ThyNVM: Enabling Software-Transparent Crash Consistency in Persistent Memory Systems

Jinglei Ren[*†] Jishen Zhao[‡] Samira Khan[†′] Jongmoo Choi[+†] Yongwei Wu[*] Onur Mutlu[†]

[†]Carnegie Mellon University [*]Tsinghua University
[‡]University of California, Santa Cruz [′]University of Virginia [+]Dankook University

jinglei.ren@persper.com jishen.zhao@ucsc.edu samirakhan@cmu.edu
choijm@dankook.ac.kr wuyw@tsinghua.edu.cn onur@cmu.edu

ABSTRACT

Emerging byte-addressable nonvolatile memories (NVMs) promise persistent memory, which allows processors to directly access persistent data in main memory. Yet, persistent memory systems need to guarantee a consistent memory state in the event of power loss or a system crash (i.e., crash consistency). To guarantee crash consistency, most prior works rely on programmers to (1) partition persistent and transient memory data and (2) use specialized software interfaces when updating persistent memory data. As a result, taking advantage of persistent memory requires significant programmer effort, e.g., to implement new programs as well as modify legacy programs. Use cases and adoption of persistent memory can therefore be largely limited.

In this paper, we propose a hardware-assisted DRAM+NVM hybrid persistent memory design, Transparent Hybrid NVM (ThyNVM), which supports *software-transparent crash consistency* of memory data in a hybrid memory system. To efficiently enforce crash consistency, we design a new *dual-scheme checkpointing* mechanism, which efficiently overlaps checkpointing time with application execution time. The key novelty is to enable checkpointing of data at multiple granularities, cache block or page granularity, in a coordinated manner. This design is based on our insight that there is a tradeoff between the application stall time due to checkpointing and the hardware storage overhead of the metadata for checkpointing, both of which are dictated by the granularity of checkpointed data. To get the best of the tradeoff, our technique adapts the checkpointing granularity to the write locality characteristics of the data and coordinates the management of multiple-granularity updates. Our evaluation across a variety of applications shows that ThyNVM performs within 4.9% of an idealized DRAM-only system that can provide crash consistency at no cost.

MICRO-48, December 05 - 09, 2015, Waikiki, HI, USA
 ACM 978-1-4503-4034-2/15/12...$15.00
DOI: http://dx.doi.org/10.1145/2830772.2830802.

1. INTRODUCTION

Byte-addressable nonvolatile memory (NVM) technologies, such as STT-RAM [33, 36], PCM [64, 39], and ReRAM [2], promise *persistent memory* [77, 57, 4, 29], which is emerging as a new tier in the memory and storage stack. Persistent memory incorporates attributes from both main memory (fast byte-addressable access) and storage (data persistence), blurring the boundary between them. It offers an essential benefit to applications: applications can directly access persistent data in main memory through a fast, load/store interface, *without* paging them in/out of storage devices, changing data formats in (de)serialization, or executing costly system calls [45].

Persistent memory introduces an important requirement to memory systems: it needs to protect data integrity against partial/reordered writes to NVM in the presence of system failures (e.g., due to power loss and system crashes). Consider an example where two data structures A and B stored in NVM are updated to complete an atomic operation, and one of these updates to A or B reach the NVM first. If the system crashes or loses power after only one update completes, the in-memory structure can be left in an inconsistent state, containing partially updated data. With volatile memory, this is not an issue because the in-memory data is cleared each time the software program restarts. Unfortunately, with NVM, such inconsistent states persist even after a system restart. Consequently, persistent memory systems need to ensure that the data stored in the NVM can be recovered to a consistent version during system recovery or reboot after a crash, a property referred to as *crash consistency* [37, 15, 71]. Maintaining crash consistency used to be a requirement of solely the storage subsystem. Yet, by introducing data persistence in main memory, it becomes a challenge also faced by the memory subsystem.

Most prior persistent memory designs rely on programmers' manual effort to ensure crash consistency [77, 13, 76, 29, 83, 57]. Application developers need to explicitly manipulate data storage and movement in persistent memory, following particular programming models and software interfaces to protect data against write reordering and partial updates. This approach offers programmers full control over which pieces of data they would like to make persistent. Unfortunately, *requiring* all programmers to manage persistent memory via new interfaces is undesirable in several ways. First, significant effort is imposed on programmers: they need to implement new programs or modify legacy code

using new APIs, with clear declaration and partitioning of persistent and transient data structures. Second, applications with legacy code deployed in various systems will not be able to take advantage of the persistent memory systems. Third, as most previous persistent memory designs require transactional semantics to enable version control and write ordering for consistency, use cases of persistent memory becomes limited to transactional memory applications, which face challenges in scalability [10, 55, 17]. Fourth, persistent memory application implementations depend on specific software interfaces and runtime systems, which vary from one computer system to another [77, 13, 76, 29, 72]. This substantially degrades the portability of persistent memory programs across different systems.

Our **goal**, in this paper, is to devise an efficient *software-transparent* mechanism to ensure crash consistency in persistent memory systems. The hope is that such a design can enable more use cases of persistent memory (e.g., for unmodified legacy programs and non-transactional programs) and allow more programmers to use persistent memory *without* making modifications to applications to ensure crash consistency.

To this end, we propose *Transparent hybrid Non-Volatile Memory*, *ThyNVM*, a persistent memory design with *software-transparent crash consistency support* for a hybrid DRAM+NVM system. ThyNVM allows both transaction-based and non-transactional applications to directly execute on top of the persistent memory hardware with full support for crash consistency. ThyNVM employs a periodic hardware-assisted checkpointing mechanism to recover to a consistent memory state after a system failure. However, naive checkpointing can stall applications while it persists all the memory updates in the NVM. Reducing the application stall time due to the use of checkpointing is a critical challenge in such a system.

ThyNVM reduces checkpointing overhead in two ways. First, it overlaps checkpointing time with application execution time. Second, to do this efficiently, it dynamically determines checkpointing granularity of data by making the new observation that there is a tradeoff between the application stall time and metadata storage overhead of checkpointing. Checkpointing data at a small granularity incurs short stall time, yet generates a large amount of metadata that requires excessive hardware space. Checkpointing data at a large granularity, in contrast, yields a much smaller amount of metadata, yet incurs a long latency for creating checkpoints, potentially leading to longer application stall times. As a result, any single checkpointing scheme with a uniform granularity (either small or large) is suboptimal. To address this, we propose a *dual-scheme checkpointing mechanism*, which synchronously checkpoints sparse (low spatial locality) and dense (high spatial locality) persistent memory updates at cache block and page granularities, respectively. Compared to previous persistent memory designs that use logging [77, 13] or copy-on-write (CoW) [14, 76] to provide crash consistency, ThyNVM either significantly reduces metadata storage overhead (compared to logging) or greatly increases effective memory bandwidth utilization (compared to CoW).

This paper makes the following **contributions**:

(1) We propose a new persistent memory design with *software-transparent crash consistency support*. Our design allows both transaction-based and unmodified legacy applications to leverage persistent memory through the load/store interface.

(2) We identify a new, important *tradeoff between application stall time and metadata storage overhead* due to checkpointing of data at different granularities. Checkpointing data with a uniform granularity (e.g., cache block or page granularity) is suboptimal.

(3) We devise a new *dual-scheme checkpointing* mechanism which substantially outperforms a single-granularity checkpointing scheme. We observe that updates with low spatial locality are better checkpointed at cache block granularity, but updates with high spatial locality are better checkpointed at page granularity. Our design reduces stall time by up to 86.2% compared to uniform page-granularity checkpointing, while incurring only 26% of the hardware overhead (in the memory controller) of uniform cache-block-granularity checkpointing.

(4) We implement ThyNVM (with a formally-proven consistency protocol) and show that it guarantees crash consistency of memory data while providing high performance by efficiently overlapping checkpointing and program execution. Our solution achieves 95.1% of performance of an idealized DRAM-only system that provides crash consistency support at no cost.

2. MOTIVATION AND OBSERVATIONS

ThyNVM incorporates three essential design choices: (1) Crash consistency support is software-transparent, instead of software-based; (2) The crash consistency mechanism is based on checkpointing, instead of logging or copy-on-write; (3) Memory updates are checkpointed at two granularities, instead of a single granularity. This section discusses the reasons that result in these design choices.

2.1 Inefficiency of Software-based Crash Consistency Support

Most prior persistent memory designs [14, 77, 13, 83, 76] require application developers to explicitly define persistent data and use particular libraries. Crash consistency support is closely coupled with software semantics in these models. We illustrate the inefficiencies of such persistent memory designs with an example program shown in Figure 1. The figure compares two implementations of a function that updates an entry in a persistent hash table (the code is adapted from STAMP [47]). Figure 1(a) demonstrates an implementation that involves software transactions similar to previous designs [14, 77] and Figure 1(b) shows the code with software-transparent ThyNVM. The figures show that involving software in supporting crash consistency is undesirable and inefficient from several perspectives.

First, manually partitioning transient and persistent data is both burdensome for the programmer and error-prone. Figure 1(a) shows various indispensable program annotations: programmers need to carefully determine what data structures need to be persistent and how to manipulate them. For example, Line 3 in Figure 1(a), which reads a chain from the hash table, is correct only if the hash table implements a fixed number of chains. Otherwise, another concurrent thread can relocate the chains and the result of Line 3 would be incorrect. To avoid this, the programmer needs to use transactions, for example, to protect the chain addresses (accessed in Line 3) from updates of concurrent threads.

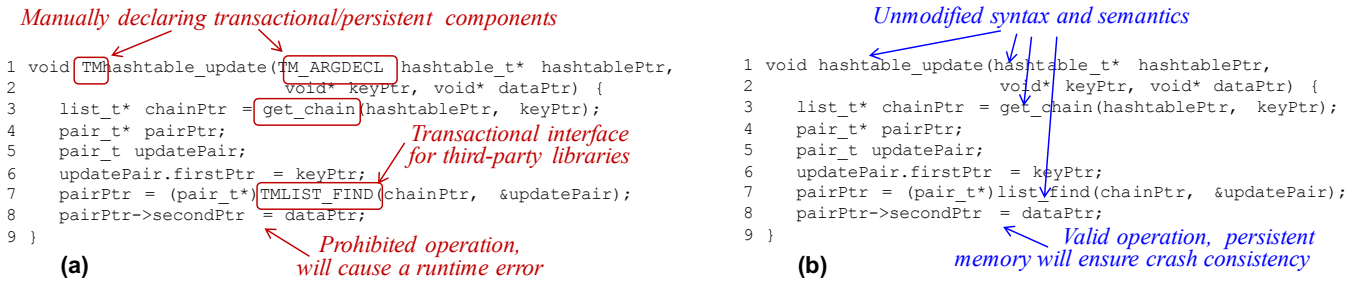


Figure 1: Code examples of updating an entry in a persistent hash table, implemented by (a) adopting a transactional memory interface or (b) employing software-transparent ThyNVM to ensure crash consistency.

Second, references between transient and persistent data in a unified memory space require careful management. However, software support for this issue is suboptimal. For example, NV-to-V pointers (nonvolatile pointers to volatile data, Line 8 in Figure 1(a)) can leave the NV pointer as a dangling pointer after a system failure, because the volatile data it points to is lost. NV-heaps [13] simply raises a runtime error that prohibits such pointers; this can impose substantial performance overhead by program halt or complex error handling.

Third, applications need to employ transactions to update persistent data, typically through a transactional memory (TM) interface (e.g., TinySTM [18] used by Mnemosyne [77]), as shown in Line 7 in Figure 1(a). Unfortunately, transactional memory, either hardware-based or software-based, has various scalability issues [10, 55, 17]. Furthermore, application developers need to implement or reimplement libraries and programs using new APIs, with nontrivial implementation effort. Programmers who are not familiar with such APIs require training to use them.

Finally, applications need to be written on top of specialized libraries, such as libmnemosyne [77] and NV-heaps [13]. This can substantially degrade the portability of persistent memory applications: an application implemented on top of one library will need to be reimplemented to adapt to another library. Compilers can mitigate programmers' burden on annotating/porting code to support persistent memory. However, without manual annotations, compilers instrument all (or most) memory reads and writes, imposing significant performance overhead. We evaluated STAMP transactional benchmarks [47] with GCC libitm [79] and found that the runtime overhead due to instrumentation alone can incur 8% performance degradation.

We adopt a usage model that allows software programs to directly access persistent memory without a special interface for crash consistency support. As illustrated in Figure 1(b), data structures can persist without any code modification.

2.2 Inefficiency of Logging and Copy-on-Write

Most previous software-based persistent memory designs adopt logging [77, 13] or copy-on-write (CoW) [14, 76] to ensure crash consistency: Logging-based systems maintain alternative copies of original data in a log, which stores new data updates (redo logging) or old data values (undo logging). CoW-based systems always create a new copy of data on which the updates are performed. Unfortunately, both techniques can impose undesirable overhead towards developing software-transparent crash consistency support. Logging can consume a much larger NVM capacity than the original data, because each log entry is a tuple consisting of both data and the corresponding metadata (e.g., the address of the data), and typically every memory update has to be logged [77, 13].[1] In addition, log replay increases the recovery time on system failure, reducing the fast recovery benefit of using NVM in place of slow block devices. CoW has two drawbacks. First, the copy operation is costly and incurs a long stall time [69]. Second, it inevitably copies unmodified data and thus consumes extra NVM bandwidth, especially when updates are sparse [70].

In contrast, checkpointing [78] is a more flexible method. In checkpointing, volatile data is periodically written to NVM to form a consistent snapshot of the memory data. We adopt it to overcome inefficiencies associated with logging and copy-on-write. However, it is critical to minimize the overheads of checkpointing, and to this end, we develop new mechanisms in this work.

2.3 Tradeoff between Checkpointing Latency and Metadata Storage Overhead

Checkpointing involves taking a snapshot of the working copy of data (i.e., the copy of data that is actively updated) and persisting it in NVM (as a checkpoint). There are two concerns in checkpointing: (1) latency of checkpointing the working copy of data, and (2) metadata overhead to track where the working copy and checkpoint of data are. Ideally, we would like to minimize both. However, there are complex tradeoffs between these two that one should consider when designing an efficient checkpointing mechanism.

First, the metadata overhead is determined by the granularity at which we keep track of the working/checkpoint data, which we call the *checkpointing granularity*. By using a large checkpointing granularity (e.g., page granularity), we can keep track of the location of large amounts of data with only a small amount of metadata.[2]

Second, checkpointing latency is affected by the *location of the working copy* of data. In a hybrid memory system, the working copy can be stored in either DRAM or NVM. We analyze the two options. (1) *DRAM*. Because DRAM writes are faster than NVM writes, "caching" the working copy in DRAM can improve the performance of write operations coming from the application. However, if we cache the working copy in DRAM, this working copy has to be *written back to NVM during checkpointing*, resulting in long checkpointing latency. (2) *NVM*. We can significantly reduce the

[1] An improved variant of logging [73] uses an indexing structure to coalesce updates.

[2] One can think of this similar to tag overhead in CPU caches: small blocks (small granularity) have large tag overhead, and large blocks (large granularity) have small tag overhead.

checkpointing latency by storing the working copy in NVM and updating it in-place in NVM when a memory write is received. Because the working copy in NVM is already persistent, checkpointing becomes fast: we only need to persist the metadata during checkpointing. However, keeping the working copy in NVM and updating it in-place requires that we *remap* the working copy to a new location in NVM that is different from the checkpoint data, upon a write operation from the application, such that the application can operate on the new working copy without corrupting the checkpoint. The speed of this remapping depends on the granularity of data tracking (i.e., checkpointing granularity): a small granularity leads to fast remapping, enabling the application to quickly update the data, yet a large granularity leads to slow remapping (because it requires copying a significant amount of data, e.g., an entire page), which can delay the write operation and thus stall the application for a long time.

Table 1 summarizes the impact of checkpointing granularity and location of working copy of data on the tradeoff between metadata overhead and checkpointing latency.

		Checkpointing granularity	
		Small (cache block)	Large (page)
Location of working copy	DRAM (based on writeback)	❶ *Inefficient* ✘ **Large metadata overhead** ✘ **Long checkpointing latency**	❷ *Partially efficient* ✔ **Small metadata overhead** ✘ Long checkpointing latency
	NVM (based on remapping)	❸ *Partially efficient* ✘ Large metadata overhead ✔ **Short checkpointing latency** ✔ Fast remapping	❹ *Inefficient* ✔ Small metadata overhead ✔ Short checkpointing latency ✘ **Slow remapping** (on the critical path)

Table 1: Tradeoff space of options combining checkpointing granularity choice and location choice of the working copy of data. The table shows four options and their pros and cons. Boldfaced text indicates the most critical pro or con that determines the efficiency of an option.

We consider the combination of small checkpointing granularity and storing working copy in DRAM (❶) *inefficient* because this incurs large metadata overhead (as apposed to using large granularity to track data ❷) while also leading to long checkpointing latency (as opposed to keeping the working copy directly in NVM ❸).

We consider the combination of large checkpointing granularity and storing working copy in NVM (❹) *inefficient* because it incurs prohibitive latency for remapping in-the-flight updates in NVM, which is *on the critical path* of application's execution. For example, if we track data at the page granularity, to remap a cache-block-sized update in NVM, we have to copy *all other cache blocks* in the page as well into a new working copy, before the application can perform a write to the page. This remapping overhead is on the critical path of a store instruction from the application.

Based on these complex tradeoffs, we conclude that no single checkpoint granularity or location of working copy is best for all purposes. In the schemes we introduce in §3, we design a multiple-granularity checkpointing mechanism to achieve the best of multiple granularities and different locations of working copy. In particular, we aim to take advantage of the two *partially efficient* combinations from Table 1 that have complementary tradeoffs: (1) Small granularity checkpointing that keeps working data in NVM by remapping writes in NVM (❸). This motivates our *block remapping* scheme (§3.2). (2) Large granularity checkpointing that keeps working data in DRAM and writes back updated pages into NVM (❷). This motivates our page writeback scheme (§3.3). We design judicious cooperation mechanisms to obtain the best of both schemes (§3.4), thereby achieving *both* small metadata overhead *and* short checkpointing latency.

3. ThyNVM DESIGN

ThyNVM provides software-transparent crash consistency of DRAM+NVM hybrid persistent memory by adopting hardware-based mechanisms. ThyNVM enforces crash consistency over all memory data transparently to application programs. To address the tradeoffs of persisting data at different granularities, we propose a novel dual-scheme checkpointing mechanism, which checkpoints memory data updates at two granularities simultaneously: sparse writes are checkpointed at cache block granularity using a *block remapping* scheme (§3.2); dense writes are checkpointed at page granularity using a *page writeback* scheme (§3.3). In addition, we devise mechanisms to coordinate the two checkpointing schemes (§3.4).

Architecture. Figure 2 depicts an overview of the ThyNVM architecture. DRAM and NVM are deployed on the memory bus and mapped to a single physical address space exposed to the OS. We modify the memory controller to incorporate a checkpointing controller and two address translation tables, the block translation table (BTT) and the page translation table (PTT), which maintain metadata of memory access at respectively cache block and page granularities.

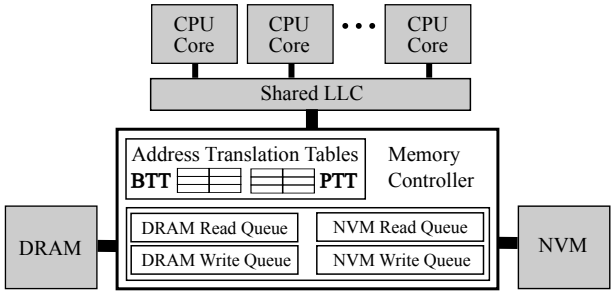


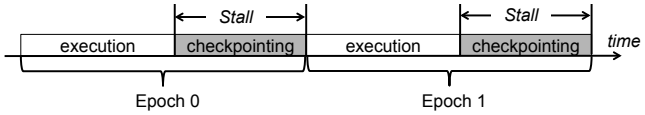
Figure 2: Architecture overview of ThyNVM.

3.1 Assumptions and Definitions

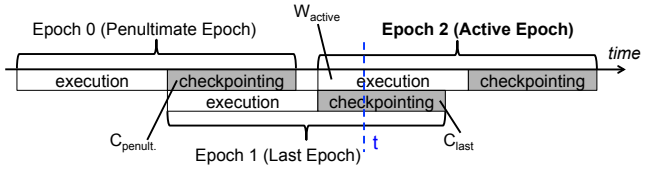
Failure model. ThyNVM allows software applications to resume CPU execution from a consistent checkpoint of memory data after system failures, such as system crashes or power loss. To this end, we periodically checkpoint memory data updates and the CPU state, including registers, store buffers and dirty cache blocks. Our checkpointing schemes protect persistent memory data and CPU states from corruption on system failures.

Epoch model. We logically divide program execution time into successive time periods, called *epochs* (Figure 3). Each epoch has an *execution phase* and a *checkpointing phase*. The execution phase updates the working copy of memory data and CPU state, while the checkpointing phase creates a checkpoint of memory data and CPU state.

Alternating execution and checkpointing phases without overlap (Figure 3 (a)) incurs significant performance degradation, as checkpointing can consume up to 35.4% of the entire program execution time with memory-intensive workloads (§5.3). Therefore, ThyNVM adopts an epoch model



(a) An epoch model with "stop-the-world" checkpointing.



(b) ThyNVM epoch model which overlaps checkpointing and execution. The model maintains three versions of data from the last three epochs: active working copy (W_{active}), last checkpoint data (C_{last}), and penultimate checkpoint data (C_{penult}).

Figure 3: Epoch models in checkpointing systems: stop-the-world vs. ThyNVM.

that overlaps the checkpointing and the execution phases of *consecutive* epochs [73] to mitigate such performance degradation. We define three consecutive epochs as *active*, *last*, and *penultimate* epochs. As Figure 3 (b) shows, in order to eliminate the stall time due to checkpointing, the active epoch (Epoch 2) starts its execution phase as soon as the last epoch (Epoch 1) starts its checkpointing phase. The last epoch (Epoch 1) can start its checkpointing phase only after the checkpointing phase of the penultimate epoch (Epoch 0) finishes. If the checkpointing phase of an epoch is smaller than the overlapping execution phase of the next epoch, then the checkpointing phase is not on the critical path of program execution (as is the case in Figure 3 (b)). However, the execution phase should locate and update the correct version of data in each epoch (which we discuss next).

Data versions. The overlapping of execution and checkpointing allows two consecutive epochs to concurrently access the same piece of data. This imposes two challenges in maintaining crash consistency: (1) the overlapping checkpointing and execution phases can overwrite data updates performed by each other, so we need to *isolate* data updates of different epochs; (2) a system failure can corrupt data of both the active and the last epochs, so we need to maintain a safe copy of data from the penultimate epoch. To address both challenges, ThyNVM maintains three versions of data across consecutive epochs: the *active working copy* W_{active}, the *last checkpoint* C_{last}, and the *penultimate checkpoint* C_{penult} (C_{last} and C_{penult} include checkpoints of both memory data and CPU states). For instance, as shown in Figure 3 (b), ThyNVM preserves the checkpoints made in Epoch 0 (C_{penult}) and Epoch 1 (C_{last}), while the system is executing Epoch 2 (which updates W_{active}). ThyNVM discards the checkpoints made in Epoch 0 *only after* the checkpointing phase of Epoch 1 completes. A system failure at time t can corrupt both the working copy updated in Epoch 2 and the checkpoint updated in Epoch 1. This is exactly why we need to maintain C_{penult}, the checkpoint updated in Epoch 0. ThyNVM can always roll back to the end of Epoch 0 and use C_{penult} as a safe and consistent copy.

3.2 Checkpointing with Block Remapping

The *block remapping* scheme checkpoints data updates in NVM at cache block granularity. This scheme enables update of the working copy of a block *directly* in NVM during the execution phase. It does so by *remapping* the new working copy (of the active epoch) to another address in NVM and persisting only the metadata needed to locate the last checkpoint in the checkpointing phase. Hence, there is no need to move the data of the block when it is being checkpointed. Instead, during checkpointing, the block that is already updated directly in NVM simply transitions from being the working copy to being the last checkpoint. Therefore, the block remapping scheme significantly reduces the checkpointing latency. In our work, we propose to checkpoint data updates with *low spatial locality* using the block remapping scheme. These updates are mostly random, sparse and of small sizes (i.e., they usually touch a single block or few blocks in a page). Therefore, it is efficient to checkpoint them individually at cache block granularity. In contrast, checkpointing such updates at page granularity would be very inefficient, as they typically dirty only a small portion of each page, and writing back the entire page to NVM even though it is mostly unchanged can greatly hamper performance.

In this scheme, the working copy W_{active} is directly written into NVM during the execution phase of each epoch. Basically, ThyNVM records the mappings between data blocks and the addresses of their W_{active} in the block translation table (BTT). A read request can identify the valid address of the working copy by looking up the BTT. In an active epoch, updates to the same block are directed to a new address allocated for the block for the epoch, i.e., the address of the new working copy W_{active}. These updates are coalesced at the new address. This working copy becomes the checkpointed copy C_{last} by simply persisting the corresponding BTT mapping during the checkpointing phase. ThyNVM persists the BTT in NVM at the beginning of each checkpointing phase.

3.3 Checkpointing with Page Writeback

The *page writeback* scheme checkpoints data updates at page granularity. It caches hot pages in DRAM during the execution phase of an epoch and *writes back dirty pages* to NVM during the checkpointing phase. In this work, we propose to checkpoint data updates with *high spatial locality* using the page writeback scheme, because those updates are typically sequential or clustered, occupying (almost) full pages. Managing such updates at page granularity, instead of at cache block granularity, reduces the amount of metadata storage that the hardware needs to keep track of to manage such updates (as described in §2.3).

The management of DRAM using the page writeback scheme is different from the management of a conventional writeback CPU cache in two ways. First, memory accesses during the execution phase do *not* trigger any page replacement or eviction; instead, dirty pages are written back to NVM *only* during checkpointing. Second, while ThyNVM writes back a dirty page from DRAM to NVM during the checkpointing phase, in-flight updates to the same page from the execution phase of the *next* epoch *cannot* directly overwrite the same physical page (in order to maintain isolation of updates from different epochs). Thus, ThyNVM directs each dirty page to a different address during checkpointing. It employs the *page translation table*, PTT, to track the address mappings for pages that are in DRAM. At the end of the checkpointing phase, ThyNVM atomically persists the PTT to NVM. This denotes the end of an epoch.

3.4 Coordinating the Two Schemes

We devise mechanisms to coordinate the two schemes, i.e., block remapping and page writeback, in order to (1) reduce application stall time due to page writeback based checkpointing and (2) adapt checkpointing schemes to applications' dynamically changing memory access behavior.

Reducing application stall time due to page writeback. Block remapping allows the system to persist *only metadata* in each checkpointing phase. Therefore, it typically completes much faster than page writeback in the checkpointing phase of an epoch. In this case, page writeback based checkpointing can potentially block the execution of the program, because the subsequent epochs cannot update the DRAM pages that are not yet checkpointed in NVM. To mitigate this issue, ThyNVM allows the system to proactively start executing the next epoch by using the block remapping scheme to temporarily accommodate incoming writes (from the next epoch) that should have otherwise been managed by page writeback. By switching to block remapping when page writeback blocks progress, we hide the stall time of checkpointing due to page writeback, write to DRAM and NVM in parallel, and increase memory bandwidth utilization.

Switching between the two schemes. ThyNVM adapts the checkpointing scheme dynamically to match changing spatial locality and write intensity characteristics of the data. The memory controller determines whether a page needs to switch its checkpointing scheme from one to the other at the beginning of each epoch, based on spatial locality of updates to the page. Switching checkpointing schemes involves both metadata updates and data migration. To switch from page writeback to block remapping, we discard the corresponding entry in the PTT, and move the page in DRAM to the NVM region managed by block remapping. As a result, the next time any block of this physical page is updated, the block remapping scheme handles the block and adds its metadata to the BTT accordingly. To switch from block remapping to page writeback, we need to add an entry to the PTT to store the metadata of the page, combine all blocks of the page from potentially different locations (using the BTT to locate them), and copy the entire page to the corresponding newly-allocated location in the DRAM. The memory controller handles all these operations, whose overheads can be hidden by the execution phase.

4. IMPLEMENTATION

This section describes our implementation of ThyNVM. We first describe the address space layout and address space management (§4.1) and metadata management (§4.2) mechanisms. We then discuss how ThyNVM services loads and stores (§4.3) and how it flushes data during checkpointing (§4.4). We finally describe how ThyNVM handles memory data during system recovery (§4.5).

4.1 Address Space Layout and Management

Our ThyNVM implementation has a particular address space layout to store and manage multiple versions of data. The software's view of the *physical address space* is different from the memory controller's view of the *hardware address space*, as shown in Figure 4. Portions of the hardware address space, which is larger than the physical address space, are visible only to the memory controller such that it can store checkpoints of data and processor states. We describe different regions of the hardware address space and their purposes below.

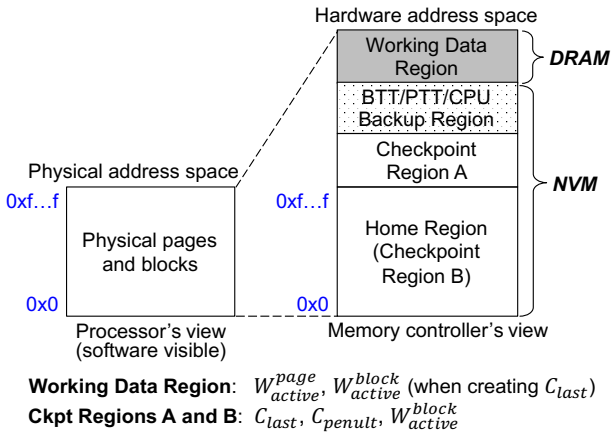


Figure 4: ThyNVM address space layout.

Our address space management mechanisms have two key goals: (1) store different versions of data in different memory locations in the hardware address space and (2) expose a single consistent version of the data to the processor upon a load/store access or during system recovery. We discuss how ThyNVM achieves these goals in this section.

Data regions in memory and version storage. There are two types of data in ThyNVM. The first type is data that is *not subject to checkpointing*, i.e., data that is *not* updated in any of the last three epochs. This type of data has only a single version as it is *not* subject to updates. As such, it is not kept track of by the BTT/PTT tables. When this data is accessed by the processor, the physical address of the data is directly used, without being remapped by the BTT/PTT, to locate the data in memory. We call the portion of memory where such data is stored as *Home Region* (as shown in Figure 4). For example, read-only data always stays in the Home Region of the hardware address space.

The second type is data that is *subject to checkpointing*, i.e., data that *is* updated in *at least* one of the last three epochs. This data can have multiple versions, e.g., W_{active}, C_{last}, or C_{penult} (as explained in §3.1). Metadata in the BTT/PTT tables keeps track of the location of this data. BTT/PTT tables map the *physical address* of this data to a *hardware address* that is potentially different from the physical address. Thus, when this data is accessed by the processor, ThyNVM's mechanisms direct the processor to the appropriate version of the data.

ThyNVM maintains three data regions in DRAM and NVM, each storing one version of such data that is subject to checkpointing: W_{active}, C_{last}, or C_{penult}. As shown in Figure 4, these regions include a *Working Data Region* in DRAM[3], and two *Checkpoint Regions* in NVM. Working Data Region stores the active working copy of data managed by *page writeback*, W_{active}^{page}. Note that, in the checkpointing

[3]In the implementation we describe, we assume that the Working Data Region is mapped to DRAM to obtain the performance benefits of DRAM for data that is actively updated. Other implementations of ThyNVM can distribute this region between DRAM and NVM or place it completely in NVM. We leave the exploration of such choices to future work.

phase of an epoch, such data managed by page writeback can be temporarily handled by *block remapping* to avoid stalling the program execution as described in §3.4. In that case, updates to such data are kept track of by the BTT and remapped *also* to this Working Data Region.

Checkpoint Regions A and B store checkpoint data C_{last} and C_{penult} in an interleaved manner: if C_{penult} is stored in Checkpoint Region A, ThyNVM writes C_{last} into Checkpoint Region B, and vice versa. Note that Checkpoint Region B is the same as the Home Region to save memory space as well as entries in BTT/PTT: if the data is *not* subject to checkpointing, this region stores the only copy of the data so that it does not require any address mapping in BTT or PTT; if the data *is* subject to checkpointing, this region stores the appropriate checkpoint version, and its metadata is stored in BTT or PTT accordingly.

Storage of W_{active}^{block}. We now discuss where the data managed by the *block remapping* scheme is stored. This data may reside in either one of the Checkpoint Regions or the Working Data Region, depending on whether or not the last epoch has completed its checkpointing phase, as we describe in the next two paragraphs.

As described in §3.2, the block remapping scheme aims to write the working copy of data (W_{active}^{block}) directly to NVM, in order to reduce the checkpointing latency of data updates with low spatial locality. Our ThyNVM implementation realizes this by overwriting C_{penult}^{block} in one of the Checkpoint Regions with W_{active}^{block}, during the execution phase of an epoch, when it is safe to do so. As long as the last epoch has completed its checkpointing phase (i.e., the C_{last} version of data is safe to recover to), we can overwrite C_{penult} of data (recall this from the epoch model and data versions in §3.1). Thus, Checkpoint Regions also contain W_{active}^{block} of data written by the block remapping scheme.[4]

However, when the last epoch has not yet completed its checkpointing phase, i.e., when C_{penult} is unsafe to overwrite, ThyNVM*cannot* update W_{active}^{block} directly in NVM. This is because overwriting C_{penult} would result in loss of the consistent system state to recover to, as we discussed in §3.1. In this case, our ThyNVM implementation *temporarily* stores W_{active}^{block} in the Working Data Region. When the active epoch starts its checkpointing phase, the W_{active}^{block} that is temporarily stored in the Working Data Region is written to one of the Checkpoint Regions in NVM (and becomes C_{last}).

Storage for BTT/PTT and CPU state backups. Finally, ThyNVM hardware address space dedicates part of the NVM area, *BTT/PTT/CPU Backup Region*, to store BTT/PTT and CPU state backups.

Mapping of physical and hardware address spaces. To provide a single consistent software-visible version of the data,[5]ThyNVM creates a mapping between the physical and hardware address spaces, managed by, respectively, the operating system and the memory controller. The hardware address space comprises the actual DRAM and NVM device addresses, which are visible only to the memory controller. The physical address space, exposed to the software (through the operating system), is mapped by the memory controller to the appropriate portions of the hardware address space. We discuss what constitutes the software-visible version of data.

For data that is not subject to checkpointing, the software-visible version is in the Home Region of the hardware address space. The hardware address of this data is simply the physical address concatenated with some constant bits (to accommodate the larger hardware address space).

For data that is subject to checkpointing, the software-visible version is as follows (the BTT/PTT tables ensure that the physical address is mapped to the appropriate location in the hardware address space).

$$\begin{cases} W_{active}, & \text{if } W_{active} \text{ exists} \\ C_{last}, & \text{if } W_{active} \text{ does not exist} \end{cases}$$

Hence, software can read or write to the location that stores either the working copy (W_{active}) or the last checkpoint copy (C_{last}) via loads and stores, for data that is subject to checkpointing, depending on whether or not the working copy exists. Recall that when the data is not modified in the active epoch, we don not have a working copy (W_{active}) for it, and the last checkpoint (C_{last}) actually houses the latest copy of data that can be safely accessed by the software.

Upon recovery from a system crash, W_{active} stored in DRAM is lost. The following version becomes restored, i.e., software visible after recovery (depending on the state of the checkpointed BTT/PTT entries).

$$\begin{cases} C_{last}, & \text{if the last checkpoint has completed} \\ C_{penult}, & \text{if the last checkpoint is incomplete} \end{cases}$$

During recovery, ThyNVM uses the checkpointed BTT/PTT tables to perform the restoration of C_{last} or C_{penult}.

4.2 Metadata Management

ThyNVM maintains metadata in two address translation tables, BTT and PTT. These tables contain information that enables three things: (1) translation of the physical address of each *memory request* issued by the processor into the appropriate hardware address, (2) steering of the data updates *for checkpointing* to appropriate hardware addresses, and (3) determination of when to migrate data between the two checkpointing schemes.

Both BTT and PTT have five columns (Figure 5): (1) a block/page index consisting of the higher-order bits of the physical address, which has 42 bits in BTT and 36 bits in PTT; (2) a two-bit Version ID denoting W_{active}, C_{last}, or C_{penult}; (3) a two-bit Visible Memory Region ID denoting which memory region the software-visible version is located in; (4) a one-bit Checkpoint Region ID denoting the checkpoint region for C_{last} (which implies that C_{penult} is located in the other checkpoint region); (5) a six-bit store counter that

[4]Note that we do not have a separate region in NVM for storing *only* W_{active}^{block} because W_{active}^{block} simply becomes C_{last} after the metadata associated with it is checkpointed during the checkpointing phase of the active epoch. Thus, checkpointing the active epoch automatically results in the turning of the working copy of a block into a checkpointed copy, without requiring any data movement. If there were to be a separate region that stored the working copy of blocks managed by block remapping, checkpointing them would require the movement of each block to one of the separate Checkpoint Regions, not only requiring data movement, hence increasing checkpointing latency, but also wasting memory capacity.

[5]The software-visible version of data is the data version that can be read or written to by the software via load and store instructions.

Physical Block/Page Index	Version ID	Visible Memory Region ID	Checkpoint Region ID	Store Counter
42 bits for BTT/ 36 bits for PTT	2 bits	2 bits	1 bit	6 bits

Figure 5: Fields in BTT and PTT.[6]

records the number of writes performed on the block/page during the current epoch to determine its write locality.[6]

To translate a requested physical address to the hardware address, the required information includes (a) the memory region that a request should be serviced from and (b) the mapping between physical and hardware addresses in that region. The memory controller uses the *Visible Memory Region ID* to identify the memory region where the software-visible version of the requested data is located. To determine the hardware address with simple logic, we enforce that the offset of each table entry from the first entry is the same as the offset of each block/page from the first block/page in the corresponding memory region. Therefore, we can use the offset of a table entry to calculate the hardware address of the block/page within the memory region.

To steer the data updates for checkpointing to appropriate hardware addresses, the memory controller uses the Checkpoint Region ID. We use the offset of the table entry to determine the hardware address the same way as mentioned in the above paragraph.

To determine when to migrate data between the two checkpointing schemes tailored for different amounts of spatial locality, the memory controller identifies the spatial locality of data based on the values of the *store counters* in BTT/PTT. The memory controller collects the counter values at the beginning of each epoch, and resets them after deciding the checkpointing scheme for the pages. A large number of stores to the same physical page within one epoch, beyond a predefined threshold, indicates that the page has high spatial locality and high write intensity. As such, we employ page writeback for checkpointing this page in a new epoch. A store counter value that is lower than another predefined threshold indicates low spatial locality, i.e., that we need to adopt block remapping in the new epoch. We empirically determined these threshold values and set them to 22 and 16 for switching from block remapping to page writeback and the other way around, respectively.

Checkpointing BTT and PTT. The metadata identifies the latest consistent checkpoint. Hence, BTT and PTT need to persist across system failures. In each checkpointing phase, we checkpoint BTT and PTT entries in the BTT/PTT/CPU Backup Region of the NVM (Figure 4). One challenge is to atomically persist BTT/PTT into NVM. To accomplish this, we use a single bit (stored in the BTT/PTT/CPU Backup Region) that indicates the completion of the checkpointing of BTT/PTT.

Size of BTT and PTT. The number of BTT entries depends on the write intensity of the working set (as a BTT entry is created upon the first write to a block). The number of PTT entries depends on the size of DRAM (as the PTT needs to have an entry for each page in DRAM to enable maximum utilization of DRAM). In case certain workloads require gigabytes of DRAM, large PTT storage overhead can be tolerated by virtualizing the PTT in DRAM and caching hot entries in the memory controller [44, 80, 8]. We perform a sensitivity study of system performance to various BTT sizes (Figure 12). The total size of the BTT and PTT we use in our evaluations is approximately 37KB. Note that as memory controllers become more sophisticated to manage hybrid memories and incorporate new features, a trend in both academia (e.g., [74, 35, 81, 75, 51, 50, 1, 82, 34]) and industry (e.g., the logic layer of HMC [56] and the media controller of HP's The Machine [27]), the hardware cost of our proposal will be even more affordable.

[6]Because not all combinations of fields (Version ID, Visible Memory Region ID, Checkpoint Region ID) exist, they can be compressed into seven states, as articulated in our online document [65]. The state machine protocol saves hardware space and simplifies hardware logic.

4.3 Servicing Loads and Stores

To service a load request, ThyNVM identifies the software-visible data by querying the BTT/PTT using the physical address of the request. If the physical address misses in both BTT and PTT, the corresponding visible data is in the Home Region of NVM. To service a store request, as illustrated in Figure 6 (a), ThyNVM first uses the PTT to determine whether to use block remapping or page writeback. ThyNVM performs block remapping for all requests that miss in the PTT, regardless of whether or not the physical address hits in the BTT. A physical address that misses in both tables indicates that the corresponding data block is in the Home Region. In that case, a new entry is added to the BTT for the physical address to record the metadata update of the store request. In the case of BTT/PTT overflow, where no entry is either available or can be replaced, ThyNVM starts checkpointing the current epoch and begins a new epoch. This enables the BTT/PTT entries belonging to the penultimate checkpoint to be freed.

Accommodating initial data accesses. Initially, when the system starts to execute, both address translation tables are empty; all visible data is in the Home Region of NVM. As the PTT is empty, we employ block remapping to checkpoint the data updates of the first epoch. Over time, ThyNVM identifies and switches data with high spatial locality to use the page writeback scheme.

Checkpointing. When checkpointing the memory data, ThyNVM ensures that the associated metadata is checkpointed *after* the corresponding data. In particular, we adopt the following checkpointing order, as illustrated in Figure 6 (b): (1) write W_{active}^{block} from DRAM to NVM (if it is temporarily stored in DRAM, as discussed in §4.1); (2) checkpoint BTT in NVM; (3) write back W_{active}^{page} from DRAM to NVM; (4) checkpoint PTT in NVM.

4.4 Data Flush

To guarantee consistent checkpoints, ThyNVM flushes data out of the processor into the NVM during the checkpointing phase of each epoch. We adopt a hardware-based mechanism to perform the data flush. The memory controller notifies the processor when an execution phase is completed. After receiving the notification, the processor stalls its execution and issues a flush operation. This operation writes all register values into a special NVM region, flushes store buffers, and cleans the dirty cache blocks by initiating writebacks without invalidating the data in the cache (to preserve locality of future accesses, as described in [68] and similarly to the functionality of Intel's CLWB

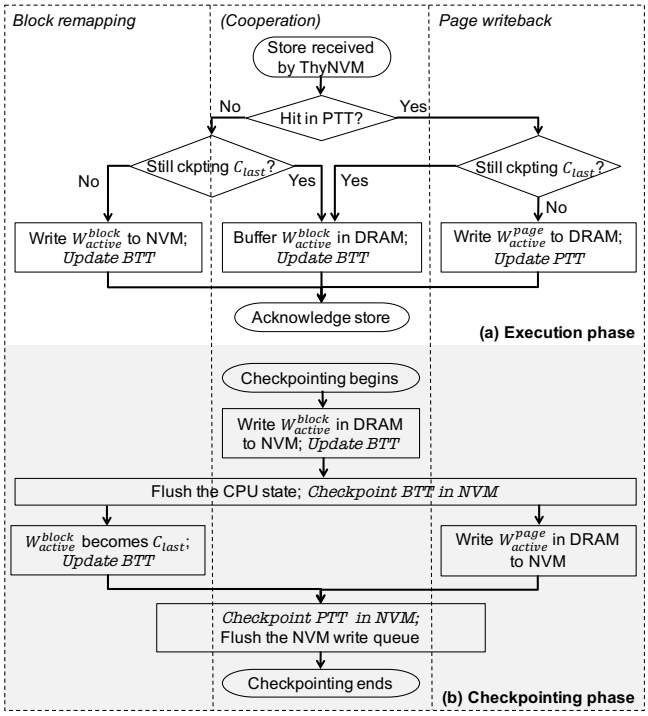


Figure 6: Control flow of **(a)** servicing a store request in the execution phase of an epoch and **(b)** checkpointing memory data in the checkpointing phase of the epoch. To show the relationship between the stored data in (a) and the checkpointed data in (b), we draw three background frames denoting both checkpointing schemes and their cooperation: within each background frame, the data stored in (a) goes through the corresponding processes in (b) when being checkpointed.

instruction [28]). At the end of a checkpointing phase, ThyNVM also flushes the NVM write queue of the memory controller to ensure that all updates to the NVM are complete (the last step of checkpointing in Figure 6 (b)). Once the flush of the NVM write queue is complete, the checkpoint is marked as complete (by atomically setting a single bit in the BTT/PTT/CPU Backup Region of NVM).

4.5 System Recovery

With a system that employs ThyNVM to maintain crash consistency, recovery involves three major steps to roll back the persistent memory and the processor state to the latest checkpoint. First, the memory controller reloads the checkpointed BTT and PTT, by copying their latest valid backup stored in NVM. As such, we recover the metadata information that will be used to recover the data from the latest consistent checkpoint (i.e., the software-visible version defined in §4.1). Second, the memory controller restores software-visible data managed by page writeback. As the software-visible version of such pages should be stored in the Working Data Region (i.e., DRAM, in our implementation), the memory controller copies the corresponding checkpointed data in NVM to the Working Data Region of the hardware address space. Third, after both metadata and memory data are recovered, the system reloads the checkpointed processor registers from their dedicated memory space in the NVM.

A formal verification of the correctness of our consistency protocol and recoverability of memory data is provided online [66].

Once the memory data and the processor state are restored, system execution resumes. Because we do not record external device states (e.g., in the network card), the resumed program may face device errors due to loss of these states. Therefore, ThyNVM exposes system failure events to software programs as device errors upon system recovery. We rely on existing exception/error handling code paths of programs to deal with these errors. It is important to note that ThyNVM provides mechanisms for consistent *memory state* recovery, and it is *not* our goal to recover from external device state failures, which can be handled via other exception handling and recovery mechanisms. We discuss this issue in more detail in §6.

5. EVALUATION

5.1 Experimental Setup

In this section, we describe our simulation infrastructure, processor and memory configurations, and benchmarks. Our experiments are conducted using the cycle-level micro-architectural simulator gem5 [7]. We use gem5's detailed timing model for the processor and extend its memory model [26] to implement ThyNVM mechanisms as described in §4. Table 2 lists the detailed parameters and architectural configuration of the processor and the memory system in our simulations. We model DRAM and NVM with the DDR3 interface. We simulated a 16 MB DRAM and 2048/4096 BTT/PTT entries. The epoch length is limited to 10 ms (comparable to [61, 73]). The source code of our simulator and our simulation setup are available at `http://persper.com/thynvm`.

Processor	3 GHz, in-order
L1 I/D	Private 32*KB*, 8-way, 64*B* block; 4 cycles hit
L2 cache	Private 256*KB*, 8-way, 64*B* block; 12 cycles hit
L3 cache	Shared 2*MB*/core, 16-way, 64*B* block; 28 cycles hit
Memory Timing	DDR3-1600
	DRAM: 40 (80) ns row hit (miss).
	NVM: 40 (128/368) ns row hit (clean/dirty miss).
	BTT/PTT: 3 ns lookup

Table 2: System configuration and parameters. NVM timing parameters are from [81, 38, 45].

Evaluated Systems. We compare ThyNVM with four different systems.
(1) *Ideal DRAM*: A system with only DRAM as main memory, which is assumed to provide crash consistency *without any overhead*. DRAM is of the same size as ThyNVM's physical address space.
(2) *Ideal NVM*: A system with only NVM as main memory, which is assumed to provide crash consistency *without any overhead*. NVM is of the same size as ThyNVM's physical address space.
(3) *Journaling*: A hybrid NVM system using journaling [16, 41] to provide the crash consistency guarantee. *Journaling* is one possible implementation of the general *logging* technique [77, 13] (§2.2). Our implementation follows [3]. A journal buffer is located in DRAM to collect and coalesce updated blocks. At the end of each epoch, the buffer is written back to NVM in a backup region, before it is committed

in-place. This mechanism uses a table to track buffered dirty blocks in DRAM. The size of the table is the same as the combined size of the BTT and the PTT in ThyNVM. This system is denoted as *Journal* in the figures.
(4) *Shadow paging*. A hybrid NVM system using shadow paging [6] to provide the crash consistency guarantee. *Shadow paging* is one possible implementation of the general *copy-on-write (CoW)* technique [14, 76] (§2.2). It performs copy-on-write on NVM pages and creates buffer pages in DRAM. When DRAM buffer is full, dirty pages are flushed to NVM, without overwriting data in-place. The size of DRAM in this configuration is the same as ThyNVM's DRAM. This system is denoted as *Shadow* in the figures.

Benchmarks. We evaluate three sets of workloads.
(1) *Micro-benchmarks* with different memory access patterns. In order to test the adaptivity of ThyNVM to different access patterns, we evaluate three micro-benchmarks with typical access patterns. (i) *Random*: Randomly accesses a large array. (ii) *Streaming*: Sequentially accesses a large array. (iii) *Sliding*: Simulates a working set that slides through a large array. At every step, it randomly accesses a certain region of the array, and then moves to the next consecutive region. Each of these workloads has 1:1 read to write ratios.
(2) *Storage-oriented in-memory workloads*. Persistent memory can be used to optimize disk-based storage applications by placing storage data in memory. In order to evaluate such applications, we construct two benchmarks with key-value stores that represent typical in-memory storage applications (similar to prior work [13, 83]). These benchmarks perform search, insert, and delete operations on respectively hash tables and red-black trees.
(3) *SPEC CPU2006*. As ThyNVM is designed for hosting legacy applications of various types, we also evaluate SPEC benchmarks. A process with such a workload can benefit from ThyNVM by safely assuming a persistent and crash-consistent address space. We select the eight most memory-intensive applications from the SPEC CPU2006 suite and run each for one billion instructions. For the remaining SPEC CPU2006 applications, we verified that ThyNVM has negligible effect compared to the Ideal DRAM.

5.2 Micro-Benchmarks

Overall Performance. Figure 7 shows the execution time of our evaluated systems compared to the Ideal DRAM/NVM systems. We make three observations from this figure. (1) ThyNVM consistently performs better than other consistency mechanisms for all access patterns. It outperforms journaling and shadow paging by 10.2% and 14.8% on average. (2) Unlike prior consistency schemes, ThyNVM can adapt to different access patterns. For example, shadow paging performs poorly with the random access pattern, because even if only few blocks of a page are dirty in DRAM, it checkpoints the entire page in NVM. On the other hand, shadow paging performs better than journaling in the other two access patterns, as a larger number of sequential writes in these workloads get absorbed in DRAM and the checkpointing cost is reduced. ThyNVM can adapt to different access patterns and does not show any pathological behavior, outperforming both journaling and shadow paging on all workloads. (3) ThyNVM performs within 14.3% of the Ideal DRAM and 5.9% better than the Ideal NVM, on average.

We draw two major conclusions: (1) ThyNVM can flexibly adapt to different access patterns, and outperforms journaling and shadow paging in every micro-benchmark; (2) ThyNVM introduces an acceptable overhead to achieve its consistency guarantee, performing close to a system that provides crash consistency without any overhead.

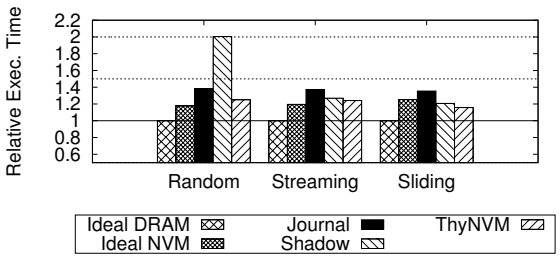


Figure 7: Execution time of micro-benchmarks.

NVM Write Traffic. Figure 8 shows the total amount of NVM write traffic across different workloads, for three components: (1) Last-level cache writeback to NVM, which is the direct NVM write traffic from the CPU; (2) NVM writes due to checkpointing; and (3) NVM writes due to page migration (described in §3.4). We make three observations. First, journaling and shadow paging exhibit large amounts of write traffic in at least one workload because they cannot adapt to access patterns (e.g., *Random* for shadow paging). In contrast, ThyNVM does not lead to an extremely large amount of NVM writes in any workload. On average, ThyNVM reduces the NVM write traffic by 10.8%/14.4% com-

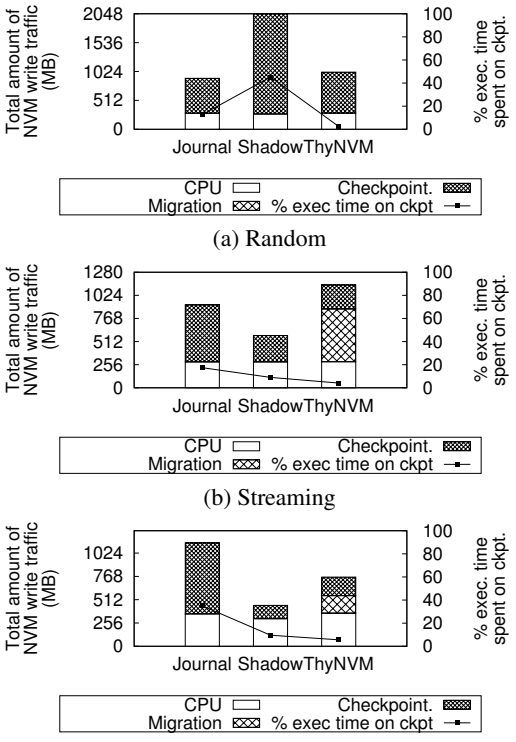


Figure 8: NVM write traffic and checkpointing delay (% of execution time spent on checkpointing) of micro-benchmarks on different memory systems. "CPU" represents writes from CPU to NVM, and "Checkpoint."/"Migration" represents NVM writes due to checkpointing/migration.

pared to journaling and shadow paging, respectively. Second, although ThyNVM reduces NVM write traffic across all access patterns *on average*, it incurs more NVM write traffic than the consistency mechanism that has the lowest NVM write traffic for *each* individual access pattern. The reason is that, as it overlaps checkpointing with execution, ThyNVM stores more versions and therefore can checkpoint more data than the traditional checkpointing mechanisms. Third, we observe that based on the access pattern, ThyNVM can result in different amounts of NVM write traffic due to page migration. For example, *Streaming* results in a large amount of migration traffic, as pages are moved in and out of DRAM without coalescing any write. In contrast, in the *Sliding* workload, pages that are migrated to DRAM can coalesce substantial amounts of writes before they are migrated back to NVM as the working set gradually moves.

We draw two conclusions: (1) Adaptivity of ThyNVM to different access patterns reduces overall NVM write traffic across various patterns; (2) By overlapping execution with checkpointing, ThyNVM reduces execution time, but introduces more NVM write traffic than the best performing traditional checkpointing system for each individual workload.

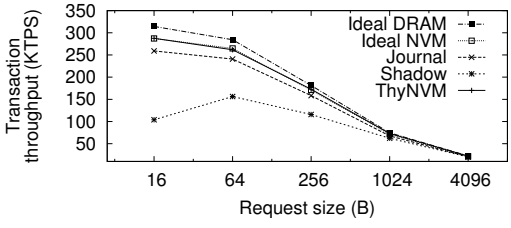
Checkpointing Delay. Figure 8 also includes the percentage of time each workload spends on checkpointing. Journaling/shadow paging spend 18.9%/15.2% time on checkpointing, while ThyNVM reduces this overhead to 2.5% on average. We conclude that ThyNVM can effectively avoid stalling by overlapping checkpointing with execution.

5.3 Storage Benchmarks

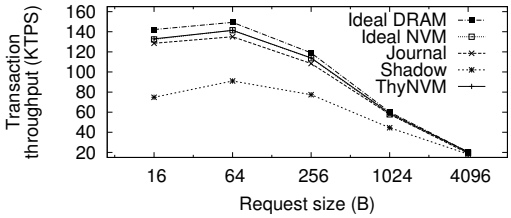
The results with our storage benchmarks represent the performance of ThyNVM under realistic workloads.

Throughput Performance. Figure 9 shows the transaction throughput of our in-memory storage workloads, where we vary the size of requests sent to the two key-value stores from 16B to 4KB. We make two observations from this figure. (1) ThyNVM consistently provides better transaction throughput than the traditional consistency mechanisms. Overall, averaged across all request sizes, ThyNVM provides 8.8%/4.3% higher throughput than journaling and 29.9%/43.1% higher than shadow paging with the hash table/red-black tree data structures. (2) ThyNVM's transaction throughput is close to that of the ideal DRAM-based and NVM-based systems. ThyNVM achieves 95.1%/96.2% throughput of Ideal DRAM with the hash table/red-black tree workloads, respectively. We conclude that ThyNVM outperforms traditional consistency mechanisms, and incurs little throughput reduction compared to the Ideal DRAM/NVM systems, for realistic in-memory storage workloads.

Memory Write Bandwidth. Figure 10 shows the NVM write bandwidth consumption of our storage workloads across different request sizes. We make two observations. (1) ThyNVM uses less NVM write bandwidth than shadow paging in most cases. These workloads exhibit relatively random write behavior, so shadow paging causes high bandwidth consumption during checkpointing because it frequently copies entire pages with small number of dirty blocks. In contrast, ThyNVM reduces NVM write bandwidth consumption by 43.4%/64.2% compared to shadow paging with hash table/red-black tree by adapting its checkpointing granularity to access locality. (2) ThyNVM introduces more NVM writes than journaling under these workloads (journaling has 19.0%/14.0% less NVM writes), because ThyNVM has to maintain more versions to overlap checkpointing and execution. This observation confirms a tradeoff between memory write bandwidth and performance: ThyNVM consumes more NVM write bandwidth to maintain multiple versions but improves performance by reducing stall time due to checkpointing. Overall, we can see that the NVM write bandwidth consumption of ThyNVM approaches that of journaling and is much smaller than that of shadow paging.

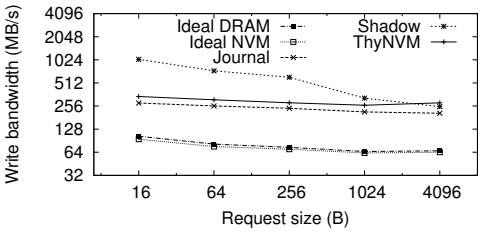


(a) Hash table based key-value store

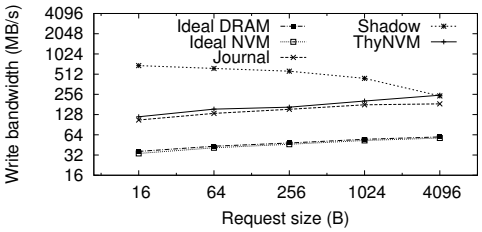


(b) Red-black tree based key-value store

Figure 9: Transaction throughput for two key-value stores: (a) hash table based, (b) red-black tree based.



(a) Hash table based key-value store



(b) Red-black tree based key-value store

Figure 10: Write bandwidth consumption of the key-value stores based on the hash table and the red-black tree, respectively. "Write bandwidth" refers to DRAM writes in the Ideal DRAM, and NVM writes for the rest.

5.4 Compute-Bound Benchmarks

Figure 11 shows the IPC (Instructions Per Cycle) performance of memory-intensive SPEC CPU2006 workloads normalized to the ideal DRAM-based system. We make two

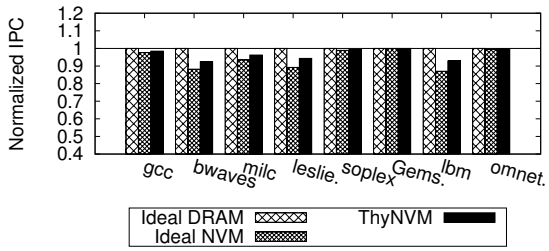


Figure 11: Performance of SPEC CPU2006 benchmarks (normalized to Ideal DRAM).

observations. (1) ThyNVM slows down these benchmarks on average by only 3.4% compared to the ideal DRAM-based system. (2) ThyNVM speeds up these benchmarks on average by 2.7% compared to the ideal NVM-based system, thanks to the presence of DRAM. We conclude that ThyNVM significantly reduces the overhead of checkpointing and provides almost on par performance with idealized systems that provide crash consistency at no cost.

5.5 Sensitivity Analysis

Figure 12 depicts the sensitivity of ThyNVM to the number of BTT entries while running in-memory storage workloads using hash tables. We draw two conclusions. (1) The NVM write traffic reduces with a larger BTT, which reduces the number of checkpoints and in turn writes to NVM. (2) Transaction throughput generally increases with the BTT size. This is mainly because a larger BTT reduces the write bandwidth into NVM, alleviating memory bus contention.

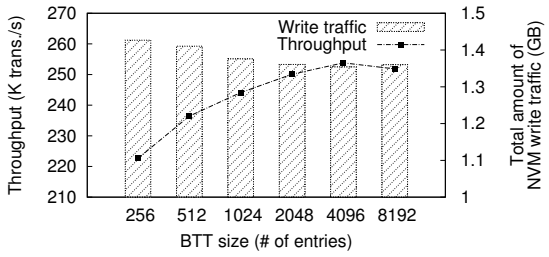


Figure 12: Effect of BTT size on storage benchmarks.

6. DISCUSSION

Comparison with Whole-System Persistence (WSP) [52]. Our model and goal are different from the Whole-System Persistence model and goal. WSP aims to support recovery of the whole system state, *including device states*, whereas our goal in ThyNVM is to support crash consistency of only *memory data*. ThyNVM does not deal with device states, leaving the recovery of any external device state to the system software (as discussed below). WSP assumes a pure NVM configuration so that it needs to flush only CPU states upon power failures and does not deal with volatile DRAM state. ThyNVM assumes a DRAM+NVM hybrid memory that can offer better performance than NVM alone [62, 81], but requires periodic checkpointing of both CPU states and memory data to provide crash consistency. Unlike WSP, ThyNVM provides crash consistency of memory data *without requiring or assuming* purely nonvolatile memory.

Loss of external states on system failures. ThyNVM does not protect external states (e.g., in the network card). These states are lost after system reboots, leading to device errors during recovery. As such, system failure events are *visible* to programs in the form of device errors. Device errors are common in computer systems [24, 20]. Commodity systems already incorporate mechanisms to handle such errors [9, 67, 53]. This failure model we assume is inline with many previous works, e.g., [73, 61, 25, 21].

Software bug tolerance. Software bugs can lead to system crashes by corrupting memory states. Yet, crash consistency and bug tolerance are two different problems. Any system that maintains crash consistency, including databases and file systems that do so, is susceptible to software bugs; ThyNVM is not different. For example, bugs in commonly-used software, e.g., Eclipse (a widely used development suite) and Chromium (a popular browser), have caused data corruption in *databases* and *file systems* [59, 5]. ThyNVM provides crash consistency, and it cannot (and is not designed to) prevent such software bugs. However, it can be extended to help enhance bug tolerance, e.g., by copying checkpoints to secondary storage periodically and devising mechanisms to find and recover to past bug-free checkpoints. Studies of such mechanisms can open up attractive new research directions for future work building upon ThyNVM.

Explicit interface for persistence. Our checkpointing mechanism can be seamlessly integrated in a system with a configurable persistence guarantee [60, 12, 46]. Such a system is only allowed to lose data updates that happened in the last n ms, where n is configurable. ThyNVM can be configured to checkpoint data every n ms and roll back to the last consistent checkpoint after a crash. Persistence of data can also be explicitly triggered by the program via a new instruction added to the ISA that forces ThyNVM to end an epoch.

7. RELATED WORK

To our knowledge, ThyNVM is the first persistent memory design that can execute unmodified, legacy applications transparently *without requiring* (1) programmer effort to modify software or (2) special power management. Most other persistent memory designs [77, 13, 76, 29, 72, 83, 49, 32, 42, 57] require programmers to *rewrite* their code following new APIs developed for manipulating data updates in persistent memory.

We have already compared ThyNVM in detail to major software mechanisms to guarantee crash consistency, logging [77, 13] and copy-on-write (CoW) [14, 76], and shown that ThyNVM outperforms both approaches. We now briefly discuss other related works.

APIs in software-based persistent memory design. Persistent memory has been extensively studied in the software community [77, 13, 76, 29, 72, 71, 11, 4]. Most previous works developed application-level or system-level programming interfaces to provide crash consistency. For example, Mnemosyne [77] and NV-heaps [13] adopt application-level programming interfaces for managing persistent regions; programmers need to explicitly declare, allocate, and deallocate persistent objects. Consistent and Durable Data Structures [76] require application developers to modify existing data structures to achieve high-performance persistent data updates. EROS [71] and Rio file cache [11] require system developers to reimple-

ment operating system components to perform efficient management of persistence. The APIs introduced by Intel [29] and SNIA [72] require programmers to explicitly declare persistent objects [77, 13], reimplement in-memory data structures [76], or modify legacy programs to use transactional interfaces [77, 13].

APIs in hardware-based persistent memory design. Hardware support for persistent memory is receiving increasing attention [45, 14, 83, 49, 58, 32, 42, 57, 43, 84]. Most of these studies employ similar APIs in software-based persistent memory systems. For example, Kiln [83] adopts a transaction-based API to obtain hints on critical data updates from software. Moraru *et al.* [49] propose a design that requires programmers to explicitly allocate persistent objects. The relaxed persistence model proposed by Pelley *et al.* [57] employs a new API to support their special persistence model. BPFS [14] uses the persistent memory through a file system, incurring significant software overhead.

ThyNVM does not require any new APIs at the application or system level. Hence, it offers an attractive way for users to take advantage of persistent memory without changing programs.

Crash consistency with special power management. Whole-System Persistence [52] can provide crash consistency without programmer effort, but relies on the residual energy provided by the system power supply to flush *CPU caches and CPU state* on system/power failure. This model is limited by the data size that can be flushed with the residual energy and thus inapplicable for hybrid main memory that contains volatile DRAM. Auto-commit memory [19] uses a secondary power supply to commit volatile data into flash memory on system failure. Such extra power supplies increase the total cost of ownership of the system and add potential sources of unreliability.

Checkpointing and granularity choice. Checkpointing is a widely used technique in different domains [78, 48]. For example, ReVive [61], SafetyNet [73], Mona [22], and the checkpointing schemes for COMA [23] checkpoint memory data at the fine granularity of cache lines in order to improve availability of data for updates; while other works [30, 54, 63] use coarse granularity. In contrast, our mechanism employs two different granularities to form one checkpoint to get the best of multiple granularities (as explained in §2.3). To our knowledge, this is the first work that adapts checkpointing granularity to access pattern locality (and thereby achieves high performance at low overhead). Note that the flash translation layer can leverage two granularities [31, 40] but does not provide crash consistency as ThyNVM, and it is specific to flash memory.

8. CONCLUSION

We introduce a software-transparent mechanism that provides crash consistency of memory data for persistent and hybrid memory systems. Our design, ThyNVM, is based on automatic periodic checkpointing of the main memory state in a manner that minimally impacts application execution time. To this end, we introduce a new dual-scheme checkpointing mechanism that efficiently overlaps checkpointing and application execution by (1) adapting the granularity of checkpointing to the spatial locality of memory updates and (2) providing mechanisms to coordinate memory updates with two different granularities of checkpointing. Our evaluations using multiple different access patterns, realistic storage workloads, and compute-bound workloads show that ThyNVM provides crash consistency of memory data at low overhead, with performance close to an ideal system that provides crash consistency at no performance overhead. We believe ThyNVM's efficient support for software-transparent crash consistency can enable (1) easier and more widespread adoption of persistent memory, and (2) more efficient software stack support for exploiting persistent memory. We hope that our work encourages more research in providing automatic and programmer-friendly mechanisms for managing persistent and hybrid memories. To foster such research, we have open sourced ThyNVM at `http://persper.com/thynvm`.

ACKNOWLEDGMENTS

We thank the anonymous reviewers, our shepherd, and the SAFARI group members for their valuable feedback. This work was partially supported by National High-Tech R&D (863) Program of China (2013AA01A213), National Basic Research (973) Program of China (2011CB302505), National Natural Science Foundation of China (61433008, 61373145, 61170210, U1435216), US National Science Foundation grants 0953246, 1065112, 1212962, 1320531, Semiconductor Research Corporation, and the Intel Science and Technology Center for Cloud Computing. We thank our industrial partners, including Google and Samsung, for the support they have provided. Jinglei Ren and Yongwei Wu are affiliated with Dept. of Computer Science and Technology, Tsinghua National Lab for Information Science and Technology, Beijing, and Research Institute of Tsinghua University, Shenzhen. A majority of this work was carried out while Jinglei Ren was a visiting scholar, Samira Khan a postdoctoral researcher, and Jongmoo Choi a visiting professor in Onur Mutlu's SAFARI Research Group.

REFERENCES

[1] J. Ahn *et al.*, "A scalable processing-in-memory accelerator for parallel graph processing," in *ISCA*, 2015.

[2] H. Akinaga *et al.*, "Resistive random access memory (ReRAM) based on metal oxides," *Proc. IEEE*, vol. 98, no. 12, 2010.

[3] R. H. Arpaci-Dusseau *et al.*, "Crash consistency: FSCK and journaling," 2015. [Online] http://pages.cs.wisc.edu/~remzi/OSTEP/

[4] J. Arulraj *et al.*, "Let's talk about storage & recovery methods for non-volatile memory database systems," in *SIGMOD*, 2015.

[5] Bean...@gmail.com, "Unrecoverable chrome.storage.sync database corruption," 2014. [Online] https://code.google.com/p/chromium/issues/detail?id=261623

[6] P. A. Bernstein *et al.*, *Principles of Transaction Processing*. Morgan Kaufmann, 2009.

[7] N. Binkert *et al.*, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, Aug. 2011.

[8] I. Burcea *et al.*, "Predictor virtualization," in *ASPLOS*, 2008.

[9] C++ Tutorials, "Exceptions," 2015. [Online] http://www.cplusplus.com/doc/tutorial/exceptions/

[10] C. Cascaval *et al.*, "Software transactional memory: Why is it only a research toy?" *ACM Queue*, vol. 6, no. 5, Sep. 2008.

[11] P. M. Chen *et al.*, "The Rio file cache: Surviving operating system crashes," in *ASPLOS*, 1996.

[12] J. Cipar *et al.*, "LazyBase: Trading freshness for performance in a scalable database," in *EuroSys*, 2012.

[13] J. Coburn *et al.*, "NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," in *ASPLOS*, 2011.

[14] J. Condit *et al.*, "Better I/O through byte-addressable, persistent memory," in *SOSP*, 2009.

[15] G. Copeland *et al.*, "The case for safe RAM," in *VLDB*, 1989.

[16] D. J. DeWitt *et al.*, "Implementation techniques for main memory database systems," in *SIGMOD*, 1984.

[17] D. Dice *et al.*, "Early experience with a commercial hardware transactional memory implementation," in *ASPLOS*, 2009.

[18] P. Felber *et al.*, "Dynamic performance tuning of word-based software transactional memory," in *PPoPP*, 2008.

[19] D. Flynn *et al.*, "Apparatus, system, and method for auto-commit memory," 2012, US Patent App. 13/324,942.

[20] D. Ford *et al.*, "Availability in globally distributed storage systems," in *OSDI*, 2010.

[21] C. Fu *et al.*, "Exception-chain analysis: Revealing exception handling architecture in Java server applications," in *ICSE*, 2007.

[22] S. Gao *et al.*, "Real-time in-memory checkpointing for future hybrid memory systems," in *ICS*, 2015.

[23] A. Gefflaut *et al.*, "COMA: An opportunity for building fault-tolerant scalable shared memory multiprocessors," in *ISCA*, May 1996.

[24] S. Ghemawat *et al.*, "The Google file system," in *SOSP*, 2003.

[25] J. B. Goodenough, "Exception handling: Issues and a proposed notation," *Commun. ACM*, vol. 18, no. 12, Dec. 1975.

[26] A. Hansson *et al.*, "Simulating DRAM controllers for future system architecture exploration," in *ISPASS*, 2014.

[27] HP Labs, "The Machine: A new kind of computer," 2015. [Online] http://www.hpl.hp.com/research/systems-research/themachine/

[28] Intel, "Intel architecture instruction set extensions programming reference," 2015. [Online] https://software.intel.com/sites/default/files/managed/07/b7/319433-023.pdf

[29] Intel, "The NVM library," 2015. [Online] http://pmem.io/

[30] T. Islam *et al.*, "MCREngine: A scalable checkpointing system using data-aware aggregation and compression," in *SC*, 2012.

[31] J.-U. Kang *et al.*, "A superblock-based flash translation layer for NAND flash memory," in *EMSOFT*, 2006.

[32] S. Kannan *et al.*, "Reducing the cost of persistence for nonvolatile heaps in end user devices," in *HPCA*, 2014.

[33] T. Kawahara *et al.*, "2 Mb SPRAM (SPin-Transfer Torque RAM) with bit-by-bit bi-directional current write and parallelizing-direction current read," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 1, 2008.

[34] H. Kim *et al.*, "Bounding memory interference delay in COTS-based multi-core systems," in *RTAS*, 2014.

[35] Y. Kim *et al.*, "Thread cluster memory scheduling: Exploiting differences in memory access behavior," in *MICRO*, 2010.

[36] E. Kultursay *et al.*, "Evaluating STT-RAM as an energy-efficient main memory alternative," in *ISPASS*, 2013.

[37] C. Lamb *et al.*, "The ObjectStore database system," *Commun. ACM*, vol. 34, no. 10, Oct. 1991.

[38] B. C. Lee *et al.*, "Architecting phase change memory as a scalable DRAM alternative," in *ISCA*, 2009.

[39] B. C. Lee *et al.*, "Phase change technology and the future of main memory," *IEEE Micro*, 2010.

[40] S.-W. Lee *et al.*, "A log buffer-based flash translation layer using fully-associative sector translation," *ACM Trans. Embed. Comput. Syst.*, vol. 6, no. 3, Jul. 2007.

[41] Linux Community, "Ext4 (and ext3) filesystem wiki," 2015. [Online] https://ext4.wiki.kernel.org

[42] R.-S. Liu *et al.*, "NVM Duet: Unified working memory and persistent store architecture," in *ASPLOS*, 2014.

[43] Y. Lu *et al.*, "Loose-ordering consistency for persistent memory," in *ICCD*, 2014.

[44] J. Meza *et al.*, "Enabling efficient and scalable hybrid memories using fine-granularity DRAM cache management," *Computer Architecture Letters*, 2012.

[45] J. Meza *et al.*, "A case for efficient hardware/software cooperative management of storage and memory," in *WEED*, 2013.

[46] J. Mickens *et al.*, "Blizzard: Fast, cloud-scale block storage for cloud-oblivious applications," in *NSDI*, 2014.

[47] C. C. Minh *et al.*, "STAMP: Stanford transactional applications for multi-processing," in *IISWC*, 2008.

[48] A. Moody *et al.*, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *SC*, 2010.

[49] I. Moraru *et al.*, "Consistent, durable, and safe memory management for byte-addressable non-volatile main memory," in *TRIOS*, 2013.

[50] O. Mutlu *et al.*, "Stall-time fair memory access scheduling for chip multiprocessors," in *MICRO*, 2007.

[51] O. Mutlu *et al.*, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems," in *ISCA*, 2008.

[52] D. Narayanan *et al.*, "Whole-system persistence," in *ASPLOS*, 2012.

[53] T. Ogasawara *et al.*, "EDO: Exception-directed optimization in Java," *ACM Trans. Program. Lang. Syst.*, vol. 28, no. 1, Jan. 2006.

[54] A. J. Oliner *et al.*, "Cooperative checkpointing: A robust approach to large-scale systems reliability," in *ICS*, 2006.

[55] V. Pankratius *et al.*, "A study of transactional memory vs. locks in practice," in *SPAA*, 2011.

[56] J. T. Pawlowski, "Hybrid memory cube (HMC)," in *Hot Chips*, 2011.

[57] S. Pelley *et al.*, "Memory persistency," in *ISCA*, 2014.

[58] S. Pelley *et al.*, "Storage management in the NVRAM era," in *VLDB*, 2013.

[59] R. Platt, "Undo/redo and save actions corrupting files," 2015. [Online] https://bugs.eclipse.org/bugs/show_bug.cgi?id=443427

[60] D. R. K. Ports *et al.*, "Transactional consistency and automatic management in an application data cache," in *OSDI*, 2010.

[61] M. Prvulovic *et al.*, "ReVive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors," in *ISCA*, 2002.

[62] M. K. Qureshi *et al.*, "Scalable high performance main memory system using phase-change memory technology," in *ISCA*, 2009.

[63] R. Rajachandrasekar *et al.*, "MIC-Check: A distributed check pointing framework for the Intel many integrated cores architecture," in *HPDC*, 2014.

[64] S. Raoux *et al.*, "Phase-change random access memory: A scalable technology," *IBM Journal of Research and Development*, vol. 52, no. 4, 2008.

[65] J. Ren, "State machine protocol of the ThyNVM checkpointing schemes," 2015. [Online] http://persper.com/thynvm/state-machine.pdf

[66] J. Ren, "Verification of the ThyNVM checkpointing schemes," 2015. [Online] http://persper.com/thynvm/verification.pdf

[67] M. P. Robillard *et al.*, "Analyzing exception flow in Java programs," in *ESEC/FSE*, 1999.

[68] V. Seshadri *et al.*, "The dirty-block index," in *ISCA*, 2014.

[69] V. Seshadri *et al.*, "RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization," in *MICRO*, 2013.

[70] V. Seshadri *et al.*, "Page overlays: An enhanced virtual memory framework to enable fine-grained memory management," in *ISCA*, 2015.

[71] J. S. Shapiro *et al.*, "EROS: A fast capability system," in *SOSP*, 1999.

[72] SNIA, "NVM programming model (NPM) version 1.1," 2015. [Online] http://www.snia.org/tech_activities/standards/curr_standards/npm

[73] D. Sorin *et al.*, "SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery," in *ISCA*, 2002.

[74] L. Subramanian *et al.*, "MISE: Providing performance predictability and improving fairness in shared main memory systems," in *HPCA*, 2013.

[75] L. Subramanian *et al.*, "The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory," in *MICRO*, 2015.

[76] S. Venkataraman *et al.*, "Consistent and durable data structures for non-volatile byteaddressable memory," in *FAST*, 2011.

[77] H. Volos *et al.*, "Mnemosyne: lightweight persistent memory," in *ASPLOS*, 2011.

[78] Y.-M. Wang *et al.*, "Checkpointing and its applications," in *FTCS*, 1995.

[79] G. Wiki, "Transactional memory in GCC," 2015. [Online] https://gcc.gnu.org/wiki/TransactionalMemory

[80] D. H. Yoon *et al.*, "Virtualized and flexible ECC for main memory," in *ASPLOS*, 2010.

[81] H. Yoon *et al.*, "Row buffer locality aware caching policies for hybrid memories," in *ICCD*, 2012.

[82] H. Yoon *et al.*, "Efficient data mapping and buffering techniques for multi-level cell phase-change memories," *TACO*, 2014.

[83] J. Zhao *et al.*, "Kiln: Closing the performance gap between systems with and without persistence support," in *MICRO*, 2013.

[84] J. Zhao *et al.*, "FIRM: Fair and high-performance memory control for persistent memory systems," in *MICRO*, 2014.