2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 978-1-6654-2235-2/20/$31.00 ©2021 IEEE | DOI: 10.1109/HPCA51647.2021.00022

Stealth-Persist: Architectural Support for Persistent Applications in Hybrid Memory Systems

Mazen Alwadi[1], Vamsee Reddy Kommareddy[1], Clayton Hughes[2], Simon David Hammond[2], Amro Awad[3]
University of Central Florida[1], Sandia National Laboratories[2], North Carolina State University[3]
{mazen.alwadi, vamseereddy8}@knights.ucf.edu, {chughes,sdhammo}@sandia.gov, ajawad@ncsu.edu

***Abstract*—Non-volatile memories (NVMs) have the characteristics of both traditional storage systems (persistent) and traditional memory systems (byte-addressable). However, they suffer from high write latency and have a limited write endurance. Researchers have proposed hybrid memory systems that combine DRAM and NVM, utilizing the lower latency of the DRAM to hide some of the shortcomings of the NVM – improving system's performance by caching resident NVM data in the DRAM. However, this can nullify the persistency of the cached pages, leading to a question of trade-offs in terms of performance and reliability. In this paper, we propose Stealth-Persist, a novel architecture support feature that allows applications that need persistence to run in the DRAM while maintaining the persistency features provided by the NVM. Stealth-Persist creates the illusion of a persistent memory for the application to use, while utilizing the DRAM for performance optimizations. Our experimental results show that Stealth-Persist improves the performance by 42.02% for persistent applications.**

***Index Terms*—persistent applications, hybrid memories**

I. Introduction

Emerging non-volatile memories (NVMs) are maturing up to a level close to mass production stage and wide adoption [11], [15], [16], [59], [65]. For instance, very recently, Intel released the Optane DC product, which is a memory module that uses 3D XPoint technology [2]. These NVM-based memory modules operate at ultra-low idle power but still have very high densities. For example, each Optane DC module can have 512GB of capacity [3]. Therefore, they present a very compelling addition to servers where high memory capacities are needed but may be power constrained. While DRAM modules must perform frequent, costly, refresh operations, NVMs do not, which eliminates a large percentage of the power consumption [38]. Additionally, NVMs retain data after power failure or shutdown, making them very useful for crash-consistent applications [9], [17] and hosting fast access filesystems [52]. However, the read and write latencies of emerging NVMs are multiple times slower than those of DRAM. For instance, on Intel's Optane DC, the read latency is 300ns, whereas DRAM has read latency close to 70ns [46] – a 4.3x slower read accesses. While write latencies of NVMs can be hidden through external buffering or by leveraging battery-backed internal write buffers (e.g., Intel's Write Pending Queue), the device write latencies of NVMs can be tens of times slower than those of DRAM [13], [33].

The persistence feature of emerging NVMs, is attractive for many applications where data recovery and crash consistency are critical [40]. Moreover, emerging NVMs allow direct access and updates to persistent files without incurring expensive page faults [52]. For example, databases of hundreds of gigabytes can reside on NVM, and can be read and written directly through applications with conventional load/store operations, similar to DRAM [52], [59]. Moreover, Intel's Persistent Memory Development Kit (PMDK) allows developing applications that leverage NVM's persistence to make durable updates for critical data structures [9]. Ideally, after a crash, persistent applications should be able to recover by reading their data structures from the NVM [41]. However, the high read/write latency of NVMs can significantly slow down accesses to such persistent data. In other words, persistent applications need to choose between using the slow (relative to DRAM) NVM that enables crash consistency, or using DRAM and lose the persistence capabilities in the memory subsystem. With the increasing adoption of emerging NVMs, along with the increasing accessibility of persistent programming libraries (e.g., PMDK), we expect more and more applications will utilize the persistence feature of NVMs. Therefore, improving the performance of such applications while still ensuring persistence of data is a crucial design point.

Emerging NVMs can be integrated as storage devices (e.g., inside Solid-State Drives), such as Intel's Optane Drive [1] or as part of the system's memory hierarchy. For integrating NVMs into the memory hierarchy, there are several standards and options [2], [3], [6]. Most notably, Intel's DIMM-like NVM modules (called Optane DC [3]) can be integrated either as the main memory, or as a part of the main memory along with other memory options (e.g. DRAM and HBM). When used as a part of the main memory, it can be exposed as a separate physical memory address range extending the physical address range of DRAM, or the DRAM can be used as a hardware-managed cache of the physical range of the Optane DC [3]. The former is called *application direct mode*, which is similar to exposing different memory zones to the system in Non-Uniform Memory Architectures (NUMA), whereas the latter is called *memory mode*. Memory mode gives up on the persistence feature, as memory blocks could be updated in the volatile DRAM when applications flush their updates from internal caches. However, since DRAM

Part of this work was done when Vamsee and Mazen were working under the supervision of Amro Awad at UCF. Amro Awad is now with the ECE Department at NC State. Mazen and Vamsee have equal contributions in this work, i.e., equally first co-authors of this paper.

978-1-6654-2235-2/21/$31.00 ©2021 IEEE
DOI 10.1109/HPCA51647.2021.00022

caches a large number of the NVM pages, it significantly improves the access latency, especially for frequently-used pages. On the other hand, application direct mode ensures persistence of pages mapped to the NVM address range, but incurs significant latencies as it relies on the capacity-limited internal processor caches (not the external DRAM). Therefore, the current integration options for Optane memory modules as (part of) the main memory ignores the performance of persistent applications that require both persistence and high performance (e.g., cacheability in DRAM).

JEDEC also provides several standards for memory modules containing NVMs. In particular, JEDEC defines three different standards for DIMMs containing NVMs (called NVDIMM), namely NVDIMM-N, NVDIMM-P, and NVDIMM-F [5], [6]. The three different options provide different exposed capacity, persistency guarantees, and management complexity trade-offs. In particular, NVDIMM-N only exposes the DRAM to the software and utilizes a supercapacitor to power the DIMM during a crash, providing the capability to copy DRAM data to the NVM (currently flash-based). Therefore, NVDIMM-N has exposed latencies similar to DRAM but limits the memory capacity to the DRAM size. NVDIMM-F exposes the NVM (currently flash-based) to the software and is accessed directly as a block device. Meanwhile, NVDIMM-P is more broadly defined for different NVM technologies and allows internal DRAM caching within the NVDIMM-P with several persistence options, such as *deep flush* commands in case the NVDIMM-P is not energy-backed. One main advantage of NVDIMM-P is that it leverages a transaction protocol, which allows it to use non-deterministic timing as opposed to NVDIMM-N and NVDIMM-F that rely on deterministic timing. Clearly, among the three options, NVDIMM-P is the most suitable for emerging NVMs (not tailored for flash) and high capacity systems. Moreover, with sufficient energy-backing on the DIMM, the internal DRAM can be thought of as a persistent fast cache of NVM inside the NVDIMM-P module. NVDIMM-P, without energy-backing, is similar to using Intel's Optane DC in memory mode except that the DRAM cache is inside the NVDIMM-P, not independent module as in memory mode.

While NVDIMM-P is interesting as a concept, the fast and persistent caching capability for large NVM capacity is limited due to the following reasons. First, if tens or hundreds of gigabytes of DRAM is needed to efficiently cache large NVMs, then bulky and potentially expensive battery support is needed to provide energy-backing. Moreover, with the energy-backing only supporting the internal DRAM, customers are limited to the same vendor and the specific capacities of DRAM cache available in NVDIMM-P. Alternatively, leveraging memory mode with independent NVM and DRAM modules solves the flexibility limitation of NVDIMM-P, but expensive and environmentally unfriendly (and bulky) battery backing is needed [43]. In other words, leveraging DRAM as a fast persistent cache of NVM is limited by energy-backing (or residual energy) capabilities, however such capabilities need to be further boosted when large DRAM modules are needed. Therefore, our goal in this paper is to allow very fast persistent caching of NVMs but without the need for any additional energy-backed capabilities to flush the DRAM cache content to NVM. ***Thus, we enable the integration of preferred DRAM modules in systems with NVMs while also allow caching of persistent data in DRAM, without sacrificing persistence or requiring additional battery-backing capabilities.***

To enable fast persistent DRAM caching of NVM, we propose a novel memory controller design that leverages *selective NVM mirroring* for persistent pages cached in DRAM. Our design supports both memory mode and application direct mode, and transparently ensures durability of updates to persistent pages cached in DRAM. Moreover, our memory controller minimizes the number of writes to NVMs by relaxing the mirroring of DRAM cached pages' updates if their source pages in NVMs are in the logically non-persistent part of NVM (i.e., used for hosting pages that do not need to be persisted). Similar to memory mode support in current processors, our memory controller transparently migrates pages between NVM and DRAM. However, we ensure persistence of DRAM cached pages by inferring their semantic from their original address in NVM. Our scheme only incurs additional writes to DRAM if the page is cached there, in addition to the NVM write which would have occurred anyway. However, future reads will be served from DRAM, which enables fast and persistent caching of durable NVM pages. Additionally, by allowing persistent pages to be located in DRAM, our scheme leverages additional bank-level parallelism for accessing persistent pages, instead of forcing all accesses to NVM. Our scheme is similar in spirit to the write-through scheme typically used in internal processor caches, but involves novel optimizations and design considerations due to the nature of writes and how DRAM is exposed to the system (memory mode or application direct mode). While all prior work on persistent applications explored optimizations for writing to persistent objects, this is the *first work to explore optimizing the* ***read operations*** *of persistent objects.*

To evaluate our design, we use persistent applications from the Whisper benchmark suite [41]. To study the robustness of our approach, we also developed 6 memory intensive benchmarks, similar to previous work by Janus [37]. An open-source architectural simulator, Structural Simulation Toolkit (SST) [50] is used to simulate our approach. On average, we observe 42.02% performance improvement and 88.28% reduction in NVM reads when compared to only using NVM for persistent applications. Note that using NVM-only for persistent data is the only option that allows data persistence without any backup battery and is thus used as our baseline.

In summary, the contributions of our work are as follows:

- We propose Stealth-Persist, a novel hardware support to improve the performance of persistent applications by enabling DRAM caching of the hot persistent pages in DRAM while ensuring the data persistency, without the need of external power backing, and software-transparently.

- We discuss how to integrate Stealth-Persist with vertical and horizontal implementations of hybrid DRAM-NVM main memory systems.
- We discuss several design options for Stealth-Persist, that provide trade-offs between the performance and the number of mirrored pages.
- We extensively analyze the overheads of Stealth-Persist for different region sizes, different replacement policies, and different mirroring threshold.

II. Background

In this Section, we discuss the most related topics to our work to help the reader understand our work, followed by the motivation of this work.

A. Emerging Non-Volatile Memories

Emerging NVMs such as 3D XPoint and Intel's Optane DC feature higher density, byte addressability, lower cost per bit, lower idle power consumption than DRAM, and non-volatility, but have higher access latency and limited write endurance [15], [32], [33], [35]. Due to the non-volatility feature, they can be used as a storage to host filesystem, or as a memory either persistent or non-persistent. For instance, NVM-based DIMMs can be used to hold files and memory pages, which can be accessed using regular load/store operations. To realize this type of accesses, recent operating systems (OSes) started to support configuring the memory as persistent or conventional non-persistent through the DAX filesystems [52]. In DAX filesystems, a file can be directly memory-mapped and accessed using regular load/store operations without copying its content to the page cache [15]. However, NVM's access latency is 3-4x slower than the DRAM's access latency. Therefore, researchers proposed to build memory systems that have both NVM and DRAM portions [48], [62].

B. Hybrid Main Memory (HMM)

Hybrid main memory (HMM) systems are expected to have a large NVM portion due to its density and ultra-low idle power, and a small DRAM portion due to its fast read/write operations. HMM can be deployed in two different schemes, *horizontally* or *vertically*. In the vertical scheme, the NVM is connected as a new memory tier and the DRAM is used to cache the NVM's data [3], [6]. This scheme allows faster access to the large memory pool (NVM), and requires a special hardware to migrate data from the NVM to the DRAM, e.g., the caching of cachelines is handled by Intel's Xeon scalable processor's memory controller in Intel's Optane DC memory mode. However, such a scheme does not provide persistency due to the DRAM's volatility. In the second approach, a horizontal implementation of the HMM system exposes both the NVM and the DRAM to the physical address space, as in NVDIMM-P and Optane DC's application direct mode, and relies on the OS to handle data accesses and page migrations if required [27], [47], [48], [58]. In both cases, a hybrid memory management scheme is required to manage different persistency and performance requirements.

Different hybrid memory management schemes have been proposed in the literature based on the memory hierarchy. Schemes such as HetroOs [27], RTHMS [44], and Nimble [58] proposed software solutions to detect which pages to migrate to the fastest memory (e.g., DRAM). These schemes work with a horizontal implementation of hybrid memory systems when both DRAM and NVM are memory mapped and exposed to the OS. On the other hand, vertical implementation of hybrid memory systems uses the DRAM as a cache. Therefore, the DRAM is not exposed to the OS, wherein caching pages is handled using dedicated hardware, typically an extension of the memory controller as in Intel's Optane DC memory mode [3]. Schemes like the one proposed by Ramos et al. [48] rank the pages based on how frequently each page is accessed using a Multi-Queue (MQ) structure, then use the pages' ranks to decide which pages to migrate to the DRAM and which pages to keep in the NVM. However, tracking all the pages and checking the MQ structure to promote and demote pages entails high overheads, therefore only the head of the queue is checked in each epoch.

After discussing the hybrid memory system's management schemes, we discuss some of the used schemes for page caching in HMM.

C. Page Caching Policy

The page caching policy is used to determine which pages should be cached in the DRAM, if used to cache the NVM pages. In this Section, we discuss two policies that we use later in our design.

First touch policy: This policy caches the pages on the first access and selects a page for eviction based on the LRU algorithm.

Multi-Queue (MQ): The MQ was originally designed to rank disk blocks, and later used by Ramos et al. [48] for page placement in hybrid memory systems. The MQ works as follows: MQ defines `M` LRU queues of block descriptors. The queues are numbered from 0 to `M-1`, with blocks at queue `M-1` are the most accessed blocks. Each descriptor contains the block's number, a reference counter, and a logical expiration time. On the first access to a block, its descriptor is placed in the tail of queue 0, and its expiration time is updated to `CurrentTime + LifeTime`. Both times are measured in the number of accesses, and the `LifeTime` represents the number of consecutive accesses to different blocks before the block is expired. Every time the block is accessed, its expiration time is reset to `CurrentTime + LifeTime`, its reference counter is incremented, and its descriptor is pushed to the tail of its current queue. After a certain number of accesses to the block's descriptor in queue `i`, it gets promoted to queue `i+1` saturating in queue `M-1`. On the other hand, blocks that have not been accessed recently get demoted. On each access, descriptors at the heads of all queues are checked for expiration. If the descriptor is expired, it is placed in the tail of the below queue and has its life time reset, and its demotion flag is set [48]. If a descriptor receives two consecutive demotions, the descriptor is removed

from the MQ structure. In order to reduce the overhead of promotion/demotion, these operations are only performed at the end of each epoch.

As it has been proven that MQ is superior to other algorithms in selecting pages to replace [48], [64], it aligns with our goal, as it facilitates detecting the performance critical pages. Thus, in our experiments, we use the MQ design proposed by Ramos et al. [48]. After discussing the caching policies, we now mention the currently available industrial implementations of hybrid memory systems.

D. Current Industrial HMM Systems

Currently, there are different types of HMM systems available in the market. For instance, JEDEC defines three different standards for HMM known as NVDIMM. NVDIMM types have different characteristics, persistency, and performance features. Moreover, Intel recently revealed details about the memory mode and application direct mode for the Optane DC.

NVDIMM-N contains a DRAM portion, a NVM portion, and a super capacitor. The system uses the DRAM in normal execution, and the NVM is only used to copy the DRAM data using the super capacitor power during crashes. [6]

NVDIMM-F module is a NVM attached to the DDR bus, the access latencies of which is relatively higher than the DRAM. Thus, a DRAM can be installed in the system and used to cache the NVDIMM-F data at the cost of data persistency [6].

NVDIMM-P is still a proposal for a DIMM that have memory mapped DRAM and NVM, wherein the software places the data either on NVM or in DRAM, based on the size and the persistency requirements [6], [7].

Optane DC Memory Mode is an operating mode of Intel's persistent memory, which is similar to the vertical implementation of the HMM. The NVM is used as the system's main memory, and the DRAM is used to cache the NVM's content. This mode provides access to a large memory with access latencies close to DRAM, but *does not provide persistency* [3].

Optane DC Application Direct Mode is an operating mode of Intel's persistent memory, which is similar to the NVDIMM-P.

E. Persistent Memory Programming Model

Due to the persistency feature of NVMs, accessing an NVM memory object is like accessing a storage file. Thus, applications need a way to re-connect to previously allocated memory objects. Therefore, persistent memory regions need names and access control to be accessed. Storage Networking Industry Association (SNIA) recommended OSes to provide standard file semantics for naming, permissions, and memory mappings. Thus, Direct-Access (DAX) support for filesystems was added by several OSes [4]. DAX allows the application to directly use the persistent memory without using the system's page cache. Figure 1 shows how persistent memory aware filesystem works [52].

Using persistent memory (PM) objects requires the programmer to consider multiple issues to ensure the data persistency and consistency. One of these issues is `atomicity`;

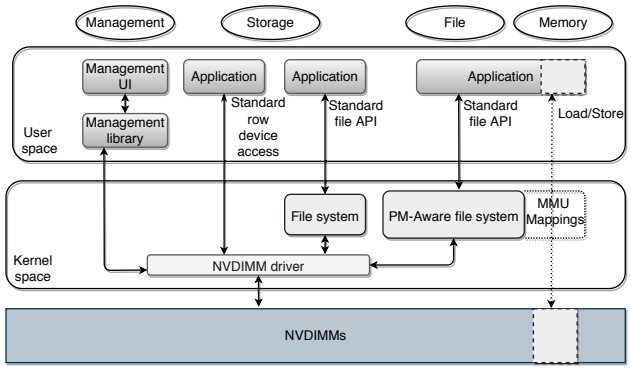

Fig. 1: Persistent memory aware file system.

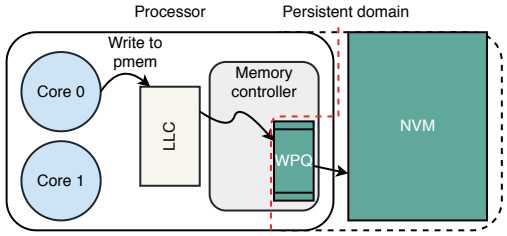

Fig. 2: Persistent domain.

what kind of support is provided by the hardware, and what is left for the software to handle [52]. Intel's hardware ensures the atomicity for 8-byte writes, thus if an object is larger than 8 bytes, it is the software's responsibility to ensure the atomicity of updating the object [52]. Moreover, ensuring data persistency requires pushing the data all the way to the persistent domain, as most of the data updates are done in the volatile processor caches. The persistent domain starts with the `Write Pending Queue` (WPQ), which is a small buffer in the memory controller. The WPQ is supported by the Asynchronous DRAM (ADR) refresh feature. The power provided by the ADR ensures flushing the WPQ content to the NVM in case of power failure [15], [55], [59], [65]. Figure 2 shows the persistent domain in a system with persistent memory.

Listing 1: NVM programming example

```
// a, a_end in PM
a[0] = foo();     // store foo() in a[0]
msync(&(a[0]), ...); // sync to PM
a_end = 0;        // store 0 in a_end
msync(&a_end, ...); // sync to PM
. . .
n = a_end + 1;    // store a_end+1 in n
a[n] = foo();     // store foo() in a[n]
msync(&(a[n]), ...); // sync a[n]
a_end = n;        // store n in a_end
msync(&a_end, ...); // sync to PM
```

In order to flush the data all the way to the persistent region, ensure atomicity, and ordering, a set of specific instructions need to be followed. Listing 1 shows a code example taken

from SNIA NVM Programming Model V1.2 [8]. The code shows the persistent objects `a` and `a_end`. To ensure the persistency, atomicity, and ordering of updates to these persistent objects, `msync` operation is called each time one of these persistent objects is updated. Note that the update at line 7 was not followed by the `msync` operation as it is not updating a persistent object. The `msync` operation is used to force the updates of a memory range into the persistent domain. Moreover, it creates a barrier to guarantee that previous stores are performed before proceeding, `fsync` operation does the same functionality for files [52].

F. Motivation

Having a persistent portion of the main memory enables applications with different persistency requirements. However, to ensure the data persistency, application's persistent data should be placed in the NVM portion of the memory, which hinders the performance of these applications, due to the slow access latencies of NVM. On the other hand, placing the application's data on the DRAM, will lead to better performance but fails to meet the data persistency requirement of such applications. To ensure the application's data persistency, persistent applications should follow the programming model mentioned in Section II-E. As discussed earlier, available persistent memory technologies either provide small memory capacity but fast and battery-backed DRAM-based persistent region, or high-capacity NVM (no need for battery backup) but slow persistent region. The former requires system's support, bulky items, and can limit the size of persistent DRAM depending on the size of the ultra-capacitor or battery. Moreover, it requires certain DIMM changes to support backup mode. Meanwhile, the latter incurs significant performance degradation due to the slow read accesses of persistent objects. While the size of persistent application's data is unlikely to fit in the volatile caches, caching such persistent data in the much larger DRAM can provide significant read speed-ups for persistent objects. Meanwhile, expecting battery-backup, limited DRAM size, and limiting the options (e.g., vendor) of DRAM modules to be integrated in the system, are major drawbacks for the available solutions. Thus, it is important to support caching of persistent data objects in DRAM by just relying on minor changes to the processor chip.

TABLE I: Technologies comparison.

Technology	High persistent capacity	Persistent region performance	Flexibility
NVDIMM-N	✗	✓	✗
NVDIMM-P	✓	✗	✗
Optane DC memory mode	✗(none)	✗(none)	✓
Optane DC app direct mode	✓	✗	✓
Stealth-Persist	✓	✓	✓

Table I compares between the available technologies. From Table I, we can observe the gap between supporting high-performance persistent memory, and high-capacity persistent memory, and hence Stealth-Persist aims to bridge this gap. Figure 3 shows the performance overheads for persistent applications running on Optane DC app direct mode (all persistent data is in NVM), compared to running on a system with DRAM that does not provide data persistency. From Figure 3, we can observe that applications running on Optane DC's app direct mode incur an average of 2.04x slowdown.

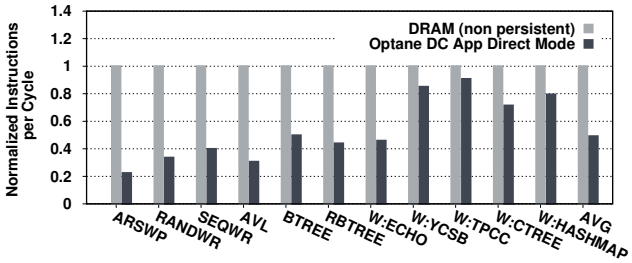

Fig. 3: Normalized performance of persistent applications with DRAM and Optane DC app direct mode with respect to DRAM.

III. Design

In this section, we discuss Stealth-Persist's design in light of possible design options and their trade-offs. First, we start by discussing the design requirements, and the potential design options.

A. Design Requirements

Our design should meet the requirements necessary to allow wide adoption and high-performance, while preserving the semantics of persistent objects. In summary, the requirements are as following:

- **Flexibility:** our design should allow the integration of any DRAM module, regardless of its capacity, in a NVM-equipped system, without requiring any special battery back-ups or specific DIMM modifications.
- **Persistency:** any memory page or object that is supposed to be persistent (i.e., recoverable from crashes) should be recoverable without any extra battery backup support, regardless of where the page is located (NVM or DRAM).
- **High-Performance:** accesses to persistent pages and objects should be as fast as accesses to DRAM.
- **Transparency:** applications that leverage persistent memory for crash recovery should not need to explicitly manage caching and persisting of objects currently residing in DRAM.

To put these requirements in the context of persistent applications, we can imagine a persistent application that accesses tens of gigabytes of persistent objects. Ideally, the system should be able to have DRAM modules integrated in addition to the NVM modules. Systems' owners should have the flexibility on what capacity and vendors to choose such DRAM and NVM modules from, which provides ***flexibility***. However, updates to persistent objects should be durable

and persistent across crashes, regardless of where they exist (DRAM or NVM). While updates to an object in the volatile caches are made durable through the persistency model and framework, i.e., clflushes and memory fences, there is no current support to guarantee the durability of persistent objects if they are cached in the off-chip DRAM, which brings us to the ***persistency*** requirement. Finally, the application should ideally have its persistent objects cacheable in DRAM to minimize the cost of fetching persistent objects that do not fit in the volatile processor caches, which are typically a few megabytes. The requirement to fetch off-chip persistent objects with a latency shorter than the slow NVM's latency (300ns read latency vs 70ns for DRAM) brings us to the third element of our design requirements, ***high-performance***. Thus, persistent applications should be able to cache their persistent objects, that do not fit in the internal volatile caches, in the fast off-chip DRAM, while preserving their persistence capability. Finally, all operations for caching and persisting pages of persistent objects should happen transparently to the software, without exposing such details to the application, which brings us to the final requirement, ***transparency***.

B. Design Options

We will now discuss the design options that can potentially meet our requirements.

One option is to support new instructions that do not commit until a cacheline is flushed – not only from volatile caches, but also from the off-chip DRAM, to the NVM. Such a design option can be realized by introducing new instructions to the instruction set architecture (ISA) with support from the memory controller, or by modifying the implementation of current instructions so that they flush cachelines from the internal volatile caches (e.g., `clflush`), as well as from the DRAM to the NVM. Assuming that the DRAM is operated as a hardware-managed cache for the NVM's data through the memory controller, such instructions would need to have the memory controller first check if the cacheline to be persisted is currently in the DRAM, read it, then flush it to the NVM. The main issues of this approach are: (1) it requires changes to the ISA, persistency programming libraries, and the processor core to support such new instructions. Additionally, (2) the latency to persist data will be significantly increased, especially if the flushed block is marked dirty in the DRAM. Note that even if the DRAM is caching pages instead of cachelines, it will still require similar support but with new instructions that operate at the page granularity, instead of clflush.

One another option is to leverage small fixed size backup capability (e.g., ultra-capacitor) to power flushing a specific portion of the DRAM. For instance, sufficient power to flush 8GB of DRAM, regardless of the total size of the module. The memory controller or the system's software can potentially migrate or place persistent pages in this subregion of the address space, marked as being persistent. When a power failure occurs, the memory controller (or external system circuitry) has sufficient power to flush that portion of the DRAM. While such a solution is similar in spirit to NVDIMM-N, it provides flexibility for choosing any DRAM module and capacity. However, the size of the portion has persistence support is limited to the backup capability of the system. On the other hand, such a solution requires external system support and limits the size of the persistent portion of the DRAM to the power backup capability. Again, such backup capabilities are typically costly, requires high area (bulky), and can be environmentally unfriendly.

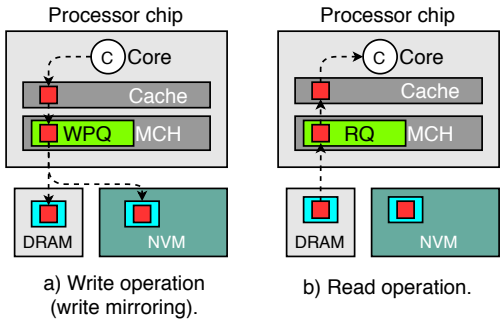

Fig. 4: Read/Write operations in Stealth-Persist.

While the first option provides *high-performance*, *persistency* and *flexibility*, it lacks the *transparency*. Meanwhile, the second option has partial *flexibility* (requires system support and possibly ISA changes), partially *high-performance* (only a small portion of DRAM can be used as persistent memory), *transparency* and *persistency*. Thus, our design should provide full transparency, high-performance, persistency, and flexibility, without any additional system support or backup capabilities beyond what is provided in modern systems.

C. Stealth-Persist Design

While meeting the aforementioned design requirements, our design should also be compatible with the different ways to integrate hybrid memory systems. In particular, vertical memory mode (e.g., memory mode of Optane DC) and horizontal memory mode (e.g., app direct mode of Optane DC).
Before delving into the details of Stealth-Persist support in different integration modes, we will discuss how Stealth-Persist meets the design requirements.

To meet the flexibility requirement, Stealth-Persist is implemented to support mirroring of updates to the persistent region to NVM when cached in DRAM. Thus, it does not require any support from the system and works with any DRAM size. By mirroring updates to persistent pages cached in DRAM, the persistency requirement is met. To make our solution transparent to software, Stealth-Persist's mirroring operations occur at the memory controller and do not require any changes to the application or persistent programming library. Finally, to support high-performance access to persistent pages, our scheme serves read requests to persistent objects from the DRAM, if cached there. Figure 4 depicts the read and write operations in Stealth-Persist, at a high-level.
As shown in Figure 4, the Memory Controller Hub (MCH) handles mirroring of writes to persistent pages if cached in the DRAM, while serving read requests directly from the DRAM.

By doing so, Stealth-Persist ensures the durability of writes to the NVM while allowing fast read operations to such persistent objects.

While at a high-level, the design looks similar to the write-through scheme typically used in internal processor caches, many challenges and potential divergences arise when considering the context of hybrid memory systems. The first challenge is how to decide if a page should be mirrored or not. The second challenge is how to quickly identify if a page is cached in the DRAM or not, where it is cached in the DRAM, and how to guarantee that both copies are coherent during run-time. Third, since not all pages in the NVM need to be persisted, updates to pages stored in the NVM need to be selectively mirrored. Finally, Stealth-Persist needs to be adapted to work with the myriad of ways to integrate hybrid memory systems. The following parts of this section discuss these challenges and how we overcome them.

1) Page Mirroring: Regardless of the HMM management scheme used, horizontal (e.g., app direct mode) or vertical (memory mode), Stealth-Persist requires a part of (or the whole) DRAM to be used as a mirror region for persistent pages. In the vertical memory setup, the whole DRAM will be used as a cache for NVM, and thus, any page cached in the DRAM can be possibly mirrored to the NVM as well. Meanwhile, for the horizontal setup, since the DRAM and the NVM physical ranges are explicitly exposed to the system, we have the memory controller reserve a portion of the DRAM to be used merely as a mirror region. The remaining part of the DRAM will be exposed to the system directly as in app direct mode. Any persistent page located in the NVM can be cached in the mirror region in the DRAM regardless of the setup, i.e., the size of such region. On each memory access that targets a NVM address, we need to transparently check if the page is currently resident in the DRAM. This check is needed for both read and write operations; read operations can be served directly from the DRAM, if the accessed page is cached there, whereas write operations need to update the copy in the NVM to honor coherence between the mirrored page copies and ensure persistency. When a page is not present in the DRAM, we need to read it (or write to it) from the NVM. Since the mirror region can be thought of as a buffer/cache for persistent pages in the NVM, we need to define the insertion and evictions policies for said cache/buffer in DRAM.

For simplicity, we use a page insertion policy similar to what is used in vertical memory management schemes. By doing so, if memory mode is used, no changes are required to the management policy, except additional writes to the NVM if persistent pages are cached in the DRAM. Meanwhile, for app direct mode, the defined mirror region in the DRAM will be managed similar to the DRAM cache in memory mode, in addition to the mirroring writes to the NVM. With this in mind, we use two simple policies for page placement in the DRAM buffer: (1) first-touch policy (FTP) and (2) multi-queue (MQ) policy as proposed in prior work [48].

2) DRAM Mirror Region Lookup: To ensure Stealth-Persist can quickly check if a page is in the DRAM (mirror region) or not, Stealth-Persist keeps track of the mirror region pages using a hardware managed table. The mirror's mapping table contains the translations of the mirror's cached pages addresses, as shown in Figure 5. Each entry in the mapping table contains a group ID, which is calculated using a modulus function of the mirrored page address in the NVM over the number of pages in the mirror region. Additionally, each entry contains six pairs of translations that maps the 36-bit NVM's page address to the 36-bit mirror DRAM's page address. Additionally, we use 3 bits for each translation (18 bits total) as LRU bits for replacement policy in each entry, which makes a total of 450 bits for translations and the rest of the 512 bits are used for the group ID (32 bits) and padding. Thus, a page can be removed from the mirror region by either the clock replacement policy or by the LRU eviction within the entry.

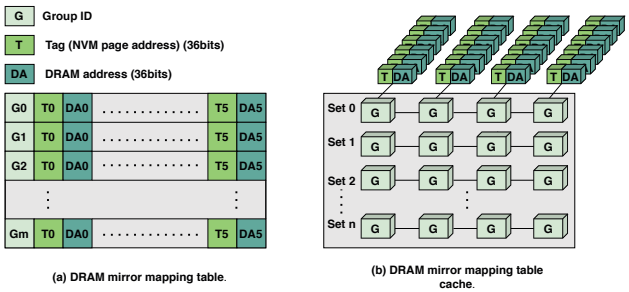

Fig. 5: Mirroring region mapping table.

Note that the storage requirement of the mirror's mapping table is 64 bytes for every 6 pages in the mirroring region. Therefore, we use a small cache in the memory controller to cache the mirror's mapping table entries while maintaining the table in the DRAM. Whenever a memory request to the persistent region is received, the group ID of the requested page is calculated and the mirror's mapping table cache is checked for the requested group ID, which can result in three different scenarios. ① The entry is cached and the page is cached → the request is served from associated DRAM page. ② The entry is cached and the page is not cached → the page is not mirrored and the request is served from the NVM. ③ The entry is not cached → mapping table in the DRAM must be checked to obtain the entry and its mirrored pages. Since a mapping table cache miss can lead to serving the request from the DRAM with two accesses, or from the NVM after checking the DRAM, we send the request to the DRAM and the NVM then serve the request from the DRAM, if the entry is in the table, or from the NVM if it was not.

3) Coherent Updates to Mirrored Pages: In Stealth-Persist, coherence between the mirror region pages and the NVM pages should be maintained. Since persistent pages are expected to be recoverable, writes to persistent pages should be durable. Therefore, writes to the mirror region should be pushed to both memories. Stealth-Persist pushes the write requests to the mirror region pages into the DRAM's volatile write buffer and to the NVM's persistent WPQ. Note that a write request is only retired once it is placed in the WPQ, which ensures the write persistency. On the other hand, mir-

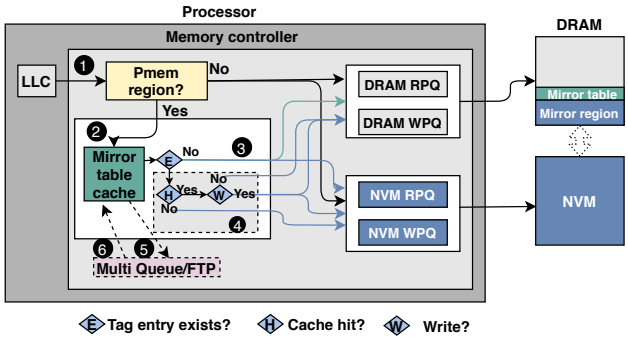

Fig. 6: Stealth-Persist overall design.

rored pages that belong to non-persistent region do not require data coherence nor recoverability, which is why Stealth-Persist implements selective mirroring.

Stealth-Persist does not have any impact on coherence. If the DRAM and the NVM modules are on the same socket, which is the configuration supported for Intel's DC PMM, coherence between the NVM and DRAM copy is managed by the MC through mirroring, whereas coherence with internal processor caches is handled in conventional systems. However, if we deviate from the current standard of having the NVM and the DRAM on the same socket, i.e., each is on a different socket, then we can designate the memory controller near the NVM as the master, and thus it will be responsible to handle mirroring, remapping, etc., and accordingly forward any requests that hit in the mirror table cache to the memory controller in the socket has the DRAM module.

4) Selective Mirroring: Stealth-Persist implements selective mirroring techniques to reduce the number of writes to the NVM, which can be done by committing the writes directed to the non-persistent region to its DRAM mirrored version only. Stealth-Persist implements selective mirroring in the vertical HMM implementation just as in the Optane DC's memory mode, and in the horizontal HMM implementation as in the Optane DC's app direct mode. In both cases, Stealth-Persist requires the address range of the persistent memory region, which can be passed to Stealth-Persist by the kernel during system bring-up – for example, the Linux command memmap=2G!8G could be used to reserve a 2GB persistent region starting at address 8G. Note that forwarding the writes of the pages in the non-persistent region to its mirrored version only, violates the coherency of these pages. However, since the pages are in the non-persistent region, and these applications are not expected to be recoverable, the writes can be committed to the mirrored page only, while the whole page will be written back to the NVM if the page gets evicted.

D. Overall

The overall Stealth-Persist design is shown in Figure 6. For every last level cache (LLC) miss, first the memory controller checks if the request is to the persistent region or not ①. If the request is to the persistent region, the mirror table cache is queried for the current status of the NVM page ②. As discussed in Section III-C2, the mirror table cache verifies the mirroring status of the NVM's page by either looking into the already cached mirror table entries, or by fetching the entries from the mirror table stored in the DRAM, and replacing a group ID and the respective mapping table entry using a LRU policy ③. If the page is mirrored, read requests are forwarded to the DRAM while write requests are forwarded to both the DRAM, to update the mirroring region, and to the NVM, to persist the data ④. In the case of a read, the persistent memory access is forwarded to the multi-queue or FTP unit ⑤. This unit decides if a page should be mirrored and if so, the mirror table cache is triggered to replace one of the mappings using the LRU policy ⑥.

E. Stealth-Persist versus NVM libraries

Several studies proposed the use of NVM libraries to address atomicity, crash consistency, and performance issues when NVMs are used as a main memory. NVM libraries focus on moving writes out of the critical path to improve the performance, but do not reduce read latency. In contrast, Stealth-Persist improves the performance by reducing the latency of the basic memory read operations, which is still required with NVM libraries. Some schemes focus on fault tolerance (e.g., Pangolin [61]), performance and strong consistency (e.g., NOVA [57]), programming effort reduction and performance (e.g., Pronto [39]). While such schemes improve the system's performance by moving the writes overhead out of the application's critical path, or by buffering some of the updates in the DRAM, the writes to the NVM are inevitable if persistency is required. In contrast, Stealth-Persist propagates the writes to the NVM if they are directed towards a persistent region within the NVM, and buffers the writes to the non-persistent region in their DRAM cached pages. Additionally, Stealth-Persist operates in a different layer than the proposed NVM libraries, which makes Stealth-Persist orthogonal to such schemes. As a matter of fact, Stealth-Persist can be used concurrently with the mentioned schemes to improve the performance even further.

In a different direction, Hagmann [24] proposed a scheme that maintains a log to recover the filesystem in disks. Petal [34] enables the clients to access distributed disks by creating virtual disks, which improves the system's performance and increases the throughput. To provide the recoverability, Petal uses write-ahead-logging. Condit et al. [23] proposed a scheme that enables crash consistency for persistent memories using shadow paging, in which the writes are atomically committed in-place or using localized copy-on-write. BTRFS [49] for the Linux filesystem uses B-tree data structure, and uses copy-on-write as the update scheme. Rosenblum and Ousterhot [51] proposed a log structured filesystem that performs all the writes to the disk in a sequential manner, and maintains indexing information for faster data retrieval. Seltzer et al. [53] proposed a log-structured filesystem that has improved write performance, less recovery time, and enables embedded transactions and versioning. Such schemes were proposed to ensure atomicity, recoverability, and improve the performance

TABLE II: Configuration of the simulated system.

Processing Element	
Processor	4 Cores, X86-64, Out-of-Order, 2.00GHz, 2 issues/cycles, 32 max. outstanding requests.
L1 Cache	Private, 4 Cycles, 32KB,8-Way
L2 Cache	Private, 6 Cycles, 256KB, 8-Way
L3 Cache	Shared, 12 Cycles, 1MB/core, 16-Way
Cacheline Size	64Byte
Hybrid Main Memory	
DRAM	Size: 1GB, RCD=RP=14, CL=14 CL_WR=12
NVM	Size: 4GB, Read latency 150ns, Write latency 500ns
DRAM Mirror	
Size	32MB
MQ mirroring threshold level	4
Epoch interval	10000 reads
Mirroring Table cache	size: 128 entries (groups), associativity: 4, latency: 1 cycle

TABLE III: Benchmarks description.

Benchmark	Description	MPKI
ARSWP	Swap random elements of an array	31.11
RANDRW	Random updates to persistent memory	32.43
SEQRW	Sequential updates to persistent memory	6.18
AVL	Insert and look up random elements in avl tree	30.38
BTREE	Insert and look up random elements in b-tree	21.01
RBTREE	Insert and look up random elements in red-black tree	56.11
W:YCSB	N-Store variant to evaluates database management systems	3.88
W:TPCC	N-Store variant to measures the performance of online transactions	3.97
W:CTREE	NVML variant of crit-bit tree	1.75
W:HASHMAP	Hashmap implemented with NVML [56] library	0.84
W:ECHO	Scalable key-value store for persistent memory	9.54

in disks. However, the proposed schemes do not improve the performance of read operations while ensuring persistency. Thus, they are orthogonal to Stealth-Persist.

IV. Methodology

We modeled Stealth-Persist in the Structural Simulation Toolkit (SST) simulator [50]. SST is a cycle-level event-based simulator with modular designs for different hardware components. SST is widely used in the industry and academia [28], [30], [31]. We implemented a hybrid memory controller component to handle both DRAM and NVM. Stealth-Persist required components, Mirroring-Table and the MQ, are modeled in a hybrid memory controller module to perform all the relevant tasks. The configuration of the simulated system is shown in Table II. The simulated system contains 4 out-of-order cores with each core executing 2 instructions per cycle. The frequency of the cores is 2GHz. Three levels of caches, L1, L2, and L3 (inclusive) are simulated with sizes 32KB, 256KB, and 1MB respectively. The DRAM capacity is 1GB and the NVM capacity is 4GB[1]. NVM read and write latencies are 150ns and 500ns [14]

A. Workloads:

To evaluate our proposed scheme, we ran 11 persistent applications. As shown in Table III, six of the benchmarks were developed in-house, all of which are designed to stress memory usage and were used in previous work [37]. The functionality of each of these applications is described as follows.

❶ ARSWP: This benchmark randomly chooses two keys from the database and swaps them.

❷ RANDWR: Random keys are chosen and the database entry with the chosen key is updated with a random value.

❸ SEQWR: This is similar to RANDWR but the keys are chosen sequentially starting from the 1st element of the database.

❹ AVL: The database is mapped to an AVL tree and a randomly generated key is searched in the mapped database. If the key is not found an insertion operation is triggered.

❺ BTREE: This benchmark maps the database to a B-tree and similar to AVL, a random key is searched, if not found the key is inserted with dummy data.

❻ RBTREE: Similar to AVL and BTREE benchmarks, RBTREE benchmark maps the database to a RB-tree and a random key is searched.

We also ran five benchmarks from the WHISPER benchmark suite [41] (preceded by W: in Table III) developed by the University of Wisconsin-Madison in collaboration with HP Labs. The TPCC benchmark measures the performance of online transaction processing systems (OLTP) based on a complex database and various database transactions that are executed on it. The Yahoo Cloud Serving Benchmark (YCSB) is a programming suite to evaluate database management systems. W:TPCC and W:YCSB benchmarks are variants of the Whisper benchmark suite that are modeled after N-Store [12], which is a remote data base management system for persistent memory. W:CTREE and W:HASHMAP benchmarks were developed using the NVML [56] library which performs insert, delete and get operations to the persistent memory regions. W:ECHO is a scalable key-value store for persistent memory regions. Map_get functionality is evaluated for W:CTREE and W:HASHMAP benchmarks.

The key size of all these benchmarks is 512 bits and the database size is 1GB. Before evaluating these benchmarks, first the database is filled with random keys. Misses per kilo instructions (MPKI) for these benchmarks are shown in Table III. Each benchmark is evaluated for 500M instructions.

B. DRAM Mirror Configuration:

To mirror the NVM's data, a 32MB of the DRAM is used. However, we vary the size of the mirroring region from 2MB to 1GB (entire DRAM is used as mirroring region) as discussed in Section V-E. Mirroring is done at page granularity. In MQ mechanism a page is mirrored only when it reaches MQ level 4, i.e., when a page is read 16 times. The epoch interval

[1]Note that the selected sizes of DRAM and NVM are chosen due to the limitation of simulation speed, however, the most important parameters are the mirroring region size (32MB) and the average footprint of the applications (256MB). Since all the data of persistent applications will reside in NVM and can be cached persistently in the mirroring region, we focus on the ratio of application's footprint to the mirroring region (8:1 ratio), which we vary later in the paper.

is set to 10000 read operations. Although we evaluated Stealth-Persist approach with the above-mentioned configurations[2], we performed sensitivity analysis by varying the DRAM mirror size, and threshold level. Mirror table cache size maintained by the memory controller is 128 groups with each group having 6 mappings. Mirror table cache lookup latency is 1ns.

V. Evaluation

In this Section, we discuss the results of Stealth-Persist against a system using the NVM directly for persistency. We further show sensitivity analysis by varying different parameters that impact the performance.

A. Impact of Stealth-Persist on Performance

Figure 7 shows the performance improvement with Stealth-Persist methods. The baseline scheme is the Optane DC app direct mode scheme wherein all the persistent memory requests are stored to the persistent memory (NVM) only. This is the typical way of achieving data persistency for persistent applications for such systems. On average, the performance improves by 30.9% and 42.02% with Stealth-Persist MQ and FTP approaches. The application's performance improvement is a function of the mirroring region's hit rate as discussed in Section V-B. Improvement with Stealth-Persist FTP is higher than Stealth-Persist MQ method since every page that is read is mirrored in the DRAM, which leads to a huge number of pages copied from NVM to DRAM. On average, we observe Stealth-Persist FTP mirrors 542.96x more pages than Stealth-Persist MQ approach, which significantly increases the memory bus traffic and energy use. For sequential memory access benchmarks like SEQWR and W:ECHO, the improvement with Stealth-Persist FTP is substantial – 2.34x and 2.2x respectively. Since such benchmarks access the memory sequentially, the spatial locality for these benchmarks is high. Hence, when a page is read, it is mirrored immediately in Stealth-Persist FTP and is accessed for the contiguous memory accesses. On the other hand, Stealth-Persist MQ approach, first, the page should reach a threshold to be mirrored. For AVL and RBTREE workloads, Stealth-Persist MQ approach outperform Stealth-Persist FTP because Stealth-Persist FTP replaces the pages in the mirroring region very frequently, which leads to evicting hot pages from the mirroring region. On the other hand, Stealth-Persist MQ approach tends to keep hot pages in the mirroring region.

For the ARSWP workload, the performance of Stealth-Persist scheme barely changes compared to Optane DC app direct mode and, from Figure 3, it suffers significantly compared to a system using DRAM as main memory – it is 4.39x slower. However, the ARSWP application memory accesses are very sparse, and thus the reuse distance of the pages are high, which leads to evicting those pages in Stealth-Persist FTP approach before they are reused. Additionally, the pages of the ARSWP application do not reach the mirroring limit for Stealth-Persist MQ approach. Hence, the performance degrades by 3% in

[2]We used CLWB to persist the data and keep the data in the processor caches.

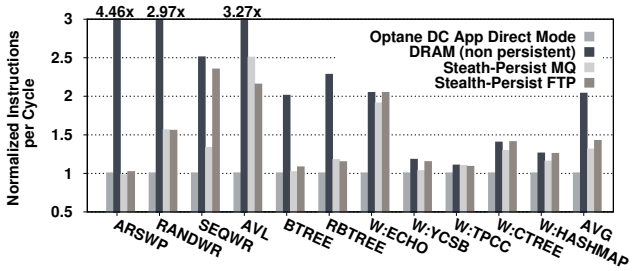

Fig. 7: Normalized performance improvement of Stealth-Persist methods compared to Optane DC app direct mode.

MQ approach due to checking the mirror region while having only 0.02% hit rate. On the other hand, Stealth-Persist FTP performance improves by 1.6% for ARSWP benchmark due to having 3% hit rate. However, the performance of ARSWP improves when the mirroring region size is increased, as shown in Section V-E1.

B. DRAM Mirror Hit Rate

Figure 8 shows the percentage of reads served by the DRAM mirroring region. We note that applications with sequential memory accesses show the best performance improvement – FTP is showing a very high hit rate for these applications. On the other hand, applications with random stride accesses and ones with hot pages, show the highest hit rates in Stealth-Persist MQ approach. As Figure 8 illustrates, the mirrored pages serve an average of 57.81% of the overall memory reads in Stealth-Persist FTP approach. For Stealth-Persist MQ, it serves an average of 24.78% of the overall reads with a reasonable number of page mirrors compared to Stealth-Persist FTP. As shown in Figure 8, memory bounded applications with the highest hit rates show the highest performance improvement. In Stealth-Persist FTP, the mirroring hit rate for WHISPER benchmarks, like CTREE and HASHMAP is high, but the performance improvement is not as much as for SEQWR and ECHO benchmarks. This is because CTREE and HASHMAP applications are not as memory intensive as EPOCH and SEQWR, which is correlated with the MPKI for CTREE and HASHMAP, as shown in Table III – CTREE has an MPKI of 1.75 and HASHMAP has an MPKI of 0.84.

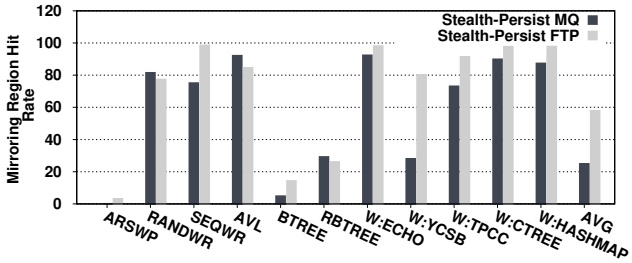

Fig. 8: Percentage of requests served by the mirroring region.

C. Impact of Stealth-Persist on NVM Reads

In this section, we show the reduction in the number of reads sent to the NVM using Stealth-Persist approaches. When

the mirroring region hit rate is high, most of the reads are served by the mirroring region, which reduces the number of reads sent to the NVM. Figure 9 shows that, on average, the number of NVM reads are reduced by 88.28% and 73.28% with Stealth-Persist FTP and MQ approaches, with respect to Optane DC app direct mode (100%). For the SEQWR and W:ECHO benchmarks, which show the highest performance improvement with Stealth-Persist FTP, NVM reads are significantly reduced by 98.42% and 98.02%, respectively.

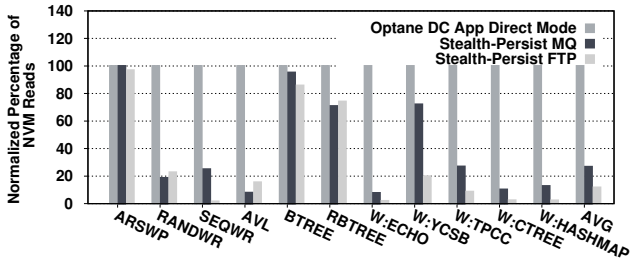

Fig. 9: Percentage of reads served by NVM with Stealth-Persist methods compared to Optane DC app direct mode.

D. Impact of Stealth-Persist on NVM Writes

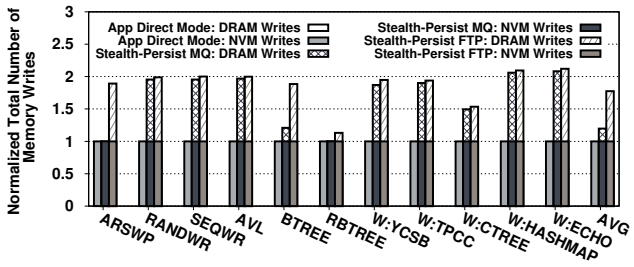

Fig. 10: Number of writes to DRAM and NVM with Stealth-Persist methods compared to Optane DC app direct mode normalized to NVM writes.

As Figure 10 shows, Stealth-Persist schemes do not have any impact on the number of writes to the NVM. However, Stealth-Persist sends the writes of the mirrored pages to the DRAM as well. Therefore, Stealth-Persist does not affect the NVM's write endurance nor increase the energy consumption, which might be caused by increasing the NVM writes.

E. Sensitivity analysis

Although Stealth-Persist FTP and MQ improve the performance by 42.02% and 30.9% on average compared to the baseline (Optane DC app direct mode), there is still a room for improvement since the mirroring region hit rate is 57.81% and 24.78%, on average. Misses can happen for many reasons, but are mainly affected by the mirroring region size and mirroring threshold in Stealth-Persist design. However, increasing the mirroring region size will increase the hardware complexity (Mirroring-Table size) while reducing the mirroring threshold may result in early replacement of required pages, which may degrade the overall performance. To fully analyze the effects of the mirroring region size and the mirroring threshold, we vary the mirroring region size and the mirroring threshold in this section. Also, we show the performance improvement on fast and slow NVMs. The average of all the workloads is shown in the sensitivity results.

1) Impact of Mirroring Region on Performance: The number of persistent pages that can be mirrored in the DRAM is dependent on the percentage of the DRAM memory reserved for mirroring. To avoid significant memory overhead, Stealth-Persist reserves only 32MB of the DRAM, which is 3.125% of the DRAM in the simulated system, for mirroring of persistent memory pages. However, as discussed previously, the more pages that can be mirrored, the greater the upper bound on system performance when using Stealth-Persist. Hence we varied the mirroring region size from 2MB to 1GB to evaluate performance improvements with Stealth-Persist. Note that when the mirroring region size is 1GB, the entire DRAM is reserved to cache mirroring pages.

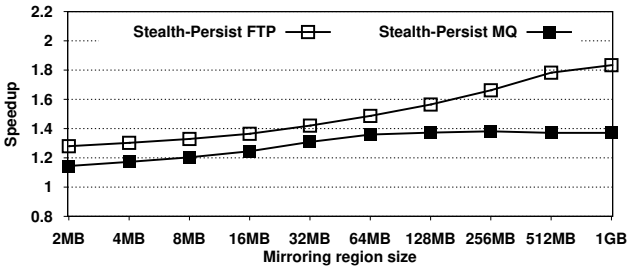

Fig. 11: Performance improvement with different mirroring region sizes.

Figure 11 shows that increasing the mirroring region size improves the performance of both FTP and MQ. As the mirroring region size increases from 2MB to 1GB, the performance improvement increases from 1.28x to 1.83x with Stealth-Persist FTP and increases from 1.14x to 1.38x with Stealth-Persist MQ. The improvement is saturated after 64MB mirroring region size with Stealth-Persist MQ, since MQ is a confirmation based approach wherein a NVM page is mirrored only if it is accessed for more than the threshold number of times(4). Hence, even though the mirroring region size is increased, the number of pages to mirror is bounded by the threshold and hence performance improvement is saturated. When mirroring region size is 64MB, the performance improvement with Stealth-Persist FTP is 1.48x and 1.35x with Stealth-Persist MQ. Also, as asserted, ARSWP benchmark which is not showing performance improvement with 32MB mirroring size, achieves an improvement of 1.06x, 1.22x, 1.75x, 2.65x, and 3.22x when the mirroring region size is 64MB, 128MB, 256MB, 512MB, and 1GB with Stealth-Persist FTP, respectively. However, with Stealth-Persist MQ we observe no improvement since the pages of the ARSWP application do not reach the mirroring threshold.

2) Mirroring Threshold Level Impact on Performance: In Figure 12, we show the results when varying the mirroring

threshold queue level. When the threshold level is decreased, the performance improvement with Stealth-Persist MQ approach is increased. We observe a performance improvement of 1.46x when the threshold level is set to 1 and, with a threshold level of 4, the performance improvement is 1.3x. Stealth-Persist behaves aggressively when the threshold level is reduced since more pages are identified as mirroring candidates. That is, when the threshold level is 1, a page is identified as a mirroring candidate if the application reads the page at least 2 times. But, when the threshold level is 4, a page is mirrored only if it is read a minimum of 16 times. Hence, the performance improvement achieved by reducing the threshold level is at the cost of increasing the number of pages to mirror.

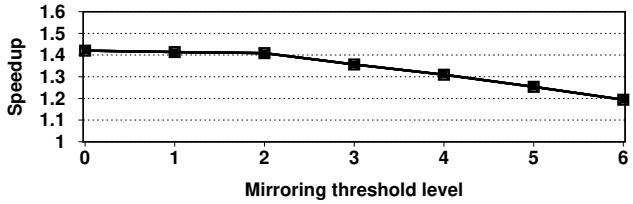

Fig. 12: Performance improvement by Stealth-Persist MQ by varying the mirroring threshold level.

On the other hand, increasing the threshold level can hurt the performance improvement due to two reasons. 1) A page is mirrored after reaching the threshold level, as the queue level increases, and the application has to access the page more frequently to be identified as a mirroring candidate. In general, the percentage of these pages is small and they are often cached in the processor. 2) The hotness of the page is lost after reaching the threshold level. For instance, if the threshold level is set to 6, a page has to be accessed for a minimum of 64 times to be mirrored. However, after accessing the page for 64 times, the application may no longer need access to this page, negating the impact of mirroring.

3) Impact of NVM Read/Write Latency on Performance: Although the NVM's read latency is comparable to the DRAM's read latency, it is still slower than the read latency of the DRAM. The write latency of the NVM suffers significantly compared to the DRAM. NVM's read/write latencies are critical while mirroring pages from the NVM to the DRAM. Hence we study the impact of Stealth-Persist for slow and fast NVM's read/write latencies. We varied the NVM's read and write latencies as shown in Figure 13. Figure 13 categorizes the NVM into 4 types - moderate; read and write latencies are 150ns and 500ns, slow: read and write latencies are 300ns and 700ns, very slow: read and write latencies are 500ns and 900ns and ultra slow: read and write latencies are 750 and 1000ns. As the NVM's read/write latencies increase the performance improvement also increases with Stealth-Persist. With ultra-slow NVM, Stealth-Persist improves the performance by 1.87x and 1.54x with FTP and MQ respectively.

VI. Related work

Hybrid Memories: Recently, a lot of work has explored how to improve the performance of hybrid memory systems.

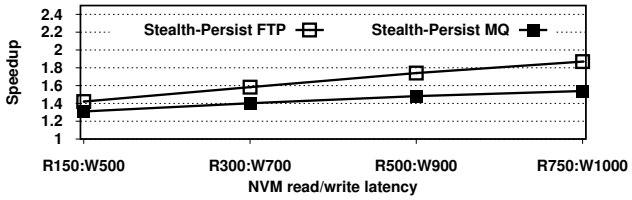

Fig. 13: Performance improvement with Stealth-Persist for different NVM's read/write latencies compared to Optane DC app direct mode respectively.

For instance, Ramos et al. [48] proposed a scheme for page placement in hybrid memory systems. The proposed scheme uses a multi-queue to rank the pages and only migrates the performance-critical pages to the DRAM. However, the scheme does not ensure data persistency and is only focused on placing the performance-critical pages in the DRAM. The authors of HetroOs [27] proposed an application transparent scheme that exploits the application's memory usage information, provided by the operating system, to decide where to place the data in heterogeneous memory systems. However, the motivation in HetroOs is purely for system performance and does not provide persistency guarantees. Therefore, applications with persistency requirements would still have to suffer high NVM latency. The authors of Nimble [58] proposed a scheme that reuses the operating system's page tracking structures to tier pages between memories. Additionally, Nimble provides several optimizations such as transparent huge page migration and multi-threaded page migration, which leads to 40% performance improvement compared to the native Linux system. However, Nimble improves page migration between memories and does not ensure the data persistency. Agarwal et al. [10] proposed a page placement scheme for GPUs in hybrid memory systems. However, the proposed scheme migrates pages between memories based on the application bandwidth requirements, which does not consider the data persistency. Yoon et al. [60] devised a policy that enables DRAM to cache pages with a high frequency of row buffer misses in the NVM memory. CAMEO [20], PoM [54], Mempod [45] and BATMAN [21] discussed the possible relaxations to maximize overall memory bandwidth. The proposed techniques rely on the compiler support or Linux kernel to detect pages of interest. Migrating remote pages to the local memory in disaggregated memory systems is explored by Lim et al. [36] and Kommareddy et al. [30]

NVM Data Persistency: Ensuring the persistency, performance, and crash consistency of NVM resident data has been under the spotlight recently. For instance, Janus [37] improves the persistent applications write latency by decomposing the back-end memory operations into smaller sup-operations, then overlapping the sup-operations. Besides the NVM libraries mentioned earlier, Intel's PMDK [9], REWIND [18], NV-Heaps [22] and LSNVMM [25] provide software based high level interface for the programmers to ensure the data persistency and provide crash consistency support. Hardware based

approaches provide consistency using transactions and low-level primitives [19], [26], [29], [42], [63]. The proposed scheme, Stealth-Persist optimizes persistent workloads read operations in hybrid memories and is orthogonal with the previous approaches.

VII. Conclusion

Improving the performance of persistent applications in hybrid memory systems requires caching the NVM resident data in the DRAM. However, caching the persistent application's data in the DRAM nullifies the persistency of those cached pages. Ensuring the persistency of DRAM cached pages can be achieved by power-backing the DRAM. However, using batteries to power-back the DRAM is expensive, unreliable, incompatible with legacy systems, and is not environmentally friendly. Therefore, we propose Stealth-Persist, a novel memory controller design that allows caching the NVM resident pages in the DRAM while ensuring the pages' persistency. By serving NVM requests from DRAM, Stealth-Persist exploits bank-level parallelism which reduces the memory contention and brings in additional performance gains. Stealth-Persist improves the system performance of persistent applications in hybrid memory systems by 42.02% on average with Stealth-Persist FTP. However, Stealth-Persist FTP requires a significant number of pages to be copied from the NVM to DRAM. With Stealth-Persist MQ approach, we show a performance improvement of 30.09% with reasonable page mirrors. Stealth-Persist achieves this improvement at the cost of small hardware managed table, a small cache in the memory controller, and by utilizing the WPQ.

VIII. Acknowledgments

This work has been funded through Sandia National Laboratories (Contract Number 1844457). Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International,Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. Part of the work was funded by NSF's CNS-1814417 and CNS-1908471, and Office of Naval Research (ONR). This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

References

[1] "Flash Memory Summit 2018 - Persistent memory DIMMs," https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/consumer-ssds/optane-ssd-9-series.html, accessed: 2020-23-07.

[2] "Intel Optane DC Persistent Memory," https://builders.intel.com/docs/networkbuilders/intel-optane-dc-persistent-memory-telecom-use-case-workloads.pdf, accessed: 2020-24-07.

[3] "Intel® Optane™ DC Persistent Memory Operating Modes Explained," https://itpeernetwork.intel.com/intel-optane-dc-persistent-memory-operating-modes/#gs.xtmcnw, accessed: 2020-24-02.

[4] "Linux Direct Access of Files (DAX)." [Online]. Available: https://www.kernel.org/doc/Documentation/filesystems/dax.txt

[5] "NVDIMM," https://www.jedec.org/sites/default/files/Bill_Gervasi.pdf, accessed: 2020-02-02.

[6] "NVDIMM - Changes are Here So What's Next," https://www.snia.org/sites/default/files/SSSI/NVDIMM\%20-\%20Changes\%20are\%20Here\%20So\%20What's\%20Next\%20-\%20final.pdf, accessed: 2020-03-29.

[7] "NVDIMM-P," https://software.intel.com/content/www/us/en/develop/articles/enabling-persistent-memory-in-the-storage-performance-development-kit-spdk.html, accessed: 2020-02-02.

[8] "NVM Programming Model v1.2," https://www.snia.org/sites/default/files/technical_work/final/NVMProgrammingModel_v1.2.pdf, accessed: 2020-02-02.

[9] "Persistent Memory Development Kit," https://pmem.io/pmdk/, accessed: 2019-03-27.

[10] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. W. Keckler, "Page placement strategies for gpus within heterogeneous memory systems," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015, pp. 607–618.

[11] M. Alwadi, K. Zubair, D. Mohaisen, and A. Awad, "Phoenix: Towards ultra-low overhead, recoverable, and persistently secure nvm," *IEEE Transactions on Dependable and Secure Computing*, 2020.

[12] J. Arulraj, A. Pavlo, and S. R. Dulloor, "Let's talk about storage & recovery methods for non-volatile memory database systems," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 707–722.

[13] A. Awad, S. Blagodurov, and Y. Solihin, "Write-aware management of nvm-based memory extensions," in *Proceedings of the 2016 International Conference on Supercomputing*, 2016, pp. 1–12.

[14] A. Awad, S. Hammond, C. Hughes, A. Rodrigues, S. Hemmert, and R. Hoekstra, "Performance analysis for using non-volatile memory dimms: opportunities and challenges," in *Proceedings of the International Symposium on Memory Systems*, 2017, pp. 411–420.

[15] A. Awad, Y. Solihin, L. Njilla, M. Ye, and K. Zubair, "Triad-nvm: Persistency for integrity-protected and encrypted non-volatile memories," in *Proceedings of the 46th International Symposium on Computer Architecture*. ACM, 2019, pp. 169–180.

[16] A. Awad, S. Suboh, M. Ye, K. Abu Zubair, and M. Al-Wadi, "Persistently-secure processors: Challenges and opportunities for securing non-volatile memories," in *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2019, pp. 610–614.

[17] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging locks for non-volatile memory consistency," *ACM SIGPLAN Notices*, vol. 49, no. 10, pp. 433–452, 2014.

[18] A. Chatzistergiou, M. Cintra, and S. D. Viglas, "Rewind: Recovery write-ahead system for in-memory non-volatile data-structures," *Proceedings of the VLDB Endowment*, vol. 8, no. 5, pp. 497–508, 2015.

[19] S. Chen, L. Liu, W. Zhang, and L. Peng, "Architectural support for nvram persistence in gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 5, pp. 1107–1120, 2019.

[20] C. C. Chou, A. Jaleel, and M. K. Qureshi, "Cameo: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014, pp. 1–12.

[21] C. Chou, A. Jaleel, and M. Qureshi, "Batman: Techniques for maximizing system bandwidth of memory systems with stacked-dram," in *Proceedings of the International Symposium on Memory Systems*, 2017, pp. 268–280.

[22] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1, pp. 105–118, 2011.

[23] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 133–146.

[24] R. Hagmann, "Reimplementing the cedar file system using logging and group commit," in *Proceedings of the eleventh ACM Symposium on Operating systems principles*, 1987, pp. 155–162.

[25] Q. Hu, J. Ren, A. Badam, J. Shu, and T. Moscibroda, "Log-structured non-volatile main memory," in *2017 USENIX Annual Technical Conference ATC 17)*, 2017, pp. 703–717.

[26] J. Izraelevitz, T. Kelly, and A. Kolli, "Failure-atomic persistent memory updates via justdo logging," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 2, pp. 427–442, 2016.

[27] S. Kannan, A. Gavrilovska, V. Gupta, and K. Schwan, "Heteroos: Os design for heterogeneous memory management in datacenter," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 521–534.

[28] A. Kokolis, D. Skarlatos, and J. Torrellas, "Pageseer: Using page walks to trigger page swaps in hybrid memory systems," in *2019 IEEE International Symposium on High Performance Computer Architecture*. IEEE, 2019, pp. 596–608.

[29] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch, "Delegated persist ordering," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2016, pp. 1–13.

[30] V. R. Kommareddy, S. D. Hammond, C. Hughes, A. Samih, and A. Awad, "Page migration support for disaggregated non-volatile memories," in *Proceedings of the International Symposium on Memory Systems*, 2019, pp. 417–427.

[31] V. R. Kommareddy, C. Hughes, S. Hammond, and A. Awad, "Investigating fairness in disaggregated non-volatile memories," in *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2019, pp. 104–110.

[32] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 2–13, 2009.

[33] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, "Phase-change technology and the future of main memory," *IEEE micro*, no. 1, pp. 143–143, 2010.

[34] E. K. Lee and C. A. Thekkath, "Petal: Distributed virtual disks," in *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, 1996, pp. 84–92.

[35] Z. Li, R. Zhou, and T. Li, "Exploring high-performance and energy proportional interface for phase change memory systems," *IEEE 20th International Symposium on High Performance Computer Architecture*, pp. 210–221, 2013.

[36] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch, "System-level implications of disaggregated memory," in *IEEE International Symposium on High-Performance Comp Architecture*. IEEE, 2012, pp. 1–12.

[37] S. Liu, K. Seemakhupt, G. Pekhimenko, A. Kolli, and S. Khan, "Janus: Optimizing memory and storage support for non-volatile memory systems," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture*. IEEE, 2019, pp. 143–156.

[38] K. T. Malladi, F. A. Nothaft, K. Periyathambi, B. C. Lee, C. Kozyrakis, and M. Horowitz, "Towards energy-proportional datacenter memory with mobile dram," in *2012 39th Annual International Symposium on Computer Architecture*. IEEE, 2012, pp. 37–48.

[39] A. Memaripour, J. Izraelevitz, and S. Swanson, "Pronto: Easy and fast persistence for volatile data structures," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 789–806.

[40] J. Meza, Y. Luo, S. Khan, J. Zhao, Y. Xie, and O. Mutlu, "A case for efficient hardware/software cooperative management of storage and memory," 2013.

[41] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with whisper," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 135–148, 2017.

[42] ——, "An analysis of persistent memory use with whisper," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 135–148, 2017.

[43] D. Narayanan and O. Hodson, "Whole-system persistence," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, 2012, pp. 401–410.

[44] I. B. Peng, R. Gioiosa, G. Kestor, P. Cicotti, E. Laure, and S. Markidis, "Rthms: A tool for data placement on hybrid memory system," *ACM SIGPLAN Notices*, vol. 52, no. 9, pp. 82–91, 2017.

[45] A. Prodromou, M. Meswani, N. Jayasena, G. Loh, and D. M. Tullsen, "Mempod: A clustered architecture for efficient and scalable migration in flat address space multi-level memories," in *2017 IEEE International Symposium on High Performance Computer Architecture*. IEEE, 2017, pp. 433–444.

[46] G. Psaropoulos, I. Oukid, T. Legler, N. May, and A. Ailamaki, "Bridging the latency gap between nvm and dram for latency-bound operations," in *Proceedings of the 15th International Workshop on Data Management on New Hardware*, 2019, pp. 1–8.

[47] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proceedings of the 36th annual international symposium on Computer architecture*, 2009, pp. 24–33.

[48] L. E. Ramos, E. Gorbatov, and R. Bianchini, "Page placement in hybrid memory systems," in *Proceedings of the international conference on Supercomputing*, 2011, pp. 85–95.

[49] O. Rodeh, J. Bacik, and C. Mason, "Btrfs: The linux b-tree filesystem," *ACM Transactions on Storage (TOS)*, vol. 9, no. 3, pp. 1–32, 2013.

[50] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls *et al.*, "The structural simulation toolkit," *SIGMETRICS Performance Evaluation Review*, vol. 38, no. 4, pp. 37–42, 2011.

[51] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 1, pp. 26–52, 1992.

[52] A. Rudoff, "Persistent memory programming," *Login: The Usenix Magazine*, vol. 42, pp. 34–40, 2017.

[53] M. I. Seltzer, K. Bostic, M. K. McKusick, C. Staelin *et al.*, "An implementation of a log-structured file system for unix." in *USENIX Winter*, 1993, pp. 307–326.

[54] J. Sim, A. R. Alameldeen, Z. Chishti, C. Wilkerson, and H. Kim, "Transparent hardware management of stacked dram as part of memory," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014, pp. 13–24.

[55] A. Tech, "Nvdimm messaging and faq," 2014.

[56] P. Von Behren, "Nvml: implementing persistent memory applications," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST). Santa Clara, California, USA*, 2015.

[57] J. Xu and S. Swanson, "Nova: A log-structured file system for hybrid volatile/non-volatile main memories," in *14th USENIX Conference on File and Storage Technologies FAST 16)*, 2016, pp. 323–338.

[58] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, "Nimble page management for tiered memory systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 331–345.

[59] M. Ye, C. Hughes, and A. Awad, "Osiris: A low-cost mechanism to enable restoration of secure non-volatile memories," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2018, pp. 403–415.

[60] H. Yoon, J. Meza, R. Ausavarungnirun, R. A. Harding, and O. Mutlu, "Row buffer locality aware caching policies for hybrid memories," in *2012 IEEE 30th International Conference on Computer Design (ICCD)*. IEEE, 2012, pp. 337–344.

[61] L. Zhang and S. Swanson, "Pangolin: A fault-tolerant persistent memory programming library," in *2019 USENIX Annual Technical Conference ATC 19)*, 2019, pp. 897–912.

[62] W. Zhang and T. Li, "Exploring phase change memory and 3d die-stacking for power/thermal friendly, fast and durable memory architectures," in *2009 18th International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2009, pp. 101–112.

[63] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: Closing the performance gap between systems with and without persistence support," in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2013, pp. 421–432.

[64] Y. Zhou, J. Philbin, and K. Li, "The multi-queue replacement algorithm for second level buffer caches." in *USENIX Annual Technical Conference, General Track*, 2001, pp. 91–104.

[65] K. A. Zubair and A. Awad, "Anubis: ultra-low overhead and recovery time for secure non-volatile memories," in *Proceedings of the 46th International Symposium on Computer Architecture*. ACM, 2019, pp. 157–168.