis scale in the figures remains the same across all paging schemes for a given workload to emphasize the substantial difference in page fault behavior exhibited by `Swap`. When a page fault occurs, the transactions have to acquire a Single Global Lock (SGL), which serializes the execution. The duration for which the lock remains held is notably longer in `Swap` due to the disk I/O involved in swap operations. In contrast, the lightweight page discard strategy used in `OpenHash` and `ImpHash` results in shorter lock hold times and, consequently, less contention. As a result, `Swap` tends to execute fewer transactions overall and accesses a smaller portion of the persistent heap, thereby reducing the absolute number of page faults. However, this apparent reduction in faults is not indicative of improved efficiency but rather a byproduct of degraded throughput and reduced memory access diversity.

The performance of the three paging strategies is strongly correlated with the number of page faults incurred. Figure 4a–c presents the overall system throughput for the `read-dominant` scenario under varying degrees of memory pressure. A notable trend is that, for this read-intensive workload, throughput begins to degrade beyond 16 threads, particularly for scenarios in which memory pressure is not as high (i.e., 100% and 90%). This degradation can be attributed to the increase in parallelism, which in turn exacerbates memory contention and leads to more transaction aborts. Across all three paging schemes, throughput declines significantly as `wcs` is reduced from 90% to 80%, and again from 80% to 70%. Below 50%, the system exhibits minimal performance gains with additional threads, indicating a saturation point beyond which paging overhead restricts parallelism due to the need of SGL-based synchronization. It can also be noted that the impact of paging is most severe under the `Swap` scheme. These results reinforce the conclusion that swap-based paging introduces substantial overhead, especially in workloads characterized by high read throughput and frequent memory access.

A contrasting behavior is observed for the `write-intensive` scenario, as depicted in Figure 4d–f. In comparison to `read-dominant`, the absolute throughput values are significantly lower (approximately an order of magnitude) due to the more write-intensive nature of the transactions. Notably, there is no sharp performance drop beyond 16 threads; instead, the system exhibits a gradual plateauing in throughput. This behavior aligns with the earlier page fault analysis, which showed that the total number of page faults grows more modestly in this workload. From Figure 4d–f, it is also evident that throughput degradation under memory pressure is more gradual than with the `read-dominant`. The system effectively reaches a point of scalability collapse under `write-intensive` only when `wcs` is reduced to approximately 20% of the total memory (not shown in the plots), indicating that write-intensive workloads are more resilient to moderate paging overhead than read-intensive ones.

In summary, these findings reinforce that the impact of paging overhead is workload-dependent. For read-dominated workloads, performance scalability becomes severely impaired when

wcs is reduced below 50% of available memory. Conversely, for write-heavy workloads, acceptable performance levels are maintained until wcs falls below 20%. In both cases, the alternative paging strategies embodied by OpenHash and ImpHash consistently mitigate performance degradation, as analyzed in more depth in the following sections.

4.3 | How Much Overhead is Added by OpenHash and ImpHash?

Two primary sources of overhead are associated with the OpenHash and ImpHash paging schemes: (i) the computational cost of annotating each modified page with a timestamp prior to transaction commit (i.e., inserting timestamps into the hashmap); and (ii) the waiting time incurred during page-in events to ensure the consistency of the reloaded page (hereafter referred to as *consistency wait*). To isolate and characterize the first source of overhead, we conducted an experiment in which no page faults are generated, thereby eliminating any consistency wait effects. Figure 5 presents the resulting throughput for both read-dominant and write-intensive workloads. As expected, the ideal case (depicted by the gray line) yields the highest throughput, as it avoids both sources of overhead.

In both workloads, OpenHash and ImpHash exhibit lower throughput relative to the ideal case, with the performance gap attributable solely to the cost of inserting timestamps into the hashmap. Notably, ImpHash, which simplifies the hashmap by maintaining only the most recent timestamp per bucket (thus allowing page aliasing), consistently outperforms OpenHash, which resolves bucket collisions using linked lists. The benefits of this optimization are twofold: it shortens the transaction duration, thereby reducing the likelihood of transaction conflicts, and it decreases the transaction's write footprint, potentially preventing capacity aborts. Capacity aborts are caused when the number of writes made by a transaction is greater than the hardware capacity to store them.

In the case of read-dominant, the overhead introduced by OpenHash reaches up to 40% at 20 threads, while ImpHash demonstrates a 26% reduction in throughput relative to the ideal case. Below 8 threads, performance differences are negligible (approximately 5%), while the average performance variation between OpenHash and ImpHash across all thread counts is roughly 10%. For the write-intensive scenario, the overhead effects are similarly pronounced: OpenHash is up to 42% slower than the ideal at 16 threads, while ImpHash remains at 26% overhead with 20 threads. However, unlike in read-dominant, even low thread counts (1–4) demonstrate more noticeable performance degradation under write-intensive. This can be attributed to the more write-intensive nature of the workload, which increases the number of pages that must be timestamped, thereby amplifying the insertion overhead, even under low contention. As the number of threads grows, the likelihood of transactional conflicts also increases. Since the hashmap insertions performed by both OpenHash and ImpHash extend the length of the transactions, the hashmap itself is a contention point. This dynamic provides further insight into the nature and magnitude of the overhead introduced by the timestamp management process.

The second source of overhead, consistency wait, is quantified in Table 1, which presents the proportion of time spent waiting for a page to become consistent (Lines 2–3 of Algorithm 1) relative to the total time required to insert a new page into memory (entire add_page procedure). The table reports this overhead as a function of the number of threads and wcs (from 100% to 50% of the total persistent heap), across the read-dominant (left) and write-intensive (right) scenarios. The overhead is influenced by the number of pages written by a transaction (which determines the volume of timestamp insertions in the hashmap) and the hashmap scheme used, either OpenHash (OH) or ImpHash (IH). In the ideal case (100%), there should be no consistency wait, as no page-outs occur and all pages mapped are newly allocated, eliminating the need for consistency checks. While this behavior holds true for OpenHash, the data in the table reveal a nonzero overhead in the 100% configuration for ImpHash. This discrepancy stems from the aliasing effect inherent to ImpHash's design. Since ImpHash employs a fixed-size hashmap with a single timestamp per bucket, a newly mapped page may collide with an existing bucket entry that was previously used by a different page. Although functionally correct, since the stored timestamp is always conservative, this can result in a spurious consistency wait, as the page may appear stale despite being newly allocated. The impact of this aliasing effect is minor but measurable: for read-dominant, the consistency wait ranges from 1.8% (1 thread) to 0.3% (12 threads), while for write-intensive, it varies from 4.5% (1 thread) to 2.4% (24 threads). The slightly higher overhead observed in write-intensive reflects the increased volume of writes,

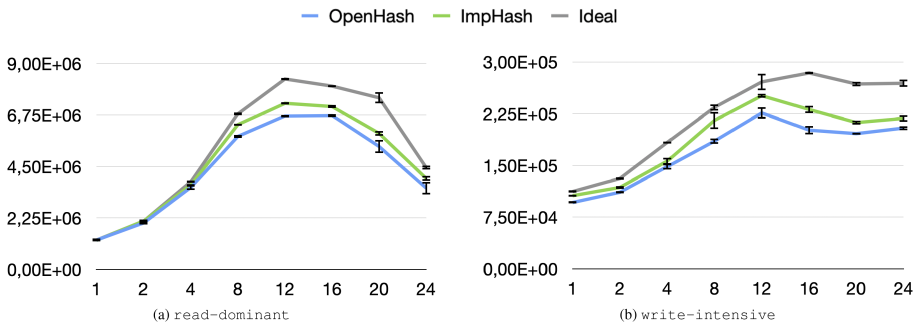

FIGURE 5 | TPC-C throughput in an ideal scenario (no page faults) showing the overhead of the alternative paging schemes, OpenHash and ImpHash, with the read-dominant (left) and write-intensive (right) workloads.

TABLE 1 | Percentage of time waiting for a consistent state of a newly mapped page relative to the total time of `add_page`.

	wcs fraction (read-dominant)												wcs fraction (write-intensive)											
	100%		90%		80%		70%		60%		50%		100%		90%		80%		70%		60%		50%	
#t	OH	IH	OH	IH	OH	IH	OH	IH	OH	IH	OH	IH	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	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	0.4	1.3	1.6	0.9	1.0	0.6	0.8	0.5	0.5	0.4	0.5	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	1.7	1.3	1.6	0.9	1.1	0.6	0.7	0.5	0.6	0.4	0.4	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	1.3	1.2	1.1	0.9	1.1	0.6	0.8	0.4	0.6	0.4	0.5	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	0.3	1.1	1.4	0.8	1.1	0.6	0.8	0.4	0.5	0.5	0.5	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	1.0	1.0	1.2	0.7	0.9	0.5	0.7	0.5	0.6	0.3	0.4	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	0.4	0.8	1.0	0.7	0.9	0.5	0.7	0.4	0.6	0.4	0.4	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	0.5	0.4	0.7	0.7	0.8	0.5	0.6	0.4	0.4	0.3	0.4	0.0	2.4	0.0	3.2	0.0	1.5	0.6	0.7	0.5	0.6	0.5	0.5

Note: The table shows the results for both `OpenHash` (OH) and `ImpHash` (IH), and both workloads: `read-dominant` (left) and `write-intensive` (right). The results are shown for different `wcs` fractions (from 100% to 50%). Cell colors indicate value intensity.

which leads to more frequent hashmap insertions and thus a greater probability of bucket collisions.

As the memory pressure increases, page faults become more frequent, thereby increasing the total duration of the `add_page` procedure. Since the consistency wait remains roughly constant while the overall page insertion time increases (due to more frequent invocations of the page eviction logic in Line 8 of Algorithm 1), the relative impact of the wait diminishes. In summary, the results indicate that the consistency wait overhead is consistently small, and its presence in `ImpHash` is outweighed by the performance benefits of its faster, alias-tolerant hashmap design. Consequently, `ImpHash` remains a worthwhile optimization, as it effectively mitigates the primary source of paging overhead (timestamp insertion) while incurring only minimal consistency wait penalties.

4.4 | How Much Improvement do `OpenHash` and `ImpHash` Offer?

This section aims to quantify the performance benefits of the alternative paging strategies, `OpenHash` and `ImpHash`, by directly comparing their throughput against the conventional swap-based approach, `Swap`, across varying degrees of memory pressure and workloads. Figure 6 presents the results for the `read-dominant` workload. In the baseline scenario with no page faults (100% memory availability), the alternative schemes underperform compared to `Swap` due to the overhead introduced by maintaining the timestamp hashmap, as discussed earlier. However, as `wcs` is gradually reduced, thereby increasing the frequency of page faults, the performance of `OpenHash` and `ImpHash` surpasses that of `Swap`. This performance gap widens as memory becomes more constrained, reflecting the disproportionately high cost of managing page-outs and page-ins in `Swap` (i.e., transferring data to/from the swap area) relative to the more lightweight operations of the alternative schemes (i.e., updating timestamps in a hashmap). Moreover, the plots indicate that performance variability becomes more pronounced for `OpenHash` and `ImpHash` at lower memory levels (70% and below), especially as the number of threads increases. This behavior can be attributed to the elevated rate of page faults, which leads to more transaction aborts and greater fluctuation in throughput across runs.

Overall, these findings highlight that although the alternative schemes incur a fixed bookkeeping overhead, they scale more gracefully under memory pressure and avoid the high latency costs of disk I/O associated with `Swap`. To better illustrate the advantages of the alternative paging schemes, we compute the slowdown experienced by each scheme as `wcs` is reduced, using the configuration without any page faults (i.e., 100% memory) as the performance baseline. The slowdown is calculated for each thread count and memory configuration using the formula $sd_{i,p} = throughput_{i,100}/throughput_{i,p}$, where $sd_{i,p}$ denotes the slowdown for i threads at a memory level of p%. For instance, $sd_{24,50}$ represents the slowdown observed when using 24 threads with `wcs` reduced to 50% of the baseline configuration.

Table 2 presents the resulting slowdown values for the `read-dominant` workload. The table also includes an average (AVG) row showing the average slowdown across all thread counts for each paging scheme and memory level. As discussed earlier, even in the absence of page faults (100% memory), `OpenHash` and `ImpHash` exhibit some additional overhead due to the management of the hashmap, particularly at higher thread counts. However, once page faults begin to occur more frequently, the benefits of the alternative schemes become increasingly evident. At the 80% memory level, the average slowdown for `Swap` is 6.8×, whereas `OpenHash` and `ImpHash` reduce this to 2.7× and 2.6×, respectively. In the most extreme case, memory level at 50%, the slowdown of `Swap` escalates to 44× on average, while `OpenHash` and `ImpHash` limit the impact to 13.1× and 12.7×, respectively. These results represent performance improvements of approximately 3.4× and 3.5× compared to the traditional swap-based scheme. Although `ImpHash` generally outperforms `OpenHash`, the margin is relatively modest. For example, in the 100% memory scenario, the slowdown for `ImpHash` is 1.1×, compared to 1.2× for `OpenHash`, an improvement of 9%. It is important to note that in this scenario, no consistency wait overhead is incurred, which helps isolate the benefits of `ImpHash`'s optimized hashmap insertion strategy.

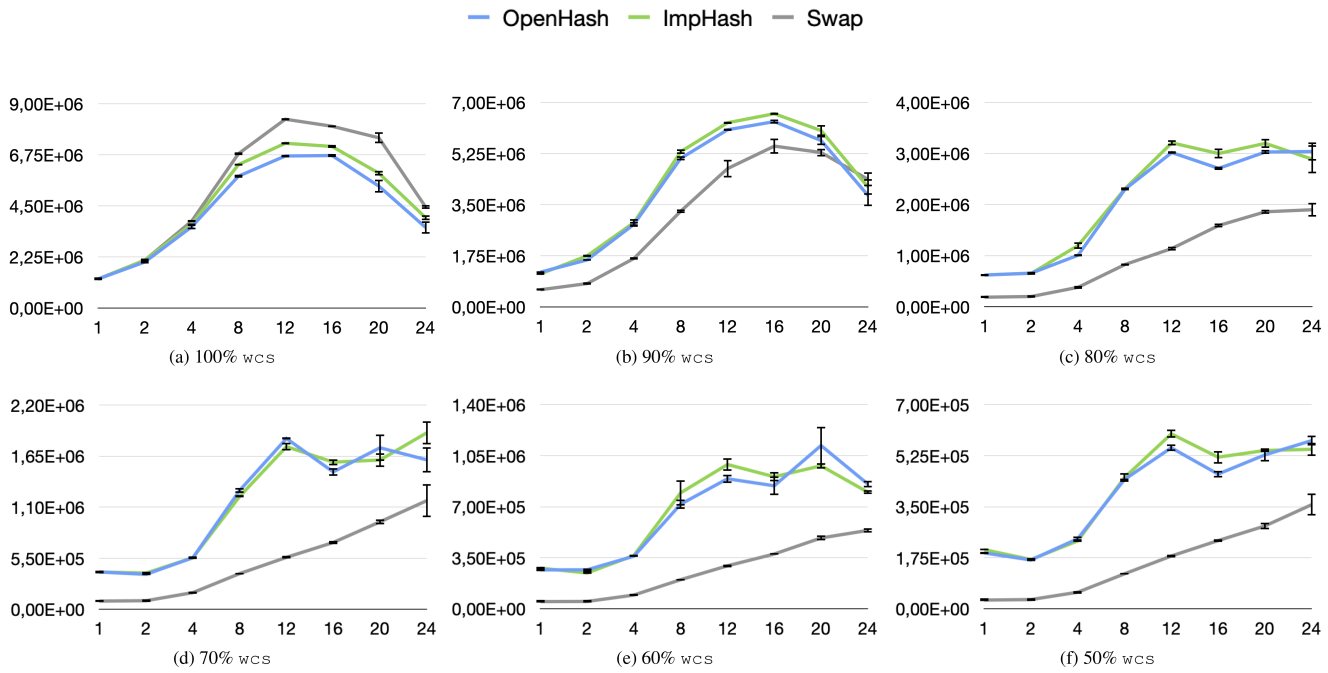

FIGURE 6 | TPC-C throughput results for the `read-dominant` workload with different degrees of memory pressure.

TABLE 2 | Performance slowdown with the `read-dominant` workload for the different paging schemes (`OpenHash` (OH), `ImpHash` (IH), and `Swap` (S)) per `wcs` fraction.

	wcs fraction																	
	100%			**90%**			**80%**			**70%**			**60%**			**50%**		
#t	**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**

Note: Cell colors indicate value intensity.

Figure 7 presents the throughput results for the `write-intensive` workload. Compared to `read-dominant`, this scenario exhibits much lower performance variability, primarily due to the absence of a steep increase in page faults even as the number of threads increases. As in the previous case, the benefits of the alternative paging mechanisms become increasingly evident as the available working copy memory is reduced, particularly from the 80% level downward.

Table 3 summarizes the corresponding slowdown values. While the performance advantages of `OpenHash` and `ImpHash` over `Swap` are still observable, the gap is less pronounced than in the `read-dominant` case. The most notable difference occurs at the 50% memory level, where the average slowdown for `Swap` reaches 7.2×, compared to 3.0× and 2.9× for `OpenHash` and `ImpHash`, respectively. These results reflect performance gains of 2.4× and 2.5×, respectively, for the alternative schemes. The more discrete improvements with `write-intensive` 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. A notable trend is that as the thread count increases, the performance gap between `Swap` and the alternative paging schemes tends to diminish. This is due to the increased contention on the hashmap: with more concurrent transactions updating page timestamps, the hashmap becomes a scalability bottleneck. This leads to a higher transaction abort rate and amplifies the overhead introduced by the additional bookkeeping required by `OpenHash` and `ImpHash`.

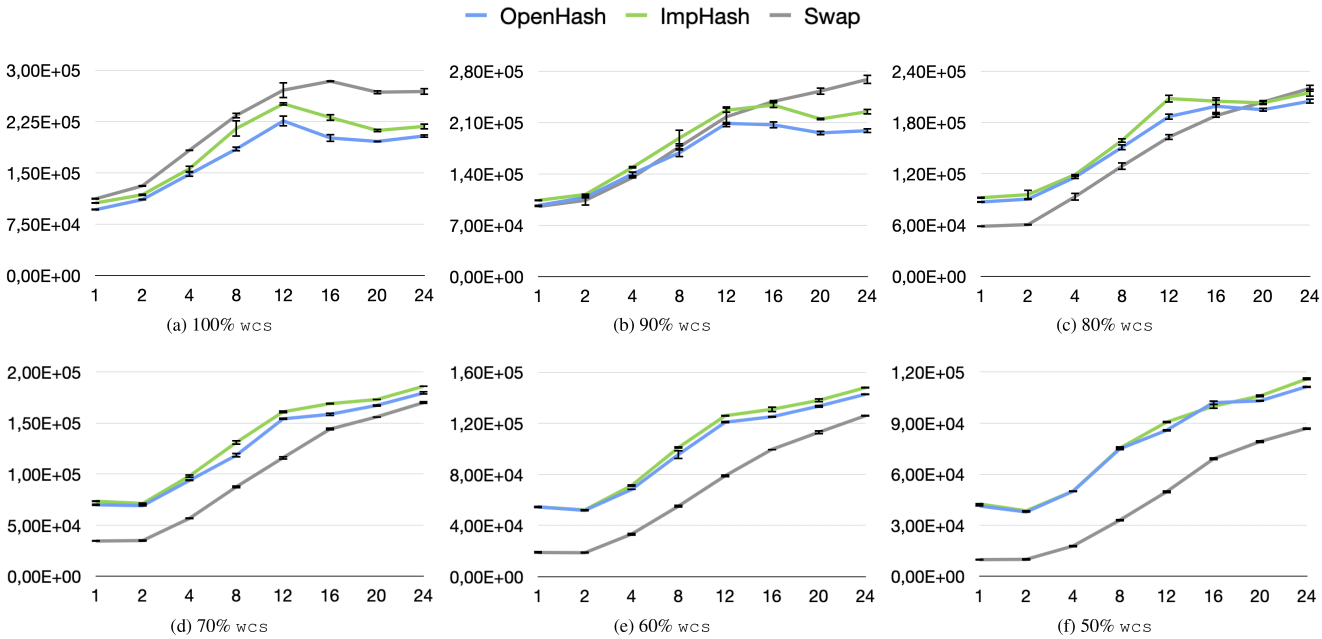

FIGURE 7 | TPC-C throughput results for the `write-intensive` workload with different degrees of memory pressure.

TABLE 3 | Performance slowdown with the `write-intensive` workload for the different paging schemes (`OpenHash` (OH), `ImpHash` (IH), and `Swap` (S)) per `wcs` fraction.

	wcs fraction																	
	100%			**90%**			**80%**			**70%**			**60%**			**50%**		
#t	**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**

Note: Cell colors indicate value intensity.

Overall, the results demonstrate consistent performance advantages for the alternative paging schemes. While some overhead is observed in the absence of page faults (i.e., at the 100% memory level), both `OpenHash` and `ImpHash` offer significant performance improvements under increasing memory pressure. In particular, under the most constrained scenario, 24 threads and `wcs` of 50%, the alternative schemes yield speedups of up to 3.4× and 3.5× for `read-dominant`, and 2.4× and 2.5× for `write-intensive`, when compared to the conventional `Swap` approach.

5 | Conclusion

This article explored the performance implications of paging mechanisms in PM systems, with a focus on scenarios where DRAM is insufficient to hold the full working copy of persistent data. Although important for practical use of modern PM systems, this aspect has not been adequately explored in the literature before, thus making our contribution of utmost importance. In order to provide an in-depth analysis of the performance of PM systems under different memory pressure scenarios, we proposed a user-level paging framework capable of simulating DRAM limitations by intercepting page faults. This framework supports alternative paging policies, including a timestamp-based scheme (`OpenHash`) inspired by DudeTM and an optimized version (`ImpHash`) that reduces overhead by simplifying the hashmap structure. Our experimental evaluation, based on the SPHT system and TPC-C-style workloads, demonstrates that the alternative paging schemes outperform the conventional swap-based approach, especially under memory pressure. For the read-dominated workloads, we observed speedups of up to

3.5× over swap when DRAM was limited to 50% of the persistent heap. The improvements were also consistent for write-intensive workloads, though more modest, reaching up to 2.5× due to its more write-intensive nature. Ultimately, this work highlights the importance of considering memory pressure in PM system design and provides a practical mechanism to evaluate the paging behavior in PM systems.

This study opens several avenues for future investigation. First, while our experiments focused on isolated execution to ensure repeatable performance measurements, extending the framework to interact with system-wide memory management would enable analysis under realistic multi-process environments. Second, although our user-mode emulation provides a controlled comparison of paging strategies, future work could incorporate fully transparent kernel-level mechanisms to more precisely quantify the cost of native swap operations. Finally, the user-level paging framework itself could be extended to support adaptive or hybrid policies that dynamically adjust eviction and consistency strategies according to workload characteristics.

Acknowledgments

The Article Processing Charge for the publication of this research was funded by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) (ROR identifier: 00x0ma614).

Funding

This work was supported by the São Paulo Research Foundation (FAPESP) (Grants 2018/15519-5 and 2022/11704-8), National Council for Scientific and Technological Development (CNPq) under Grant 303669/2022-5, and by national funds through Fundação para a Ciência e a Tecnologia, I.P. (FCT) under projects UID/50021/2025 and UID/PRR/50021/2025, through the Portuguese Recovery and Resilience Plan under project C645008882-00000055 (Center for Responsible AI), and through the EU's Horizon Europe research and innovation programme under Grant Agreement No. 101189689 (ACHILLES project).

Conflicts of Interest

The authors declare no conflicts of interest.

Data Availability Statement

The source code and data that support the findings of this study are openly available in https://github.com/andreliborio98/spht_paging.

Endnotes

[1] https://github.com/andreliborio98/spht_paging.

References

1. A. Badam, "How Persistent Memory Will Change Software Systems," *Computer* 46, no. 8 (2013): 45–51.

2. R. F. Freitas and W. W. Wilcke, "Storage-Class Memory: The Next Storage System Technology," *IBM Journal of Research and Development* 52, no. 4.5 (2008): 439–447.

3. O. Patil, L. Ionkov, J. Lee, F. Mueller, and M. Lang, "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* (Association for Computing Machinery, 2019), 288–303.

4. M. Tyson, "Intel Optane DC Persistent Memory Launched," (2019), https://hexus.net/tech/news/storage/129143-intel-optane-dc-persistent-memory-launched/.

5. I. B. Peng, M. B. Gokhale, and E. W. Green, "System Evaluation of the Intel Optane Byte-Addressable NVM," in *Proceedings of the International Symposium on Memory Systems* (Association for Computing Machinery, 2019), 304–315.

6. J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An Empirical Guide to the Behavior and Use of Scalable Persistent Memory," in *18th USENIX Conference on File and Storage Technologies* (USENIX Association, 2020), 169–182.

7. L. Xiang, X. Zhao, J. Rao, S. Jiang, and H. Jiang, "Characterizing the Performance of Intel Optane Persistent Memory: A Close Look at Its On-DIMM Buffering," in *Proceedings of the Seventeenth European Conference on Computer Systems* (Association for Computing Machinery, 2022), 488–505.

8. M. Jung, "Hello Bytes, Bye Blocks: PCIe Storage Meets Compute Express Link for Memory Expansion (CXL-SSD)," in *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems* (Association for Computing Machinery, 2022), 45–51.

9. D. Das Sharma, R. Blankenship, and D. Berger, "An Introduction to the Compute Express Link (CXL) Interconnect," *ACM Computing Surveys* 56, no. 11 (2024): 1–37.

10. A. Baldassin, J. Barreto, D. Castro, and P. Romano, "Persistent Memory: A Survey of Programming Support and Implementations," *ACM Computing Surveys* 54, no. 7 (2021): 1–37.

11. H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight Persistent Memory," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Association for Computing Machinery, 2011), 91–104.

12. J. Coburn, A. M. Caulfield, A. Akel, et al., "NV-Heaps: Making Persistent Objects Fast and Safe With Next-Generation, Non-Volatile Memories," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Association for Computing Machinery, 2011), 105–118.

13. M. Liu, M. Zhang, K. Chen, et al., "DudeTM: Building Durable Transactions With Decoupling for Persistent Memory," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Association for Computing Machinery, 2017), 329–343.

14. E. Giles, K. Doshi, and P. Varman, "Continuous Checkpointing of HTM Transactions in NVM," in *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management* (Association for Computing Machinery, 2017), 70–81.

15. D. Castro, A. Baldassin, J. Barreto, and P. Romano, "SPHT: Scalable Persistent Hardware Transactions," in *19th USENIX Conference on File and Storage Technologies* (USENIX Association, 2021), 155–169.

16. S. T. Leutenegger and D. Dias, "A Modeling Study of the TPC-C Benchmark," *ACM Sigmod Record* 22, no. 2 (1993): 22–31.

17. A. Libório, A. Baldassin, D. Castro, P. Romano, and J. Barreto, "A Thorough Analysis of Page Fault Handling in Persistent Memory Systems," in *Anais do XXV Simpósio em Sistemas Computacionais de Alto Desempenho* (SBC, 2024), 85–96.

18. J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, 1st ed. (Morgan Kaufmann, 1992).

19. N. Shavit and D. Touitou, "Software Transactional Memory," in *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing* (Association for Computing Machinery, 1995), 204–213.

20. Intel, "Intel; Architecture Instruction Set Extensions Programming Reference," (2020).

21. H. Le, G. Guthrie, D. Williams, et al., "Transactional Memory Support in the IBM POWER8 Processor," *IBM Journal of Research and Development* 59, no. 1 (2015): 1–14.

22. D. Castro, P. Romano, and J. Barreto, "Hardware Transactional Memory Meets Memory Persistency," *Journal of Parallel and Distributed Computing* 130 (2019): 63–79.

23. K. Genç, M. D. Bond, and G. H. Xu, "Crafty: Efficient, HTM-Compatible Persistent Transactions," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Association for Computing Machinery, 2020), 59–74.

24. J. Izraelevitz, H. Mendes, and M. L. Scott, "Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model," in *Proceedings of the International Symposium on Distributed Computing* (Springer Berlin Heidelberg, 2016), 313–327.

25. R. M. Krishnan, J. Kim, A. Mathew, et al., "Durable Transactional Memory Can Scale With TimeStone," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Association for Computing Machinery, 2020), 335–349.

26. A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis, "Dune: Safe User-Level Access to Privileged CPU Features," in *10th USENIX Symposium on Operating Systems Design and Implementation* (USENIX Association, 2012), 335–348.

27. The Kernel Development Community, "Userfaultfd," (2024), https://www.kernel.org/doc/html/latest/admin-guide/mm/userfaultfd.html.

28. B. Caldwell, Y. Im, S. Ha, R. Han, and E. Keller, "FluidMem: Memory as a Service for the Datacenter," *CoRR* (2017) abs/1707.07780.

29. T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. (MIT Press, 2009).

30. J. Gu, Q. Yu, X. Wang, et al., "Pisces: A Scalable and Efficient Persistent Transactional Memory," in *2019 USENIX Annual Technical Conference* (USENIX Association, 2019), 913–928.

31. K. Wu, J. Ren, I. Peng, and D. Li, "ArchTM: Architecture-Aware, High Performance Transaction for Persistent Memory," in *19th USENIX Conference on File and Storage Technologies* (USENIX Association, 2021), 141–153.

32. Council TPP, "TPC BENCHMARK C–Standard Specification 5.11," (2010).