Dual-Page Checkpointing: An Architectural Approach to Efficient Data Persistence for In-Memory Applications

SONG WU, FANG ZHOU, XIANG GAO, and HAI JIN, Huazhong University of Science and Technology, China
JINGLEI REN, Microsoft Research, China

Data persistence is necessary for many in-memory applications. However, the disk-based data persistence largely slows down in-memory applications. Emerging non-volatile memory (NVM) offers an opportunity to achieve in-memory data persistence at the DRAM-level performance. Nevertheless, NVM typically requires a software library to operate NVM data, which brings significant overhead.

This article demonstrates that a hardware-based high-frequency checkpointing mechanism can be used to achieve efficient in-memory data persistence on NVM. To maintain checkpoint consistency, traditional logging and copy-on-write techniques incur excessive NVM writes that impair both performance and endurance of NVM; recent work attempts to solve the issue but requires a large amount of metadata in the memory controller. Hence, we design a new *dual-page checkpointing* system, which achieves low metadata cost and eliminates most excessive NVM writes at the same time. It breaks the traditional trade-off between metadata space cost and extra data writes. Our solution outperforms the state-of-the-art NVM software libraries by 13.6× in throughput, and leads to 34% less NVM wear-out and 1.28× higher throughput than state-of-the-art hardware checkpointing solutions, according to our evaluation with OLTP, graph computing, and machine-learning workloads.

CCS Concepts: • **Computer systems organization → Processors and memory architectures**; • **Hardware** → *Memory and dense storage*;

Additional Key Words and Phrases: Data persistence, non-volatile memory (NVM), checkpointing, crash consistency

ACM Reference format:
Song Wu, Fang Zhou, Xiang Gao, Hai Jin, and Jinglei Ren. 2019. Dual-Page Checkpointing: An Architectural Approach to Efficient Data Persistence for In-Memory Applications. *ACM Trans. Archit. Code Optim.* 15, 4, Article 57 (January 2019), 27 pages.
https://doi.org/10.1145/3291057

1 INTRODUCTION

Data persistence makes a critical performance bottleneck of many in-memory applications [5, 74]. Particularly, latest analytics applications [15, 64] show the rise of stream processing [35], graph

This work is supported by National Key Research and Development Program under Grant No. 2018YFB1003600, National Science Foundation of China under Grants No. 61872155 and No. 61472151, and Pre-research Project of Beifang under Grant No. FFZ-1601.

Authors' addresses: S. Wu, F. Zhou, X. Gao, and H. Jin, Huazhong University of Science and Technology, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, Wuhan, 430000, China; emails: {wusong, flame, xgao, hjin}@hust.edu.cn; J. Ren, Microsoft Research, Beijing, 100000, China; email: jinren@microsoft.com.

1544-3566/2019/01-ART57
https://doi.org/10.1145/3291057

computing [27], and machine learning [33]. Persistence of in-memory data provides an important form of fault tolerance to such applications. For continuous queries or endless jobs that process data flowing from/to external sources [34, 47, 73], the computing states cannot be lost, because there is no easy way to rebuild them. The rebuild process is either time consuming (e.g., training a model) or lossy as the original data is no longer available (e.g., sensor data in an Internet of Things setting). Thus, efficient in-memory data persistence is highly demanded, especially for modern systems whose mean time between failures (MTBF) is relatively low [16, 18, 50, 61].

Underlying in-memory storage, such as Alluxio [3], Silo [66], and RAMCloud [45], invest huge efforts in overcoming the bottleneck. On the one hand, sequential data persistence and parallel I/O are exploited to increase persistence efficiency [44, 77]. On the other hand, special data abstractions are designed [32, 74] to minimize the amount of necessary data to persist. However, those solutions still largely limit the performance of in-memory applications or sacrifice the flexibility of programming.

Emerging byte-accessible non-volatile memory (NVM) promises an opportunity to achieve efficient in-memory data persistence, overcoming the traditional bottleneck. NVM technologies, such as 3D XPoint [22], PCM [29, 54], STT-RAM [4, 28], and ReRAM [2], enable performant, direct access to persistent data via load/store processor instructions. The access latency and bandwidth of NVM attached to the memory bus are orders of magnitude better than disks [30, 52]. Consequently, the overhead due to *software* manifests. To avoid such software overhead, we typically bypass traditional *heavyweight* filesystems or databases when applying NVM to data persistence [7, 12, 38, 75]. However, doing so exposes the *crash consistency* issue to applications so that they still have to employ a software transaction library to operate NVM. For example, Mnemosyne [70] and NV-heaps [9] are early efforts, and many others [19, 21, 26] continue to improve the system to fully utilize the advantages of NVM. Recently, DudeTM [36] introduces a shadow DRAM for transaction processing, employs a redo log to maintain crash consistency and performs group commit to raise the transaction throughput. Nevertheless, those sophisticated software components still incur significant overhead in data persistence.

This article explores an *architectural* way to realize in-memory data persistence and reduce the overall system complexity and overhead. It turns out that a high-frequency whole-memory *checkpointing* technique can serve the purpose. We constantly keep a consistent checkpoint of the whole memory in NVM and update it with a short checkpointing interval, which ensures that applications can recover to the latest persistent state quickly after a crash. Such an approach arises as we observe a *convergence* of the software storage library behavior and the recent development of hardware checkpointing techniques. On the one hand, a software transactional storage has the advantage of being able to synchronously persist a transaction, but, to fulfill the extremely high throughput demand of applications, group commit or its variant is employed (e.g., SiloR [77] and DudeTM [36]). On the other hand, unlike traditional software-based memory checkpointing solutions [1, 13], recent hardware checkpointing solutions support a millisecond-level checkpointing interval [56]. That means they can support comparable persistence latency to a software transactional storage optimized for throughput. Such an observation gives rise to our architectural approach to data persistence, which has several advantages. First, it requires minimal work in software by utilizing the natural address translation capability of the memory controller. Second, it automatically incorporates the group commit optimization by dividing execution time into epochs and generating a checkpoint at the end of each epoch. As a result, our solution simplifies the overall system design across software and hardware, and achieves at least *one order of magnitude* more performant data persistence than state-of-the-art *software* solutions on NVM [21, 36].

To apply a checkpointing technique to NVM, a key *challenge* is how to efficiently guarantee crash consistency of each checkpoint. Traditional hardware redo/undo logging or copy-on-write

Table 1. Tradeoff between the Amount of NVM Writes (Data) and the Space Cost of Address Mappings (Metadata)

		Metadata cost in the memory controller	
		large ✗	**small ✓**
Extra data write	**large ✗**	(undesirable)	redo/undo logging, CoW, etc.
	small ✓	ThyNVM [56], page overlays [58], etc.	*dual-page checkpointing*

(CoW) can guarantee such consistency but requires at least 2× NVM writes than the amount of actual dirty data. The extra NVM writes are strongly undesirable, because NVM writes are slow (at least one order of magnitude slower than DRAM) and the endurance of NVM is limited (10^9 P/E cycles compared to DRAM's 10^{15} cycles) [30]. To reduce the extra NVM writes, a viable way is to use *fine-grained* CoW to protect checkpoints, which avoids copying most clean data. While it reduces the amount of *data* writes, a small granularity increases the *metadata* to maintain in the memory controller. The metadata consists of address mappings required by CoW. The space overhead of address mappings at the cache-line-level granularity is usually too large for scarce hardware space. Although recent work such as ThyNVM [56] and page overlays [58] constructs sophisticated solutions to overcome the issue, such metadata and its management are still a bottleneck.

We design a new checkpointing scheme to address the challenge. It achieves *both* minimal extra NVM writes *and* minimal metadata space/management overhead. In this design, we maintain in the memory controller the coarse-grained address mappings for dirty *pages*. In these mappings, we compactly embed fine-grained states that track actual dirty *cache lines*. Particularly, each logical dirty page is mapped to two physical NVM pages to separately store both the checkpoint and working data. And, a bit vector is employed to denote dirty cache lines and their locations in either physical page. Therefore, we name our approach *dual-page* checkpointing. By maintaining two versions of the data, we guarantee that the checkpoint version keeps consistent while the working version is being written. We recycle the pages of checkpoint version and working version to reduce space overhead, and the final NVM space wastage is only 3.7% of total NVM space. Our unique *insight* here is that we can take advantage of the page-level information to efficiently perform cache-line level management. Compared to ThyNVM [56] and page overlays [58], we realize 5× to 20× lower hardware overhead of metadata.

In summary, our solution breaks the traditional trade-off between extra writes in NVM and metadata overhead in the memory controller, as shown in Table 1. Traditional redo/undo logging and CoW require little metadata in the memory controller but write excessive NVM data (more details in Section 2). Lately, ThyNVM [56] and page overlays [58] reduce extra NVM writes but increase hardware cost for metadata. In contrast, our dual-page checkpointing solution obtains the best in both metrics.

We apply dual-page checkpointing to a DRAM + NVM hybrid persistent memory system to support data persistence for in-memory computing applications. Considering that NVM has larger access latency and much lower endurance than DRAM, a small DRAM cache layer is set atop NVM [25, 52, 56]. Our solution performs system-level checkpointing, which is transparent to applications—programmers need not write code for generating and restoring checkpoints. This feature largely reduces programmers' burden. The memory controller is configured with a fixed epoch length (a.k.a., checkpointing interval).[1] Applications can acknowledge durability as long as one epoch has passed, following the external synchrony model [43]. The acknowledgement

[1]An alternative is to offer a checkpointing instruction to software.

latency can be confined as our solution offers high-frequency checkpointing, with millisecond-level epochs.

This article makes the following contributions:

- We explore an architectural approach to supporting in-memory data persistence on NVM. We demonstrate that a simple hardware-based checkpointing mechanism in the memory controller can efficiently achieve data durability and crash consistency of in-memory data and avoid the overhead of complex state-of-the-art software solutions.
- We design a new dual-page checkpointing scheme for the memory controller. It maintains address mappings at both page granularity and cache-line granularity in an *integrated* manner without incurring excessive space and management overhead. Our solution solves the traditional dilemma in the trade-off between metadata hardware overhead and extra NVM data writes.
- We implement dual-page checkpointing in a DRAM + NVM hybrid persistent memory simulator. Our evaluation with various OLTP, graph computing, and machine-learning workloads shows that our solution outperforms the state-of-the-art software solution by 13.6× higher throughput, and latest checkpointing techniques by 1.28×, on average.

The rest of the article is organized as follows. Section 2 reviews background and motivates our design rationale. Section 3 describes the design of our dual-page checkpointing system. Section 4 details key implementations and optimizations. Section 5 presents the experiment results. We discuss related work in Section 6 and conclude in Section 7.

2 BACKGROUND AND MOTIVATION

Consistency of checkpoints is a vital issue for a checkpointing mechanism. Particularly, it is required that a checkpoint is a snapshot of the memory image captured at a time point. The time interval between two consecutive checkpoints is referred to as an *epoch*. Next, we review existing approaches to protect checkpoint consistency and motivate our design.

2.1 Logging

Logging is a general way to guarantee consistency of data updates, and can be applied to hardware checkpointing. There are two types of logging, undo logging and redo logging. To protect the existing consistent checkpoint data, undo logging requires that the original data is copied to a separate, special log area before it can be updated to a new value. Should a system crash, a recovery process would scan the log area and restore the original data to ensure consistency of the checkpoint. To the contrary, redo logging requires that an update is written to the log area before modifying the original checkpoint data.

Such a logging mechanism suffers from the *write twice* issue—every memory write incurs another extra write to the separate log area, as described above. Each logging also needs to record the page information, which produces more writes. Those redundant writes occupy excessive NVM write bandwidth and consume limited NVM endurance [30]. To alleviate this issue, one optimization [61] is to track writes within one epoch using an address mapping structure. Multiple writes to the same address within one epoch can be coalesced instead of going through the full logging path. This optimization can reduce unnecessary NVM writes but entails large hardware space overhead for address mappings, because the mapping granularity is as small as the cache line size [56].

2.2 Copy-on-Write (CoW)

To solve the above address mapping overhead, we can enlarge the mapping granularity and perform CoW in the hardware to protect checkpoint consistency. Such a coarse granularity is typically

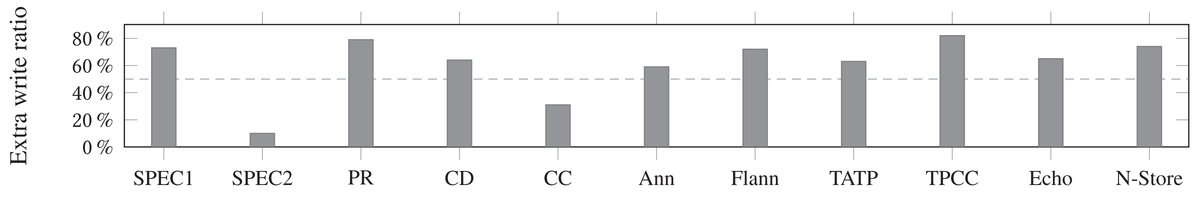

Fig. 1. The percentage of extra NVM writes by page-level address mapping. The number shows the ratio of clean cache lines in dirty pages.

set to the page size, e.g., 4KB. Within one epoch, when a page is updated for the first time, a shadow page is allocated and the original page is copied to the shadow page. Then, the update is only applied to the shadow page. Accordingly, a new address mapping is created for the page, so that following accesses within the same epoch are redirected to the shadow page. The original page, as part of the checkpoint data, is protected.

However, the problem with CoW is that it has to copy the whole page even if only a small part of it is actually dirty. We test various workloads with CoW. Figure 1 calculates the percentage of extra write (definitions of the workloads are in Section 5.1.3). The average extra write ratio is 61.1%, and 5 out of the 11 workloads have the ratio above 70%. Overall, CoW saves hardware space overhead for address mapping but brings an unacceptable amount of extra NVM writes.

2.3 Mixed-Granularity Mapping

Lately, it has been explored to manage address mappings at both fine-grained and coarse-grained granularities, to reduce extra NVM writes and take small hardware space for the mappings. Below are two representative solutions.

ThyNVM [56] incorporates two *independent* tables for cache-line level and page-level mappings, respectively. Choice of granularity is dynamically determined by the access pattern of individual pages. ThyNVM uses cache-line level mapping for pages with poor locality to reduce NVM writes, and page-level mapping for pages with high locality to decrease metadata overhead. However, one bottleneck of ThyNVM is the small number of cache-line level mappings that can be stored in the limited hardware space. The cause of the problem is that each cache-line mapping contains at least one *full* memory address to record which cache line is mapped. After optimization, it still requires at least 42 bits, assuming 48-bit address space in the current x86-64 architecture. In contrast, our work aims to minimize the bits to record a cache-line address mapping.

Page overlays [58] add to each page an overlay that stores modified cache lines of the page. This solution widens physical addresses, modifies TLB and incurs complex cache line management (e.g., each overlay is put into a fixed-length segment, and has to migrate to a larger one if the overlay outgrows the segment). Overall, the cache line mappings still require 352 bits per page in most cases. In contrast, we aim at a simple, scalable mechanism with minimal software/hardware modifications (more details in Section3.5).

3 DESIGN

This section describes our dual-page checkpointing mechanism for a persistent memory system. The mechanism constantly maintains a consistent checkpoint of the whole memory in NVM. Our design breaks the tradeoff in traditional hardware checkpointing systems and achieves both minimal extra NVM data writes and minimal hardware metadata overhead. Our key insight is that cache-line-level mappings can be compactly stored and efficiently managed by confining their scope to dirty pages and associating them to the page-level mappings.

3.1 Epoch and Consistency

Our solution follows an epoch model as most checkpointing schemes do. The whole system execution is divided into successive epochs [10, 56]. Each epoch consists of a fixed-length time period of normal execution, and a checkpointing period to stall the system and generate a checkpoint of the whole memory.[2] The checkpoint contains necessary CPU states for resuming the system execution upon recovery.

The *failure model* we adopt is identical to Whole-System Persistence [42] and ThyNVM [56]: when a crash or failure occurs, the system recovers by resuming execution from the checkpoint in NVM. If the system interacts with a client, then the software should acknowledge any operation (e.g., persistence of a transaction) *after* the operation is checkpointed. This behavior is similar to group commit and has been adopted by external synchrony [43] and DudeTM [36].

We store and only store *two* versions of data in NVM, the checkpoint version and the working version. The two versions of data only diverge when the data is updated within an epoch; otherwise, they share the same data in NVM. When the two versions diverge, we need to install an address mapping in the memory controller to record the addresses of both versions. By that mapping, subsequent memory accesses can find the right working version of the data.

3.2 Dual-Page Mapping

Our checkpointing mechanism is *efficient* in two senses: **(1)** A cache-line writeback to NVM from CPU requires minimal extra write, which has strong implications on both performance and endurance of NVM. **(2)** We minimize the hardware space overhead to track *cache-line* level mappings.

To realize such efficiency, the *key idea* is to maintain page-level mappings in a table of the memory controller and *embed* cache-line level mappings in per-page bit vectors. This idea leads to the design of dual-page mapping, where a logical page is mapped to two physical NVM pages. We refer to one page at its home location as the *base page*, and the other page, which is allocated in a specialized area, as the *derivative page*. The mapping information is stored in the memory controller so that any NVM access can look up the mapping and visit either the base page or the derivative page.

In addition to pages, we need cache-line level mappings to avoid the extra copy of clean data in CoW, and we have to store these mappings in a space-efficient form. Our solution is to avoid storing full cache-line address mappings. Instead, we reuse page-level mappings, and associate two *bit vectors* to each page mapping. The kth bit of a vector corresponds to the kth cache line of the page. The first bit vector records *checkpoint position*, i.e., whether each cache line has its checkpoint in the derivative page (set to 1) or in the base page (set to 0). The second bit vector records *dirtiness*, i.e., whether each cache line is dirty (set to 1) or not (set to 0) in the current epoch. In this way, we manage to use 2 bits to store all necessary mapping information for a cache line. As the page addresses are known, we only need to determine in which page a version of the cache line locates according to the two-bit state. Then, we can calculate the cache line address.

Figure 2 illustrates all four states of cache lines, numbered ❶–❹, and how they transfer between each other. A cache line has two versions in the base page and the derivative page, respectively.

Initially, if a cache line is not modified in an epoch, it acts as both the checkpoint and the working data (at this time, the derivative page may be not allocated yet). That is the state of ❶ on the left side, where the cache line locates in the base page, and it is not dirty yet, so the state in the two bit

[2]We currently assume the memory controller notifies the CPU to do so. It is also possible to offer a CPU instruction to OS to trigger it.

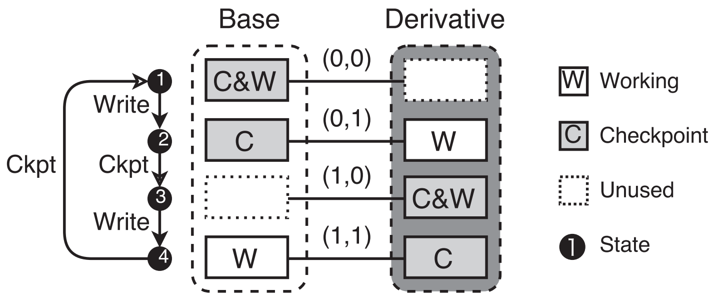

Fig. 2. Cache-line states in dual-page mapping. We use a `(checkpoint position, dirtiness)` tuple to denote the two bits of a state.

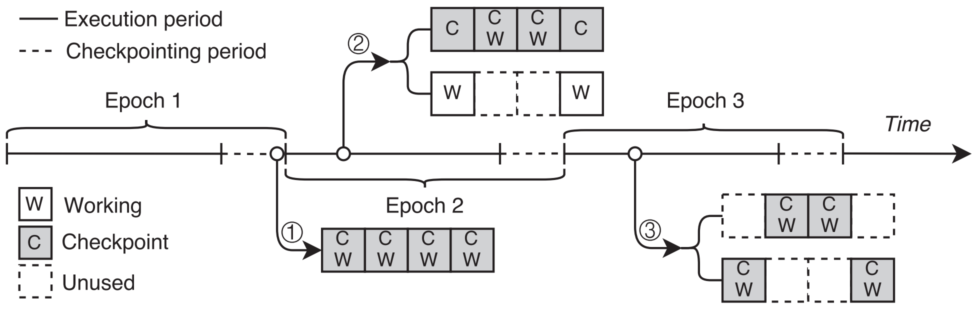

Fig. 3. A page through consecutive epochs of dual-page checkpointing.

vectors is (0, 0). Now suppose a write request arrives, as the checkpoint data cannot be overwritten for consistency protection, the new value should be put in the derivative page as separate working data, so the state of ❶ transfers to ❷.

In the state of ❷, the working data locates in the derivative page, and it is dirty. Thus, the bits of this state is (0, 1). If more writes to ❷ are received in the same epoch, then these writes can modify the working data directly, and the state of the cache line does not change. However, when the epoch ends and a new checkpoint is generated, the working data becomes part of the new checkpoint, so the state of the cache line will transfer to ❸.

The state of ❸ has two bits set to (1, 0), as the checkpoint locates in the derivative page and it is clean in the new epoch. Suppose now the cache line is to be modified, then the state shall transfer to ❹, whose bits are (1, 1). It follows a similar path as ❶ → ❷. ❹ keeps the state in the same epoch, because the dirty working data can be modified directly. Then after the epoch ends, it will transfer to ❶.

From the above process, we can see that the dual-page mapping realizes the two goals of efficiency: **(1)** A cache line writeback is mapped to a proper location without incurring any additional NVM data writes as logging or CoW does. The only requirement is to persist the mappings in checkpointing as well, which brings about only a few bits per cache line of NVM write on average. **(2)** The storage of cache-line level mappings is as low as 2 bits per mapping.

3.3 Execution and Checkpointing

To better illustrate how cache-line state transfers are aligned with epochs, we show the dual-page checkpointing and execution process with an example page in Figure 3. We focus on four cache lines in the page and examine their states through three consecutive epochs. We pick up three points in the timeline to explain.

When the first epoch finishes checkpointing, we assume the page is clean, and all cache lines are in the base page. There is no derivative page for it yet. Each cache line is in State ❶ of Figure 2.

During the execution period of the second epoch, any NVM write to the example page triggers allocation of the derivative page. Particularly, suppose the first and fourth cache lines are modified, then the new values are written to the derivative page as separate working data. The checkpoint and working data for the two cache lines diverge as shown in ②. Note that unlike logging or page-level CoW, our mechanism does not copy any extra data.

When it comes to the third epoch, cache lines that were working data have been checkpointed. Suppose the first and fourth cache lines of the deviation page at ③ are not modified in this third epoch, they are both the checkpoint and working data. Meanwhile, cache lines that acted as a checkpoint in the previous epoch are no longer in use in the current epoch.

3.4 Recovery

In a checkpointing period, all dirty cache lines in CPU caches or a DRAM cache layer have to be written back to NVM. Besides, we need to persist the following information, referred to as *recovery context*, in a particular area in NVM: (1) CPU state, including registers and store buffers; (2) the address mappings; (3) bit vectors. Note that they cannot overwrite their previous version in NVM, because that version would be used if the current checkpointing period itself is interrupted by a crash. Instead, they should write to a standby location. As long as the standby copy is finished, the previous version is reclaimed. At that time, a new checkpoint is deemed to be safely generated. After that, we can start the next epoch. On recovery, we restore from NVM the CPU state, the page mappings and bit vectors (i.e., the recovery context). Then the system resumes running [42].

The correctness of recovery is guaranteed by both *isolation* and *atomicity* of the checkpointing process. Dual-Page mapping ensures isolation, because the checkpoint data is never overwritten by working data during execution. Atomicity is realized by setting a pre-defined switch pointer that references the valid version of recovery context (containing all address mappings). The pointer is reset after all dirty data of an epoch is written back to NVM, and the corresponding recovery context is fully persistent. Atomically switching to the new version of recovery context denotes the end of the checkpointing (also the end of the epoch).

3.5 Comparison to Other Designs

A common goal of ThyNVM [56] and our design is to utilize *both* fine-grained and coarse-grained mappings. But we introduce a different design paradigm than ThyNVM. ThyNVM employs both mapping granularities in *independent* tables for *separate* data areas. It uses 42 bits index for each dirty cache line and 36 bits index for each dirty page respectively. In contrast, we integrate both granularities into one table for *all* data. Consequently, our design reduces the space overhead of storing a fine-grained mapping from 42 bits to 2 bits, and thus solves the bottleneck of ThyNVM in its limited number of fine-grained mappings. Moreover, we largely simplify the design by avoiding costly page migrations between separate data areas in ThyNVM.

Page overlays [58] introduce a complex cache-line management mechanism to page-level mappings. Let alone extensive hardware changes (e.g., widened physical address, modified TLB), directly applying page overlays to checkpointing would lead to inefficient design. Page overlays checkpointing use a segment to store dirty cache lines for each dirty page. Each segment uses 64 5-bit slot pointers and a 32-bit vector indicating the free slots within a segment total of 352 bits. For each new epoch, it has to write existing overlays back to regular pages, incurring lots of extra writes. Compared to page overlays, our design not only reduces the metadata size from 352 bits per page[3] to 64 bits but also largely simplifies the metadata management by avoiding overlay segments and their migration.

Our design stores metadata (address mappings) around 20×/5× more space-efficiently than ThyNVM/page overlays in the *memory controller*, while the tradeoff is that our design incurs extra space in *NVM storage*. In practice, such additional space is only 3.7% of the NVM.[4] Considering the space capacity of the memory controller is much more scarce than NVM, our position in the tradeoff is favorable and advantageous.

3.6 Discussion: OS and Applications

This article focuses on the *architecture* design, but here we briefly discuss the potential influence on the operating system (OS) and applications. As described in Sections 3.1 and 3.4, our basic model is to resume the software execution from the latest valid checkpoint. We also guarantee to do checkpointing in any period of t CPU time. The software can adopt this model in two ways. First, the OS buffers output of an application to external systems (e.g., an acknowledgment to a client) for at least t time so that any externally visible output is not invalidated or duplicated when resuming from a checkpoint. In this case, the applications do not need to be modified. Such an approach has been practiced by external synchrony [43]. Second, we let applications manage their output to external systems. An application has to group, for example, committed transactions by time and acknowledge the persistence of any group older than t. That is equivalent to a traditional group commit behavior. Also, note that there is no problem with a long transaction spanning multiple epochs, because the transaction can resume in recovery as well. The only requirement is that its output is buffered until the transaction has been checkpointed. Dual-Page checkpointing supports applications to adjust the length of the epoch. Transaction applications should decide the epoch at the beginning of execution, to ensure that most of the transactions can finish in an execution period.

In either case, we do not assume that peripheral device states (e.g., the network card state) are checkpointed [55, 56], because doing so is costly [42]. That means the software may encounter device errors upon recovery. Handling those device errors should not affect the persistent data consistency. Such an assumption is practical and widely adopted (e.g., by most transactional storage systems) [49, 67, 77].

Finally, the output buffering behavior enlarges the response latency of interactive applications. Since the checkpointing interval t also influences the throughput of applications (Section 5.6), it makes a tradeoff between latency and throughput, similar to that of a traditional group commit optimization. A high checkpointing frequency or short interval drags the throughput but enables faster responses. In practice, it can be determined by users' requirement such as SLA. Particularly, we can set the checkpointing interval as the required response time minus the network round-trip delay.

4 IMPLEMENTATION

In this section, we describe an implementation of dual-page checkpointing on a DRAM + NVM hybrid persistent memory architecture. We will first overview the architecture, then discuss implementation details, and finally give a summary by putting them together.

4.1 Architecture Overview

Both academia and industry show a trend to construct a hybrid DRAM+NVM memory system, as the performance and endurance of NVM is worse than DRAM. Figure 4 shows the hybrid memory

[3]One implication of Figure 1 is that, in most workloads, less than half of a dirty page is updated. We regard this as the typical case and calculate the space overhead of page overlays.
[4]We allocate 300MB derivative pages for 8GB NVM.

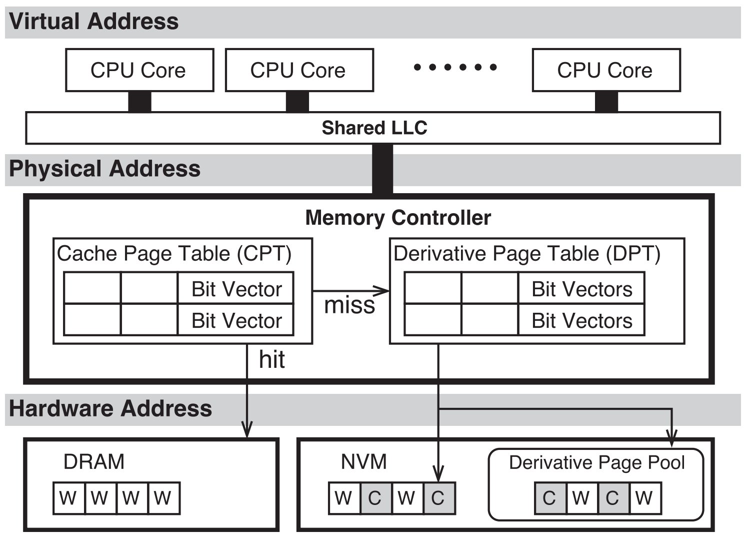

Fig. 4. Architecture overview of dual-page checkpointing.

architecture we use. It adopts DRAM as the cache of NVM. The DRAM cache can filter a significant portion of memory accesses [20, 52], which is a great help for memory performance and NVM lifetime. Our design can also be easily extended to the other hybrid memory architecture that DRAM and NVM share physical address space.

In the memory controller, Cache Page Table (CPT) records which page is cached in DRAM. Each CPT entry is associated with a bit vector to denote which cache line of the page is dirty. Therefore, when a DRAM-cached page is checkpointed, only dirty cache lines are written back to NVM.

Derivative Page Table (DPT) records the metadata of dual-page mapping, as described in Section 3.2. DPT and NVM handle two sources of data writes: evicted dirty DRAM pages during the execution period, and flush of dirty DRAM pages during checkpointing. The derivative page pool in NVM is used to allocate and manage the derivative pages. Next, we elaborate main functionalities and components of the system.

4.2 Address Translation

We introduce an additional address translation layer to the memory controller, as depicted in Figure 4. It translates a physical address to a *hardware address* according to dual-page mapping. The hardware address space has the same size as NVM, and is split into two regions, the major *base region* and the *derivative page pool*. This address translation approach makes our work transparent to other layers of the system, minimizing software and hardware modifications.

The physical hardware address translation is done by querying CPT and DPT. All accesses are first checked against CPT. The DRAM cache typically has a high hit ratio and can filter most memory accesses. Only missed accesses are looked up in DPT, and go through the dual-page mapping path to NVM. Such misses also trigger page replacement in the DRAM cache. Our implementation uses a LRU policy for cache replacement (Section 5.1.1).

4.3 Metadata Management

DPT stores page-level mappings and bit vectors in the memory controller for dual-page mapping. When a page is managed by dual-page mapping, we allocate a derivative page and a corresponding DPT entry for it. A full DPT entry has four fields, as listed in Table 2:

Table 2. DPT Entry Structures in Different Storage Media

Location	BPI	CPBV	DBV	Valid
Controller	36 bits	(64 bits)	(64 bits)	1 bit
DRAM	—	64 bits	64 bits	—
NVM	36 bits	64 bits	—	1 bit

Parentheses denote bits that can be omitted and offloaded to slower storage.

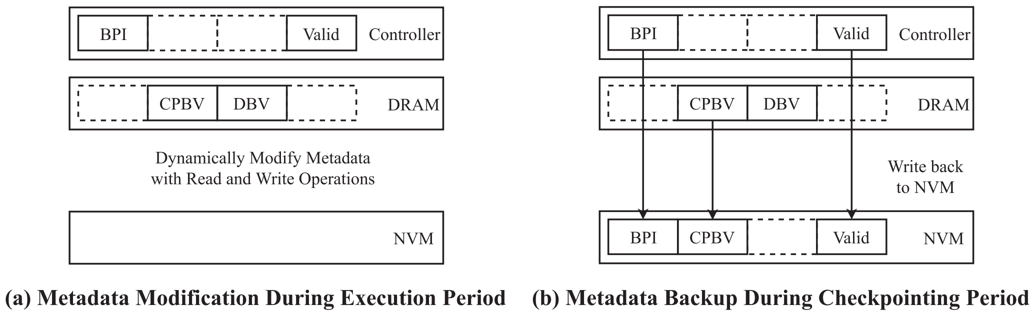
BPI
Valid
Controller
CPBV
DBV
DRAM
Dynamically Modify Metadata with Read and Write Operations
NVM
(a) Metadata Modification During Execution Period
BPI
Valid
Controller
CPBV
DBV
DRAM
Write back to NVM
BPI
CPBV
Valid
NVM
(b) Metadata Backup During Checkpointing Period

Fig. 5. Derivative page table management.

(1) Base Page Index (BPI), storing the higher-order bits of the physical address of a page;
(2) Checkpoint Position Bit Vector (CPBV), indicating whether the checkpoint of each cache line of the page is physically stored in the base or the derivative (every bit represents a cache line);
(3) Dirty Bit Vector (DBV), indicating whether each cache line of the page has been modified in the current epoch (every bit represents a cache line);
(4) Valid flag, denoting whether the DPT entry is in use.

The memory controller contains a small piece of SRAM to store the metadata [52, 53, 56, 58]. Ideally, all bit vectors can be stored in the memory controller so that address translation for individual cache lines is very fast. However, SRAM is expensive and less dense than DRAM, thus we can put the metadata in DRAM and cache frequently used entries in the memory controller.

BPI and the valid flag are used by every NVM access so that they are completely stored in the memory controller. However, the space-consuming bit vectors (CPBV and DBV) are only used when a NVM access matches a BPI. So, by default, we store the bit vectors in DRAM, and only cache the bit vectors for *one* DPT entry in the memory controller, as shown in Figure 5. It turns out that such a small cache is very effective, because both DRAM cache replacement and checkpointing are done in the page granularity. Two bit vectors, once cached, can serve all cache lines of a page. According to our evaluation in Section 5.5, compared to an ideal setup that stores all bit vectors in the memory controller, our current implementation is only 3% lower on average. Through this optimization, only a small part (37 bits) of a DPT entry need to be stored in the memory controller, which reduces the storage space by 77.6%. As Table 3 shows, the metadata storage efficiency of dual-page checkpointing is one order of magnitude better than the two state-of-the-art checkpointing systems.

Figure 5 shows the management mechanism of DPT during execution period and checkpointing period. The metadata in controller and DRAM is used to record page states of dual-page mapping, while the metadata in NVM is only used for backup and recovery. The DPT entries are stored in controller, DRAM, and NVM sequentially. During execution period, the DPT in controller and DRAM is modified dynamically when the dirty cache lines are written back to NVM (more details

Table 3. Comparison of Metadata Space Overhead

System	Hardware Space Overhead
ThyNVM [56]	42 bits per cache-line
Page Overlays [58]	352 bits per page
Dual-Page Checkpointing	37 bits per page

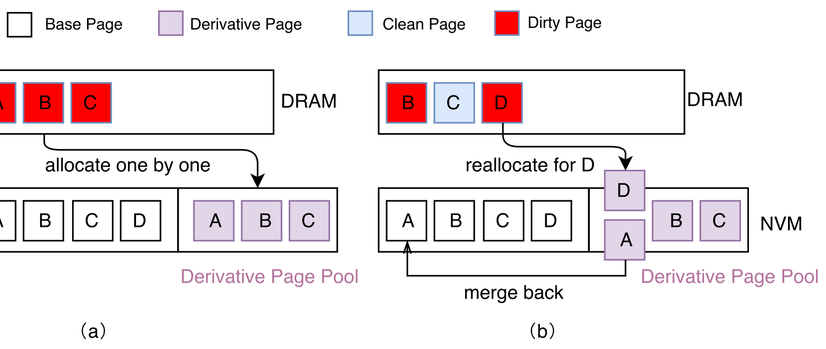

Fig. 6. Derivative page management.

in Figure 8). The modifications of DPT entries in DRAM will not be flushed back to NVM. During checkpointing period, the DPT in controller and DRAM will be written back to NVM for backup after all dirty pages in DRAM are written back to NVM, as shown in Figure 7. Such a NVM version of DPT stores CPBV for locating the checkpoint upon recovery. It does not store DBV, because the dirtiness information is unnecessary for recovery.

4.4 Derivative Page Management

Naively mandating derivative page management to OS would cause frequent interrupts, impairing the system performance. We implement a simple hardware-based derivative page management mechanism, which does not involve interrupts. We also design an optimization of pre-allocation.

4.4.1 Derivative Page Allocation. When a derivative page allocation request is received, the derivative page pool returns a free derivative page. The number of pages in the derivative page pool equals to the number of entries in the DPT. Initially, all pages in the derivative page pool are free, and all entries in the DPT have invalid flags. We allocate them one by one as Figure 6(a) shows.

4.4.2 Derivative Page Reclaim. The derivative page pool can be saturated as the dirty data written back to NVM increases. When there are no free derivative pages in derivative page pool, we reclaim clean pages as Figure 6(b) shows. To do so, the memory controller scans the DPT to find a clean page, and writes all its *checkpoint* cache lines in the derivative page back to the base page. Such writebacks do not violate consistency, because any clean page has only one version of data that acts as both the checkpoint and the working data, and those cache lines in the derivative page hold that version. Only after all writebacks are done, the derivative page can be reclaimed and its DPT entry is invalidated.

A clean page is either uncached in DRAM as Page A in Figure 6(b), or cached as Page C. Since most frequently used pages are cached, our policy is to first choose uncached pages to reclaim.

A derivative page allocation may fail if we are unable to find a clean page (we call this situation an allocation failure). That means all pages in the pool have been modified during the current epoch and there are no page for allocation in derivative page pool. In that case, we have to immediately start checkpointing, which will persist all those dirty pages as a new checkpoint and make them clean for the next epoch. Then all pages in the pool become reclaimable again.

4.4.3 Derivative Page Pre-allocation. The above reclaim process has two influences on derivative page allocation. First, the allocation is on the critical path. Before performing the NVM write, the controller has to wait for the allocation to finish. If the allocation entails an eviction, then the latency would be relatively large. Second, it is impossible to remedy a failed allocation. Although checkpointing can make all derivative pages clean and ready for allocation, it does not help in the following situation. If one allocation fails, then triggering checkpointing would cause more NVM writes, which would possibly cause more allocation failures.

Therefore, we pre-allocate derivative pages when their DRAM cache pages get dirty. Such an optimization is fairly efficient, because it moves the allocation overhead off the critical path, so that allocations can be conducted in parallel with applications execution. To perform pre-allocation, we need to detect whether the physical address of a DRAM write hits DPT. This detection is performed in the background without blocking DRAM requests. Thus, the allocation is totally off the critical path, and leaves small influence on memory access latency, because it mainly communicates with NVM while most memory requests are served by the DRAM cache during an execution period.

As we parallelize the allocation process and DRAM writes, some DRAM pages may get dirty before the derivative page pool reports an allocation failure. As a result, the dirty DRAM pages may have no derivative pages to accommodate them in the write path. We treat this rare situation in a special way. Dirty cache lines of those pages are logged into a buffer area in NVM, so that we do not lose data. We observed that the frequency of such pre-allocation failures is only 1.17 per epoch (0.01% of all per-allocations), resulting in negligible performance impact. Also, the space of 10 cache lines is enough for the special buffer area, costing only 720B.

4.5 Putting It All Together

4.5.1 During Execution. Control flow of Dual-Page checkpointing is shown in Figure 7. When Page A is evicted from DRAM by replacement algorithm, if Page A is clean, then Page A will be discarded directly. If Page A is dirty, then there are three possible paths in dual-page checkpointing system.

① If corresponding Derivative Page of Page A exists in NVM, then dirty cache lines of Page A can be written back to NVM directly by Address Translation in Section 4.2. The write path is shown in Figure 8.

②If Page A has no derivative page in NVM, then Dual-Page checkpointing will send a derivative page allocation request to Derivative Page Allocation (Section 4.4.1) mechanism. If there are free pages in derivative page pool, then Dual-Page checkpointing will allocate a derivative page to Page A, then, write dirty cache lines of Page A back to NVM (similar to ①).

③If Page A has no derivative page in NVM and there are no free pages in derivative page pool, then Dual-Page checkpointing will try to reclaim an existing derivative pages (Section 4.4.2). If Dual-Page checkpointing finds a clean page, then it will reclaim corresponding derivative page and allocate this derivative page to Page A, then write dirty cache lines of Page A back to NVM (similar to ①). If Dual-Page checkpointing cannot find a clean page, then it will return a derivative page allocation failure and immediately start checkpointing.

As illustrated in Figure 8, we show the read and write paths and metadata computation in detail. In a write path, a dirty page, Page A, is written back to NVM. Dual-Page checkpointing first updates the Dirty Bit Vector in DPT according to individual dirty cache lines of Page A. Then, it calculates a temporary bit vector P by Equation (1). The value of P indicates the position of working data. If

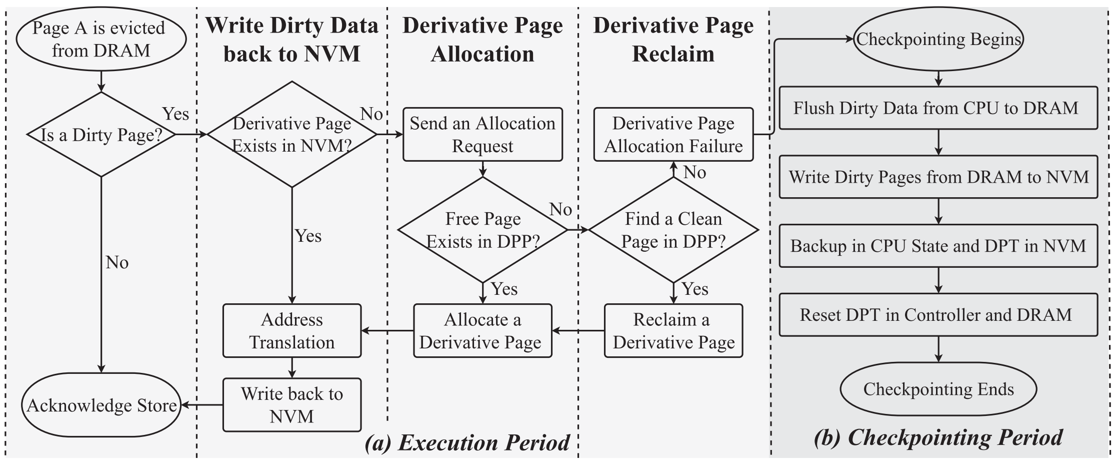
Page A is evicted from DRAM
Is a Dirty Page?
Yes
No
Acknowledge Store
Write Dirty Data back to NVM
Derivative Page Exists in NVM?
No
Yes
Address Translation
Write back to NVM
Derivative Page Allocation
Send an Allocation Request
Free Page Exists in DPP?
No
Yes
Allocate a Derivative Page
Derivative Page Reclaim
Derivative Page Allocation Failure
Find a Clean Page in DPP?
No
Yes
Reclaim a Derivative Page
(a) Execution Period
Checkpointing Begins
Flush Dirty Data from CPU to DRAM
Write Dirty Pages from DRAM to NVM
Backup in CPU State and DPT in NVM
Reset DPT in Controller and DRAM
Checkpointing Ends
(b) Checkpointing Period

Fig. 7. Control flow of **(a)** replacing a page back to NVM during execution period and **(b)** flushing all dirty data to NVM during checkpointing period.

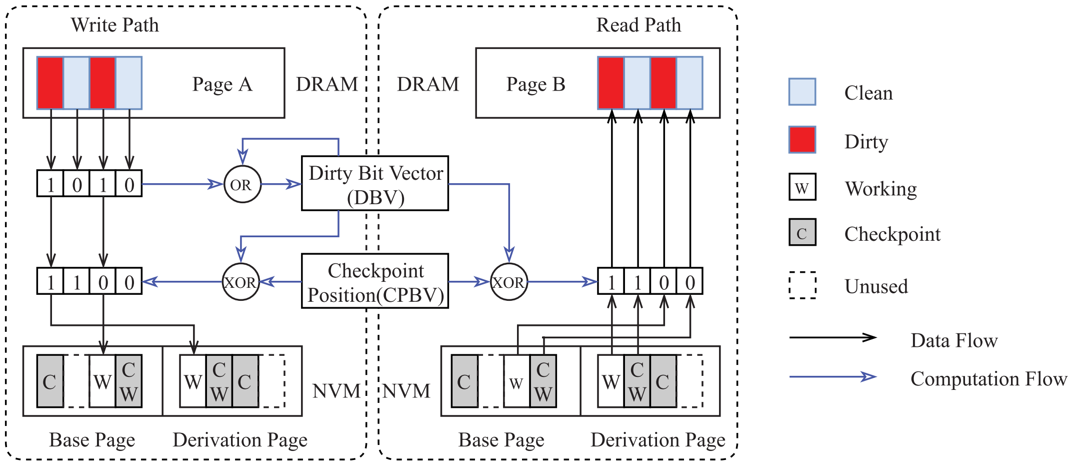

Fig. 8. Read and write path.

the kth bit of P is 1, then the kth cache line of the physical page will be written to the kth cache line of the derivative page. Otherwise, it will be written to the base page:

$$P = Dirtiness \oplus Checkpoint\ position. \tag{1}$$

In a read path, Page B will be read from NVM into DRAM. Dual-Page checkpointing finds cache lines from the base page and derivative page of Page B according to DPT. It only needs to calculate P by Equation (1), and reads cache lines from NVM according to the value of P and writes them to the DRAM page.

4.5.2 During Checkpointing. Dual-Page will flush all dirty data to NVM during checkpointing period, as shown in Figure 7. There are four steps: ①Flush dirty data from CPU to DRAM. ②Write dirty pages from DRAM back to NVM (similar to the path during execution period). ③Back up the CPU state and DPT (as described in Section 4.3) in NVM. ④Set DBV to zero, which means all pages in DRAM turn clean at the beginning of an epoch. Update CPBV so that the working data is set as checkpoint, as it becomes the checkpoint of the next epoch. After that, we can start the next epoch.

Table 4. Simulation Configuration

CPU	
Cores	4 @ 1.87GHz
L1 I Cache (Private)	128KB
L1 D Cache (Private)	128KB
L2 Cache (Shared)	2MB
-	-

Memory		
	DRAM	NVM
Capacity	128MB	4GB
Channels	1	2
Bandwidth	8GB/s	3.6GB/s (Read)
		1.3GB/s (Write)
Read/Write Latency (Normalized to DRAM)	1	4.4x (Read)
	1	12x (Write)

4.6 Discussion

4.6.1 Complexity and Scalability. To our knowledge, the hardware space/complexity of our design is the lowest of its class (compared to ThyNVM and page overlays). Currently, we implement most logic in the memory controller. We hope hardware development will make our design more affordable. Meanwhile, integrating page allocation with OS (or FPGA as demonstrated by KV-Direct [31]) is our future work.

For huge NVM, our work can scale well for two reasons: (1) data updates typically have locality even though NVM is huge, so that each single epoch hardly fills out DPT unreasonably fast; (2) some DPT entries can be offloaded to DRAM from the memory controller, with minimal overhead (we tested this case and the result is similar to Section 5.5.2).

4.6.2 Wear-Leveling. Because of the limitations of write endurance, wear-leveling techniques are important to NVM. The wear-leveling issues of dual-page checkpointing can be easily solved by related works. On the one hand, the technique in Reference [56] can solve the cacheline level wear-leveling issue in a NVM page. All cachelines of a NVM page can be periodically shifted. That only entails extra metadata of 6 bits per page, to record the cacheline index offset. On the other hand, there are many available page-level wear-leveling techniques for NVM. DPP pool only requires a 300MB sequential space and allows OS to adjust its position flexibly based on the pages wear-out in NVM.

5 EVALUATION

5.1 Methodology

5.1.1 Experiment Platform. Our experiments run on MARSSx86 [46], a full system simulator for x86-64 architecture. We set up a quad-core CPU with out-of-order pipeline. We adopt HybridSim [63] to simulate the hybrid memory system, which consists of both DRAM and PCM. It takes DRAM as the cache of PCM and adopts LRU for the cache replacement algorithm. DRAM is a 256MB DDR3 device simulated by DRAMSim2 [57], and PCM is simulated by NVMain [48] with a size of 8GB. Table 4 provides a more detailed description of the simulated system's configuration.

We proposed two implementation strategies about the latency of DPT and CPT: (1) use n RAMs instead of one to enable parallel lookups, which does not require extra storage but increases hardware complexity and power consumption. (2) use a k-bit index in each RAM, which, however, sacrifices full associativity. A manufacturer can determine n and k according to practical constraints. We assume n = 256 and k = 4 (whose feasibility was confirmed by a hardware engineer), so the lookup overhead is set to 16 cycles in our simulator. Furthermore, we also evaluated 32-cycle lookup, which introduced only around 2% slowdown.

5.1.2 Evaluated Systems. We compare our checkpointing solution with a baseline, four typical *hardware* checkpointing systems and two *software* transaction systems. They are all deployed to the above platform, with the same setup unless noted otherwise.

Table 5. Workloads

Name	Workload	DRAM Cache Miss Rate	Read/Write Ratio	Memory footprints
SPEC1	leslie3d sjeng sphinx3 wrf	16.08%	4.54	1,199MB
SPEC2	GemsFDTD lbm milc soplex	5.43%	1.80	1,473MB
PR	PageRank	17.61%	2.23	1,548MB
CD	Community Detection	15.14%	1.91	1,357MB
CC	Connected Components	5.30%	2.27	1,208MB
Ann	Ann	13.26%	2.15	773MB
Flann	Flann	12.58%	1.55	576MB
TATP	TATP (hash table)	56.72%	2.47	1,247MB
TPCC	TPC-C (hash table)	53.96%	1.72	864MB
Echo	Echo	43.57%	1.43	818MB
N-Store	N-Store	35.82%	1.38	1,090MB

- *Baseline.* The bare platform above, without any mechanism for crash consistency or data persistence.
- *Logging.* An undo-logging-based hardware checkpointing system that operates data at cache-line granularity. For fairness, we optimize it by adding an ideal, unbounded mapping table to avoid logging multiple writes to the same address in one epoch [61].
- *CoW.* A copy-on-write-based hardware checkpointing system that manages data at the page granularity. It also uses an ideal, unbounded mapping table.
- *ThyNVM.* A port of ThyNVM [56] to our simulator. We set the size of its fine-grained address translation tables to the same size as our DPT, but give extra space for its coarse-grained address translation tables.
- *PageOverlay.* An implementation of key mechanisms of page overlays [58], applied to the checkpointing use case.
- *DudeTM* [36]. A software durable transactional memory library. DudeTM requires a relatively large DRAM, so we give it 4GB DRAM so that it reaches its best performance.
- *NVML* [21]. Intel's undo-logging-based software transaction library for persistent memory. NVML requires that CPU can directly access both DRAM and NVM, so DRAM is not configured as the cache of NVM in this case.

5.1.3 Workloads. The workloads consist of two write-intensive combinations of SPEC CPU 2006 benchmarks [62], three algorithms on GraphChi [27], two common machine-learning algorithms [39, 40], two typical transaction workloads [60, 65], and two persistence workloads from WHISPER [41], as listed in Table 5. By default for each workload, we run a warm-up phase and evaluate 1.5B instructions with checkpointing per 30M cycles.

5.2 Comparison with Hardware Systems

We compare dual-page checkpointing with the hardware checkpointing systems in terms of both execution time and NVM write traffic. We first show basic facts and then summarize main observations.

Execution Time. Figure 9(a) shows the performance of all hardware systems normalized to that of dual-page checkpointing. The average slowdown of Logging, CoW, ThyNVM, and PageOverlay,

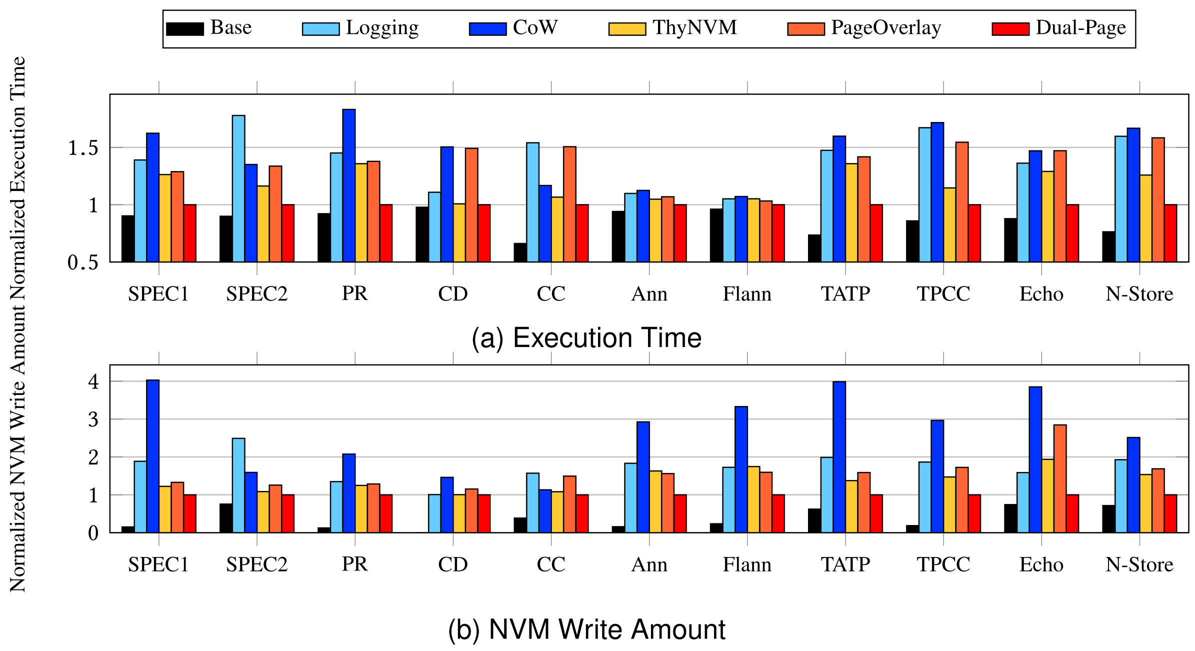

Fig. 9. Performance of hardware systems under various workloads (normalized to dual-page checkpointing).

Table 6. Checkpointing Count (Ckpt Count) and the Ratio of Total Checkpointing Time over Total Execution Time (Ckpt Time Ratio) for Each Workload Running 1.5B Instructions

Workloads	**SPEC1**		**SPEC2**		**PR**		**CD**		**CC**		**Ann**	
	Ckpt Count	Ckpt Time Ratio	Ckpt Count	Ckpt Time Ratio	Ckpt Count	Ckpt Time Ratio	Ckpt Count	Ckpt Time Ratio	Ckpt Count	Ckpt Time Ratio	Ckpt Count	Ckpt Time Ratio
Logging	7	16.30%	18	25.63%	12	25.06%	5	7.31%	18	39.02%	13	4.05%
CoW	7	18.41%	16	18.75%	14	29.51%	7	5.67%	16	35.91%	13	4.05%
PageOverlay	7	15.37%	15	18.62%	12	24.37%	7	5.66%	20	38.24%	13	4.04%
Dual-Page	6	10.49%	14	14.94%	11	18.61%	5	4.05%	15	27.86%	12	2.57%
Workloads	**Flann**		**TATP**		**TPCC**		**Echo**		**N-Store**		**Average**	
	Ckpt Count	Ckpt Time Ratio	Ckpt Count	Ckpt Time Ratio	Ckpt Count	Ckpt Time Ratio	Ckpt Count	Ckpt Time Ratio	Ckpt Count	Ckpt Time Ratio	Ckpt Count	Ckpt Time Ratio
Logging	11	5.28%	35	14.27%	30	15.78%	8	11.13%	31	14.53%	17.09	16.21%
CoW	11	5.28%	38	15.23%	34	16.95%	13	13.08%	33	15.03%	18.36	16.17%
PageOverlay	11	5.29%	34	13.59%	34	16.84%	13	13.08%	28	13.89%	17.64	15.36%
Dual-Page	11	3.79%	28	11.55%	26	12.97%	7	7.89%	24	12.46%	14.45	11.56%

over dual-page checkpointing, is 1.41×, 1.47× 1.18×, and 1.37×, respectively. Meanwhile, dual-page checkpointing only introduces 14% overhead compared to the baseline.

NVM Write Amount. Figure 9(b) depicts the amount of NVM writes occurred under different workloads. All the numbers are normalized to dual-page checkpointing. The average NVM write amounts of Logging, CoW, ThyNVM, and PageOverlay are 1.80×, 2.68×, 1.40×, and 1.62× that of dual-page checkpointing, respectively.

Checkpointing Count and Time. Table 6 lists the checkpointing count (Ckpt Count) and the ratio of total checkpointing time over total execution time. Each workload runs for 1.5B instructions, and the evaluated system does checkpointing every 30M cycles. ThyNVM can overlap its execution phases and checkpointing phases, while other systems, Logging, CoW, PageOverlay, and Dual-Page, have to stall the application during checkpointing.

Analysis. Based on Figure 9 and Table 6, we can make the following observations.

- In general, the amount of NVM writes largely affects the hardware system performance. For example, Logging outperforms CoW in SPEC1, PR, CD, TATP, TPCC, Echo, and N-Store as shown in Figure 9(a). That is because the *cache-line* granularity of Logging leads to less

NVM writes than CoW at the *page* granularity, as shown in Figure 9(b). Logging can execute more instructions per 30M cycles and finish faster than CoW, so the checkpointing count of Logging is less and the checkpointing time ratio is lower than CoW, as shown in Table 6. In SPEC2 and CC, CoW writes less to NVM than Logging, and thus has better performance. The reason is that those workloads have high spatial locality such that CoW and Logging checkpoints a similar amount of *data*. However, CoW records much less *metadata*, at the page level, than Logging whose checkpoint metadata is at the cache-line level. CoW can execute more instructions per 30M cycles and finish faster than Logging, so the checkpointing count of CoW is less and the checkpointing time ratio is lower in SPEC2 and CC.

- In certain workloads that are not very write-intensive (e.g., Ann and Flann), all systems show similar performance, checkpointing counts, and checkpointing time ratios, but they still incur different amounts of NVM writes. Saving on NVM write traffic, even though having little impact on application performance, helps extend the limited lifetime of NVM devices.
- Under most workloads, ThyNVM and PageOverlay outperform Logging and CoW. This confirms with the results in the original papers of ThyNVM [56] and PageOverlay [58].
- ThyNVM implements a new technique that can overlap the execution phases and checkpointing phases, but Dual-Page still outperforms ThyNVM. That is mainly because ThyNVM has more NVM write amount than Dual-Page in most workloads. On the one hand, to implement the overlap technique, ThyNVM needs to keep multiple versions of checkpointing and execution data in NVM, which leads to more NVM space overhead and NVM write amount. On the other hand, ThyNVM supports *cache-line* granularity checkpointing for only a small part of dirty cache lines because of the limitation of metadata overhead in memory controller. For the workloads with poor locality (e.g., TATP, TPCC, Echo, and N-Store), ThyNVM still suffers from additional write problem, as shown in Figure 9(b).
- In SPEC2, PR, and CD, PageOverlay has the similar amount of write operations compared with ThyNVM and Dual-Page, but the performance is different. This is caused by another influential factor whether the NVM traffic happens in the critical path of memory writes. Dual-Page and ThyNVM can safely acknowledge a write-back before actual NVM writing during epoch execution. In contrast, PageOverlay can only do so when no page overlays have to be moved to larger segments; otherwise, such movement is in the critical path. Table 6 shows that the checkpointing count of Dual-Page is less and the checkpointing time ratio is lower than PageOverlay in SEPC2, PR, and CD. This explains why PageOverlay may incur a similar NVM write amount but result in worse performance.
- In all workloads, dual-page checkpointing exhibits a remarkably better performance, writes less to NVM, and has the minimal checkpoint count and checkpointing time ratio than all other hardware checkpointing solutions.

In conclusion, since dual-page checkpointing manages checkpoint data in cache-line granularity and compacts metadata to the page level, it incurs a minimal amount of NVM writes and enjoys the best performance among state-of-the-art hardware checkpointing systems.

5.3 Comparison with Software Systems

We compare the performance of dual-page checkpointing with the software systems. We implemented TATP and TPCC benchmarks using the transactional memory APIs of DudeTM and NVML. In this experiment, TATP and TPCC each are run with four working threads. We also modified the

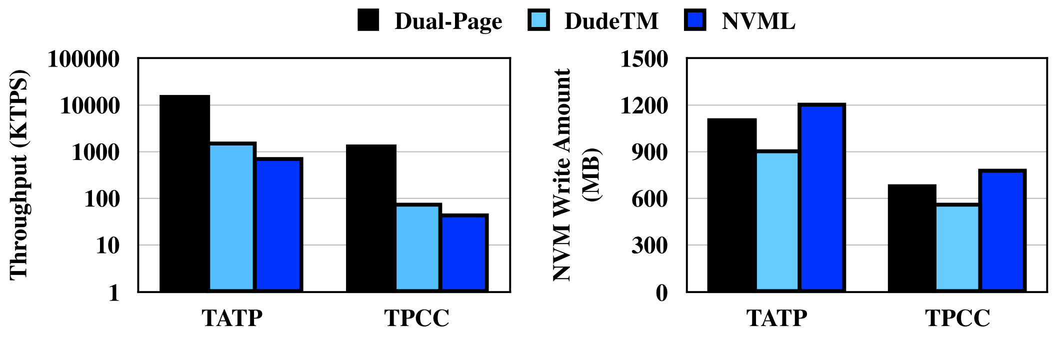

Fig. 10. Performance comparison between hardware-based and software-based data persistence.

code of the two benchmarks so that they can run on our hardware checkpointing system for data persistence (see Section 3.1).

For fairness, we set our checkpointing interval equal to the group commit interval of DudeTM, which is 16 ms. That means the response latency on DudeTM is basically the same as that on dual-page checkpointing. The latency on NVML is much lower (96–323 μs) as every transaction is acknowledged synchronously.

Next, we focus on the throughput and the amount of NVM writes of the three evaluated systems.

Throughput. From Figure 10, we can see that our hardware-based solution is 13.6× and 24.9× faster than DudeTM and NVML, respectively, in terms of throughput.

NVM Write Amount. Figure 10 shows that dual-page checkpointing incurs a larger amount of NVM writes than DudeTM, and NVML has the most NVM writes. That is because dual-page checkpointing has to flush CPU states, DPT and all the dirty data in DRAM to NVM in every epoch. DudeTM generates redo logs in DRAM and group commit them to NVM. In addition, it performs a log combination optimization that further reduces the amount of NVM writes. NVML generates and records all undo logs directly on the NVM, so it incurs most NVM writes.

Analysis. The DudeTM case clearly demonstrates that, with the fast persistent memory, the software overhead becomes dominant. Such an observation confirms with the finding in the original paper of DudeTM [36] (i.e., log flushing is not the bottleneck of DudeTM). Our dual-page checkpointing solution eliminates most of the software overhead and significantly improves the system performance. This is achieved by utilizing the natural address translation capability of the memory controller (a software implementation is not easy to do) and the efficient dirty data tracking mechanism in dual-page checkpointing.

5.4 Impact of Derivative Pages

Our solution needs to reserve a part of NVM as the derivative page pool (DPP) for storing derivative pages. The question is how the DPP size influences overall system performance. A too small size may increase the chance of derivative page allocation failures, thus degrading system performance and increasing NVM writes. Meanwhile, a large DPP would reduce the available NVM space to applications. We consider both system performance and NVM write amount in making the choice.

Execution Time. Figure 11(a) shows workload execution time with dual-page checkpointing of different DPP sizes. The smallest size is 100MB in our experiments, but it only degrades the performance by 3.2%. For write-intensive workloads, the smallest pool size (100MB) only results in 3% performance overhead compared to the largest pool size (500MB). This demonstrates the effectiveness of our derivative page management mechanism. Particularly, most of derivative page

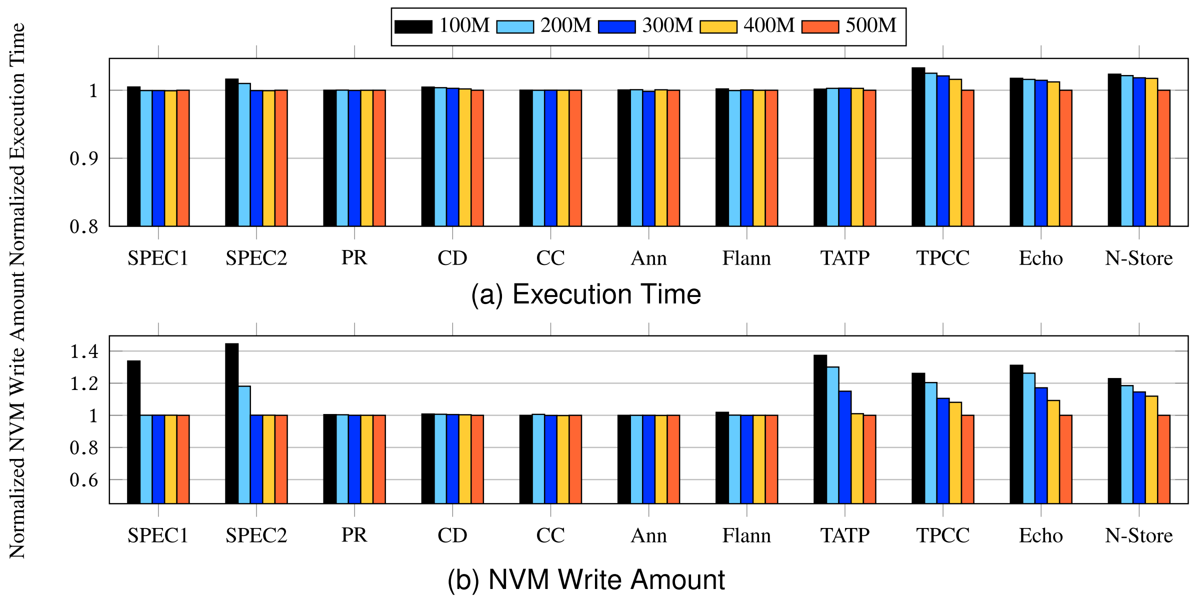

Fig. 11. Performance with different DPP sizes (normalized to pool_size = 500MB).

pre-allocation and merge operations are off the critical path, so it has limited influence on execution time (see Section 4.4).

NVM Write Amount. Figure 11(b) shows the relative amount of NVM writes for various workloads as the size of DPP changes. Shrinking DPP increases the amount of NVM writes, because clean derivative pages are frequently merged back to their base pages to accommodate dirty data. For write-intensive workloads, a 100MB derivative page pool increases the NVM write amount by up to 1.44×, compared to a 500MB size.

Analysis. We observe that shrinking DPP has a small impact on the system performance, but a large impact on the NVM write amount. Overall, we suggest reserving 300MB for DPP (i.e., 3.7% of the total NVM in our setup), because it incurs up to 20% extra writes, which is less than the average extra writes incurred by other evaluated systems, and those writes impose negligible influence on system performance. As NVM can supply storage class capacity, such a small extra space overhead is acceptable.

5.5 Metadata Overhead

We evaluate the metadata overhead and the impact of our three-version DPT optimization (i.e., the NVM version, the DRAM version, and the memory controller version) on the performance.

5.5.1 Space Overhead. Every derivative page has an entry in DPT, and the size of DPP determines the size of DPT. As discussed above, the best-fit size for DPP is 300MB, so the sizes of NVM, DRAM and memory controller versions of DPT are 947KB, 1.17MB, 347KB respectively. Such space overheads are trivial compared to the NVM capacity (8GB) and DRAM capacity (256MB). Even for SRAM in the memory controller, hundreds of KBs are acceptable [52].

5.5.2 Performance Overhead. To show the impact of DPT on system performance, we evaluate the systems with different DPT cache sizes. We follow the default configuration where DPP is 300MB and DPT stores 76,800 entries. Below is a list of cache sizes we evaluated.

- *All Metadata in MC (1.51MB).* The ideal case where the full DPT is stored in the memory controller.

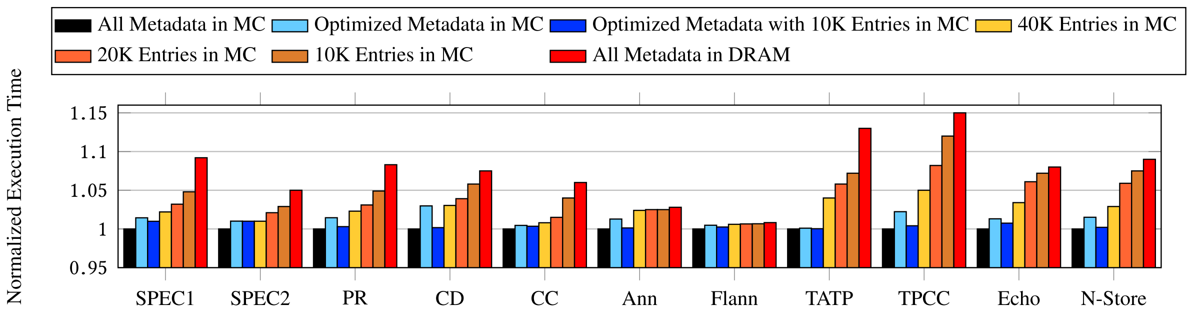

Fig. 12. Impact of reading metadata from DRAM on the performance of dual-page checkpointing (normalized to the case when all metadata is in the memory controller).

- *Optimized Metadata in MC (347KB).* The optimized case where we only cache most frequently used fields (Base Page Index of 36 bits and Valid Bit of 1 bit) of each DPT entry, to reduce the space overhead of the cache in the memory controller. Dual-Page will read the rest two fields (CPBV and DBV) from DRAM when the Valid Bit is set.
- *Optimized Metadata with 10K Entries in MC (504KB).* The case where optimized metadata (BPI and VB of all DPT entries) and 10,000 (CPBV and DBV) entries are cached in the memory controller.
- *40K Entries in MC (806KB).* The case where 40,000 DPT entries are cached in the memory controller.
- *20K Entries in MC (403KB).* The case where 20,000 DPT entries are cached in the memory controller.
- *10K Entries in MC (202KB).* The case where 10,000 DPT entries are cached in the memory controller.
- *All Metadata in DRAM.* The worst case where all DPT entries are stored in DRAM. Dual-Page will read the corresponding entry from DRAM when dirty pages are written back to NVM.

We observe that, for most workloads, the performance is sensitive to the number of cached entries in the memory controller. Especially for workloads with low locality (e.g., TATP and TPCC), the performance of the worst case (All Metadata in DRAM) is 1.13× and 1.15× lower than the ideal case (All Metadata in MC), respectively. We also observe that, for all workloads, the performance difference is within 3% between the ideal case (All Metadata in MC) and the optimized case (Optimized Metadata in MC). For all workloads, the performance of Optimized Metadata with 10K Entries is very close to the ideal case, and the performance of the optimized case is better than the case where 40K Entries are cached in MC.

Through our optimization, a major part (77.6%) of DPT does not need to be stored in the memory controller, without sacrificing much performance. This is very important, because the hardware space in the memory controller is scarce and easily makes the bottleneck.

5.6 Checkpointing Interval

To show how different checkpointing intervals impact the performance of hardware checkpointing systems, we evaluate all workloads and present the results in Figure 13. We observe that the performance of all the systems rises as the checkpointing interval is increasing. That is because the increase of checkpointing interval enables more NVM writes to coalesce and consequently reduces the total amount of data to be checkpointed. Another observation is that our dual-page checkpointing system exhibits a great performance advantage over others with different checkpointing intervals.

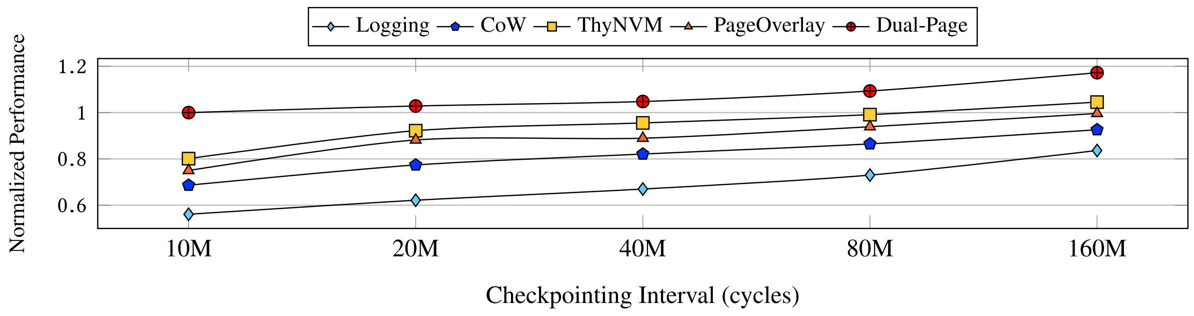

Fig. 13. Average execution time of workloads on different evaluated systems, over various checkpointing intervals (normalized to dual-page checkpointing at 10M cycles interval).

Although an extended checkpointing interval leads to a better throughput, it also means a long response latency for interactive workloads. Take an interval of 30M cycles (16ms) for example. Whether such a latency is acceptable depends on the application requirement. In this setup, the average checkpointing delay among workloads is between 3.7M and 4.8M cycles (12.4%–15.9% of the interval), which degrades the throughput. All in all, the tradeoff between throughput and latency, dictated by the checkpointing interval, is up to the users.

6 RELATED WORK

Traditional disk-based checkpointing systems [6, 14, 71, 72] backup memory data and CPU state in disks during checkpointing. However, high-frequency checkpointing by these traditional schemes will bring unacceptable performance overhead and I/O traffic.

DRAM-based checkpointing [50, 61] has been proposed to address the disk bottleneck. It achieves low access latency, but the volatility of DRAM is still a problem. Although double-memory checkpointing [76] and ReVive [50] can mitigate this problem to a certain extent, they suffer from heavy communication cost.

Other NVM-based checkpointing solutions [11, 37, 78] address different problems with ours. For example, the pre-copy mechanism to parallelize partial checkpointing and execution [17, 24]. In contrast, our focus is to minimize the metadata overhead and extra writes incurred by traditional checkpoint consistency protection techniques.

Prior works [23, 51, 61] also improved the logging technology, to obtain a better performance than copy-on-write and store data more compactly. Shadow memory [59, 69] is another commonly used solution, which maintains a shadow page or shadow cache line. This mechanism is similar to undo logging, however, it supplies the ability to quickly access its shadow storage, which gives it a higher chance to optimize checkpointing. As we explained before, however, logging and shadow memory cause excessive extra writes if applied to a checkpointing hardware.

Approximate computing/storage [8, 68] provides a parallel approach to efficient data persistence for in-memory applications. However, losing data quality is not free. As data loss is indeterministic, programmers have to evaluate and control the influence of data loss/approximation and may change code accordingly. Such burden/complexity become avoidable, since our checkpointing solution shows a minor 12% performance slowdown. It is up to applications to choose either approach according to their specific demands.

7 CONCLUSION

In this article, we present the design and implementation of a novel dual-page checkpointing system for the DRAM + NVM hybrid memory architecture. The core of our system is the dual-page mapping scheme, which protects the checkpoint consistency. We summarize key insights and findings as below.

- The checkpointing-based architectural approach to data persistence is efficient and beneficial. It largely simplifies and outperforms state-of-the-art software transaction systems.
- The efficiency of our solution comes from the dual-page mapping design. It combines dual physical pages and bit vectors to achieve space efficient cache-line level mapping and avoid most extra NVM writes. It remarkably outperforms state-of-the-art hardware checkpointing solutions.
- Our implementation and experiments show that the derivation page pool costs only 3.7% of the total NVM size. Our solution also offers flexibility in choosing the checkpoint interval.

REFERENCES

[1] Saurabh Agarwal, Rahul Garg, Meeta S. Gupta, and Jose E. Moreira. 2004. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the 18th Annual International Conference on Supercomputing (ICS'04).* ACM, 277–286. DOI: https://doi.org/10.1145/1006209.1006248

[2] H. Akinaga and H. Shima. 2010. Resistive random access memory (ReRAM) based on metal oxides. *Proc. IEEE* 98, 12 (2010). DOI: https://doi.org/10.1109/JPROC.2010.2070830

[3] Alluxio Open Foundation. 2017. Open Source Memory Speed Virtual Distributed Storage. Retrieved from http://www.alluxio.org/.

[4] Dmytro Apalkov, Alexey Khvalkovskiy, Steven Watts, Vladimir Nikitin, Xueti Tang, Daniel Lottis, Kiseok Moon, Xiao Luo, Eugene Chen, Adrian Ong, Alexander Driskill-Smith, and Mohamad Krounbi. 2013. Spin-transfer torque magnetic random access memory (STT-MRAM). *ACM J. Emerg. Technol. Comput. Syst.* 9, 2, Article 13 (May 2013), 35 pages. DOI: https://doi.org/10.1145/2463585.2463589

[5] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD'15).* 1383–1394. DOI: https://doi.org/10.1145/2723372.2742797

[6] Austin R. Benson, Sven Schmit, and Robert Schreiber. 2015. Silent error detection in numerical time-stepping schemes. *Int. J. High Perform. Comput. Appl.* 29, 4 (2015), 403–421. DOI: https://doi.org/10.1177/1094342014532297

[7] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Viglas. 2015. REWIND: Recovery write-ahead system for in-memory non-volatile data-structures. *Proc. VLDB Endow.* 8, 5 (Jan. 2015), 497–508. DOI: https://doi.org/10.14778/2735479.2735483

[8] Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. 2013. Analysis and characterization of inherent application resilience for approximate computing. In *Proceedings of the 50th Annual Design Automation Conference (DAC'13).* Article 113, 9 pages. DOI: https://doi.org/10.1145/2463209.2488873

[9] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11).* 105–118. DOI: https://doi.org/10.1145/1950365.1950380

[10] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles.* 133–146. DOI: https://doi.org/10.1145/1629575.1629589

[11] Xiangyu Dong, Naveen Muralimanohar, Norm Jouppi, Richard Kaufmann, and Yuan Xie. 2009. Leveraging 3D PCRAM technologies to reduce checkpoint overhead for future exascale systems. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis.* 57:1–57:12. DOI: https://doi.org/10.1145/1654059.1654117

[12] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System software for persistent memory. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys'14).* Article 15, 15 pages. DOI: https://doi.org/10.1145/2592798.2592814

[13] Ifeanyi P. Egwutuoha, David Levy, Bran Selic, and Shiping Chen. 2013. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *J. Supercomput.* 65, 3 (01 Sep 2013), 1302–1326. DOI: https://doi.org/10.1007/s11227-013-0884-0

[14] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. 2012. Detection and correction of silent data corruption for large-scale high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis.* 78:1–78:12.

[15] Balint Fleischer. 2016. Storage Class Memory in Scalable Cognitive Systems. *Keynote in Flash Memory Summit.* Retrieved from https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2016/20160809_Keynote5_Huawei_Fleischer.pdf.

[16] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. 2010. Availability in globally distributed storage systems. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. 61–74. Retrieved from http://dl.acm.org/citation.cfm?id=1924943.1924948.

[17] Shen Gao, Bingsheng He, and Jianliang Xu. 2015. Real-time in-memory checkpointing for future hybrid memory systems. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. 263–272. DOI: https://doi.org/10.1145/2751205.2751212

[18] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*. 29–43. DOI: https://doi.org/10.1145/945445.945450

[19] E. R. Giles, K. Doshi, and P. Varman. 2015. SoftWrAP: A lightweight framework for transactional support of storage class memory. In *Proceedings of the 31st Symposium on Mass Storage Systems and Technologies (MSST'15)*. 1–14. DOI: https://doi.org/10.1109/MSST.2015.7208276

[20] Tae Jun Ham, Bharath K. Chelepalli, Neng Xue, and Benjamin C. Lee. 2013. Disintegrated control for energy-efficient and heterogeneous memory systems. In *Proceedings of the IEEE 19th International Symposium on High Performance Computer Architecture*. 424–435. DOI: https://doi.org/10.1109/HPCA.2013.6522338

[21] Intel. 2016. The NVM Library. Retrieved from http://pmem.io/.

[22] Intel. 2017. Intel Optane Technology. Retrieved from http://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

[23] Ioannis Doudalis and Milos Prvulovic. 2012. Euripus: A flexible unified hardware memory checkpointing accelerator for bidirectional-debugging and reliability. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*. 261–272. DOI: https://doi.org/10.1109/ISCA.2012.6237023

[24] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic. 2013. Optimizing checkpoints using NVM as virtual memory. In *Proceedings of the IEEE 27th International Symposium on Parallel Distributed Processing*. 29–40. DOI: https://doi.org/10.1109/IPDPS.2013.69

[25] Kimberly Keeton. 2017. Memory-Driven Computing. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*. Retrieved from https://www.usenix.org/conference/fast17/technical-sessions/presentation/keeton.

[26] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. 2016. High-performance transactions for persistent memories. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*. 399–411. DOI: https://doi.org/10.1145/2872362.2872381

[27] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. 31–46. http://dl.acm.org/citation.cfm?id=2387880.2387884

[28] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu. 2013. Evaluating STT-RAM as an energy-efficient main memory alternative. In *Proceedings of the 2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'13)*. 256–267. DOI: https://doi.org/10.1109/ISPASS.2013.6557176

[29] B.C. Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, E. Ipek, O. Mutlu, and D. Burger. 2010. Phase-change technology and the future of main memory. *IEEE Micro* 30 (Jan. 2010), 131–141. DOI: https://doi.org/10.1109/MM.2010.24

[30] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable DRAM alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*. 2–13. DOI: https://doi.org/10.1145/1555754.1555758

[31] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-direct: High-performance in-memory key-value store with programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*. 137–152. DOI: https://doi.org/10.1145/3132747.3132756

[32] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. 2014. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC'14)*. Article 6, 15 pages. DOI: https://doi.org/10.1145/2670979.2670985

[33] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. 583–598.

[34] Harold Lim and Shivnath Babu. 2013. Execution and optimization of continuous queries with cyclops. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1069–1072. DOI: https://doi.org/10.1145/2463676.2465248

[35] Wei Lin, Haochuan Fan, Zhengping Qian, Junwei Xu, Sen Yang, Jingren Zhou, and Lidong Zhou. 2016. STREAMSCOPE: Continuous reliable distributed processing of big data streams. In *Proceedings of the 13th Usenix Conference*

on Networked Systems Design and Implementation (NSDI'16). 439–453. Retrieved from http://dl.acm.org/citation.cfm?id=2930611.2930640.

[36] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, and Jinglei Ren. 2017. DudeTM: Building durable transactions with decoupling for persistent memory. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*. 329–343. DOI: https://doi.org/10.1145/3037697.3037714

[37] A. Mirhosseini, A. Agrawal, and J. Torrellas. 2016. Survive: Pointer-based in-DRAM incremental checkpointing for low-cost data persistence and rollback-recovery. *IEEE Comput. Architect. Lett.* PP, 99 (2016), 1–1. DOI: https://doi.org/10.1109/LCA.2016.2646340

[38] Iulian Moraru, David G. Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan Binkert. 2013. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *Proceedings of the 1st ACM SIGOPS Conference on Timely Results in Operating Systems (TRIOS'13)*. Article 1, 17 pages. DOI: https://doi.org/10.1145/2524211.2524216

[39] David M. Mount and Sunil Arya. 1998. ANN: A library for approximate nearest neighbor searching. *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms.*

[40] M Muja and D. G. Lowe. 2014. Scalable nearest neighbor algorithms for high dimensional data. *IEEE Trans. Pattern Anal. Mach. Intell.* 36, 11 (2014), 2227–2240.

[41] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M Swift, Haris Volos, and Kimberly Keeton. 2017. An analysis of persistent memory use with WHISPER. *ACM SIGOPS Operat. Syst. Rev.* 51, 4 (2017), 135–148.

[42] Dushyanth Narayanan and Orion Hodson. 2012. Whole-system persistence. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*. 401–410. DOI: https://doi.org/10.1145/2150976.2151018

[43] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. 2006. Rethink the sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*. 1–14. http://dl.acm.org/citation.cfm?id=1298455.1298457

[44] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. 2011. Fast crash recovery in RAMCloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*. 29–41. DOI: https://doi.org/10.1145/2043556.2043560

[45] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. 2015. The RAMCloud storage system. *ACM Trans. Comput. Syst.* 33, 3, Article 7 (Aug. 2015), 55 pages. DOI: https://doi.org/10.1145/2806887

[46] A. Patel, F. Afram, Shunfei Chen, and K. Ghose. 2011. MARSS: A full system simulator for multicore x86 CPUs. In *Proceedings of the 48th ACM/EDAC/IEEE Design Automation Conference.* 1050–1055.

[47] Thao N. Pham, Panos K. Chrysanthis, and Alexandros Labrinidis. 2016. Avoiding class warfare: Managing continuous queries with differentiated classes of service. *VLDB J.* 25, 2 (2016), 197–221. DOI: https://doi.org/10.1007/s00778-015-0411-4

[48] M. Poremba, T. Zhang, and Y. Xie. 2015. NVMain 2.0: Architectural simulator to model (non-)volatile memory systems. *Comput. Architect. Lett.* PP, 99 (2015), 1–1. DOI: https://doi.org/10.1109/LCA.2015.2402435

[49] Vijayan Prabhakaran, Thomas L. Rodeheffer, and Lidong Zhou. 2008. Transactional flash. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. 147–160. Retrieved from http://dl.acm.org/citation.cfm?id=1855741.1855752.

[50] Milos Prvulovic, Zheng Zhang, and Josep Torrellas. 2002. ReVive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA'02)*. 111–122. Retrieved from http://dl.acm.org/citation.cfm?id=545215.545228.

[51] M. Prvulovic, Zheng Zhang, and J. Torrellas. 2002. ReVive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *Proceedings of the 29th Annual International Symposium on Computer Architecture.* 111–122. DOI: https://doi.org/10.1109/ISCA.2002.1003567

[52] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*. 24–33. DOI: https://doi.org/10.1145/1555754.1555760

[53] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. 2011. Page placement in hybrid memory systems. In *Proceedings of the International Conference on Supercomputing.* 85–95. DOI: https://doi.org/10.1145/1995896.1995911

[54] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam. 2008. Phase-change random access memory: A scalable technology. *IBM J. Res. Dev.* 52, 4 (July 2008), 465–479. DOI: https://doi.org/10.1147/rd.524.0465

[55] Jinglei Ren, Qingda Hu, Samira Khan, and Thomas Moscibroda. 2017. Programming for non-volatile main memory is hard. In *Proceedings of the 8th Asia-Pacific Workshop on Systems (APSys'17)*. Article 13, 8 pages. DOI: https://doi.org/10.1145/3124680.3124729

[56] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. 2015. ThyNVM: Enabling software-transparent crash consistency in persistent memory systems. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'15)*. 672–685. DOI: https://doi.org/10.1145/2830772.2830802 Retrieved from http://persper.com/thynvm/.

[57] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. 2011. DRAMSim2: A cycle accurate memory system simulator. *Comput. Architect. Lett.* 10, 1 (2011), 16–19. DOI: https://doi.org/10.1109/L-CA.2011.4

[58] Vivek Seshadri, Gennady Pekhimenko, Olatunji Ruwase, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, Todd C. Mowry, and Trishul Chilimbi. 2015. Page overlays: An enhanced virtual memory framework to enable fine-grained memory management. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. 79–91. DOI: https://doi.org/10.1145/2749469.2750379

[59] Weidong Shi, H. H. S. Lee, L. Falk, and M. Ghosh. 2006. An integrated framework for dependable and revivable architectures using multicore processors. In *Proceedings of the 33rd International Symposium on Computer Architecture*. 102–113. DOI: https://doi.org/10.1109/ISCA.2006.8

[60] Neuvonen Simo, Wolski Antoni, Manner Markk, and Raatikka Vilho. [n.d.]. Telecom Application Transaction Processing Benchmark. Retrieved from http://tatpbenchmark.sourceforge.net/.

[61] Daniel J. Sorin, Milo M. K. Martin, Mark D. Hill, and David A. Wood. 2002. SafetyNet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA'02)*. 123–134. http://dl.acm.org/citation.cfm?id=545215.545229

[62] Standard Performance Evaluation Corporation. [n.d.]. SPEC CPU 2006. Retrieved from http://www.spec.org/cpu2006.

[63] Jim Stevens, Paul Tschirhart, Mu-Tien Chang, Ishwar Bhati, Peter Enns, James Greensky, Zeshan Chisti, SL Lu, and B Jacob. 2013. An integrated simulation infrastructure for the entire memory hierarchy: Cache, DRAM, nonvolatile memory, and disk. *Intel. Technol. J.* 17, 1 (2013), 184–200.

[64] Nisha Talagala. 2016. The New Storage Applications: Lots of Data, New Hardware and Machine Intelligence. Keynote address. In *Proceedings of the 4th Workshop on Interactions of NVM/Flash with Operating Systems and Workloads*.

[65] The Transaction Processing Council. 2017. TPC-C benchmark Version 5. Retrieved from http://www.tpc.org/tpcc/.

[66] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. 18–32. DOI: https://doi.org/10.1145/2517349.2522713

[67] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST'11)*. 61–75. Retrieved from http://dl.acm.org/citation.cfm?id=1960475.1960480.

[68] Swagath Venkataramani, Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. 2013. Quality programmable vector processors for approximate computing. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'13)*. 1–12. DOI: https://doi.org/10.1145/2540708.2540710

[69] D. Vogt, C. Giuffrida, H. Bos, and A. S. Tanenbaum. 2015. Lightweight memory checkpointing. In *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 474–484. DOI: https://doi.org/10.1109/DSN.2015.45

[70] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*. 91–104. DOI: https://doi.org/10.1145/1950365.1950379

[71] John Paul Walters and Vipin Chaudhary. 2007. A scalable asynchronous replication-based strategy for fault tolerant MPI applications. In *Proceedings of the 14th International Conference on High Performance Computing*. 257–268.

[72] J. P. Walters and V. Chaudhary. 2009. Replication-based fault tolerance for MPI applications. *IEEE Trans. Parallel Distrib. Syst.* 20, 7 (2009), 997–1010. DOI: https://doi.org/10.1109/TPDS.2008.172

[73] Matei Zaharia. 2016. Continuous Applications: Evolving Streaming in Apache Spark 2.0: A foundation for end-to-end real-time applications. *Databricks Engineering Blog*. Retrieved from https://databricks.com/blog/2016/07/28/continuous-applications-evolving-streaming-in-apache-spark-2-0.html.

[74] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. 15–28. Retrieved from http://dl.acm.org/citation.cfm?id=2228298.2228301.

[75] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. 2015. Mojim: A reliable and highly available non-volatile memory system. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. 3–18. DOI: https://doi.org/10.1145/2694344.2694370

[76] Gengbin Zheng, Xiang Ni, and L.V. Kale. 2012. A scalable double in-memory checkpoint and restart scheme towards exascale. In *Proceedings of the IEEE/IFIP 42nd International Conference on Dependable Systems and Networks Workshops*. 1–6. DOI: https://doi.org/10.1109/DSNW.2012.6264677

[77] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. 2014. Fast databases with fast durability and recovery through multicore parallelism. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. 465–477. Retrieved from http://dl.acm.org/citation.cfm?id=2685048.2685085.

[78] Ruijin Zhou and Tao Li. 2013. Leveraging phase change memory to achieve efficient virtual machine execution. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 179–190. DOI: https://doi.org/10.1145/2451512.2451547

Received March 2018; revised October 2018; accepted October 2018