Optimizing Memory-based Checkpointing in Hybrid-Memory System

Michael Shell
School of Electrical and
Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia30332–0250
Email:http://www.michaelshell.org/contact.html

Homer Simpson
Twentieth Century Fox
Springfield, USA
Email:homer@thesimpsons.com

James Kirk
and Montgomery Scott
Starfleet Academy
San Francisco, California 96678–2391
Telephone: (800) 555–1212
Fax: (888) 555–1212

***Abstract*—Fault tolerance is a crucial problem for exascale systems. Checkpointing is common used to address this problem, however, in HDD-based high performance computing systems, it incurs huge overhead as its frequency increases. Memory-based checkpointing mitigates such huge overhead by sharing data between working memory and checkpoint, but encounters a new problem that any modification on working memory will corrupt the consistency of checkpoint. Traditional memory-based checkpointing adopts logging or copy-on-write to resolve this problem, with inevitable extra writes. For next generation memory system, which is composed by DRAM and emerging non-volatile memory, such as phase change memory (PCM), extra writes could be severe because the low PCM writing performance. Existing research has leveraged PCM to perform checkpointing, but they still suffer from unexpected double-space or double-writes because they treat PCM as storage device instead of memory.**

In this paper, we propose a memory-based checkpointing mechanism for next generation memory system, namely hybrid memory system. By utilizing our checkpointing-dedicated fine-grained address remapping mechanism, called Twins Page Mapping, we could keep checkpoint consistent without many extra writes. Twins Page Mapping maps one physical page to two PCM pages, called Base Page and Derivation Page respectively. To reduce extra writes, these pages are split into cache lines. If the n_{th} cache line of the Base Page stores a portion of checkpoint, further modification on it will be redirected to the n_{th} cache line of the Derivation page, and vice-versa. Such design reduces the metadata in that every cache line has only two possible addresses, thus every cache line needs only one bit metadata to indicate which address stores checkpoint. Our evaluation covers various workloads, and demonstrates at most 5.99x writes reduction and at most 1.88x speedup.

I. Introduction

With fast evolution of today's scientific research, future researchers require exascale systems to perform their scientific simulations. One of the most serious problems in exascale computing is the reliability issue. Specifically, the mean time between failures (MTBF) will become very short in the exascale systems because of extremely large number of cores to use in parallel. The replica execution mechanism [**?**], [**?**], [**?**], [**?**] is unqualified for the exascale fault tolerance, because it needs to launch extra processes/ranks to process the same work as the original processes, such that the total number of cores/process will double or triple. By contrast, checkpoint/restart mechanism protects the execution by saving the runtime memory periodically, avoiding the waste of the resources. Checkpoint overhead, however, could be too huge to neglect. Specifically, high frequency checkpointing may cause over 50% performance degradation [**?**] or nearly 80% of the total I/O traffic [**?**] in traditional HDD-based High Performance Computing (HPC) systems.

Memory-based Checkpointing (storing checkpoints in memory instead of hard disks) has been proposed [**?**], [**?**] for years, in order to address the huge checkpoint overhead issue. The advantage of memory-based checkpointing method is not only the low access latency and byte-addressability supplied by DRAM, but little data copy time to be incurred because the working memory just needs to be marked as checkpointing memory upon the checkpointing request.

To mitigate the volatile issue of DRAM, non-volatile memory (NVM), such as phase-change memory (PCM), has gained a lot of attention for next-generation memory system, since NVM can achieve a comparable reading performance as DRAM. Its high writing latency and limited retainability, however, still impede the performance significantly. As such, hybrid memory system with both DRAM and PCM is proposed to overcome these drawbacks.

In a hybrid memory system, extra writes could be a very severe problem because of the low PCM writing performance and the logging design or copy-on-write operations. Memory-based checkpointing shares data between working memory and checkpointing memory, such that any modification on working memory will corrupt its consistency with the checkpointing memory. To this end, traditional memory-based checkpointing adopts logging or copy-on-write to reserve checkpointing memory, introducing extra writes inevitably. In particular, logging incurs 1x extra writes because it needs to write both new and old data, and copy-on-write incurs variable extra writes as it has to write a whole page even partial data is unmodified. Although existing research [**?**], [**?**], [**?**] has leveraged PCM to perform the checkpointing, they still suffer from unexpected double-space or double-writes because they treat the PCM as storage device instead of memory (storage-based checkpointing cannot share data between working memory and checkpointing memory).

In this paper, we propose a novel memory-based checkpointing mechanism, wherein the checkpointing overhead can be minimized by leveraging the features of hybrid memory system. The key contributions are listed as follows:

1) We propose a checkpointing-dedicated fine-grained address remapping mechanism, namely *Twins Page Mapping*, to resolve the checkpointing consistency issue, with much less overhead than logging and copy-on-write. Twins Page Mapping is able to identify the address of working memory and checkpoint, and translate memory request to the address of working memory in the granularity of cache line. Specifically, Twins Page Mapping maps one physical page to two PCM pages, called Base Page and Derivation Page respectively. To reduce extra writes, these pages are split into cache lines. If the n_{th} cache line of the Base Page stores a portion of checkpointing memory, further modification on it will be redirected to the n_{th} cache line of the Derivation page, and vice-versa. Such a design can effectively avoid corrupting checkpoint, and also reduce the metadata in that every cache line has only two possible addresses. That is, every cache line needs only one bit metadata to indicate which address stores the checkpointing memory.
2) Coordinating with hybrid memory system, we also propose two optimization strategies that could reduce address translation count and the size of metadata stored in memory controller respectively. Besides that, a hardware-based Derivation Page management mechanism is proposed to improve efficiency and constrain the storage overhead.
3) We implement our checkpointing system with common used simulation tools and evaluate its performance under various workloads. The evaluation demonstrates at most 5.99x writes reduction and at most 1.88x speedup.

The rest of this paper is organized as follows. Section II elaborates the motivation of our work. Section III describes the design of our checkpointing system and key mechanism. Section IV introduces implementation details. Section V presents the evaluation and results, followed by a conclusion in Section VI.

II. Background and Design Motivation

In this section, we analyze the pros and cons of the traditional memory-based checkpointing technology and PCM-based checkpointing technology.

Memory-based checkpointing [] effectively mitigates the heavy data copying issue during checkpointing, while it suffers from the volatile issue of DRAM. Although this issue can be resolved by adopting NVM in a hybrid memory system, checkpointing consistency problem will be further introduced in turn, as shown in Figure 1. Such a problem will be rather more critical because NVM works very inefficiently on the extra writes incurred.

Memory-based checkpointing marks working memory as checkpointing memory instead of copying them to the checkpoint storage. Thus, after setting a checkpoint, there will

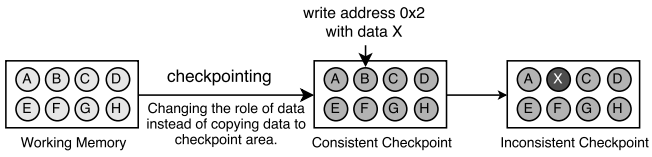


Fig. 1: Consistency Problem of Memory-based Checkpointing. Memory-based Checkpointing only changes the role of working memory to checkpoint without copy. Thus checkpoint and working memory share the same data. The modification of working memory also means the modification of checkpoint. Such modification corrupts the consistency of checkpoint and makes it can not be used for recovery.

be only one copy of data in memory, which has two roles - working memory and checkpointing memory, meanwhile. Later modifications on working memory will corrupt checkpointing memory and leave it to inconsistent state, such that the checkpointing data is no longer available to use.

Existing solutions that deal with the checkpointing consistency issue include logging mechanism and copy-on-write.

Logging mechanism has two modes, *redo logging* and *undo logging*, which suffer from the same problems, so we explain it by using only undo logging mode without loss of generality. The undo logging mechanism copies old data to a logging area before storing them, which may incur extra writing cost because of the two steps involved: the first step is copying old data to logging area (one write) and the second step writes new data to original address (one more write). Note that every logging entry also requires to write metadata (such as data address), so the undo logging mechanism actually incurs over 1x extra writes when it needs to save data.

Copy-on-Write is an address remapping mechanism, which modifies page table entry and maps the linear address of old area to a newly allocated area. The new area is generally a 4KB or 4MB page, and copy-on-write has to write the whole page even if only a few of data are actually changed in the page. We conducted an experiment to characterize the extra writes that copy-on-write may incur, with the size of cache line (more detailed experimental setting is described in Section V-A). The extra writes are measured using the number of unmodified cache lines between two consecutive checkpoints. Figure 2 clearly confirms that Page level copy-on-write may incur massive extra writes. Specifically, the average extra write ratio is up to 51.3%, and there are even four applications whose extra write ratios are greater than 90%. There are only 5 applications (from among a total of 28 applications) with 10% extra write ratios. Obviously, reducing the granularity of copy-on-write from page to cache line will reduce extra writes. However, a general-purpose fine-grained copy-on-write may cause unacceptable memory management overheads, which make it unfeasible in turn. As such, a new technology is required to resolve checkpoint consistency with less writes.

In addition to the above-mentioned memory-based checkpointing mechanism with potential checkpointing consistency

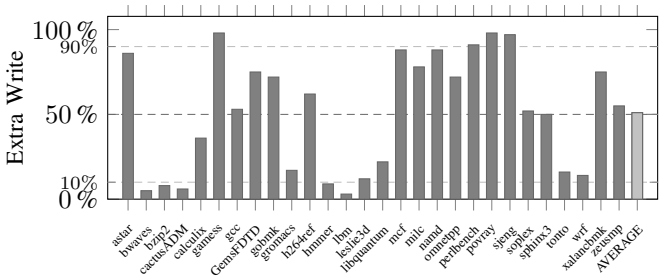


Fig. 2: The Percentage of Extra Writes of Page Level Copy-on-Write

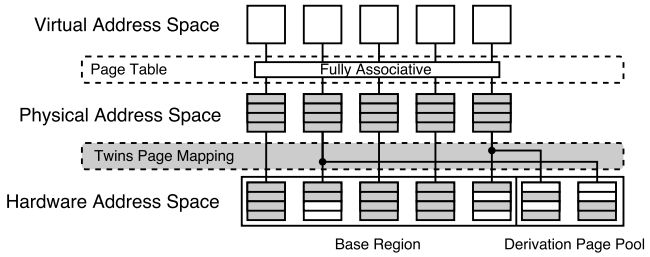


Fig. 3: Twins Page Mapping

issue, many researchers [**?**], [**?**], [**?**] utilize PCM as the checkpointing device, in order to reach a tradeoff between the higher checkpointing performance and checkpointing consistency. Although the system can access PCM directly through memory bus and PCM supplies byte-addressability, the memory used by applications cannot be dynamically allocated from PCM, such that PCM can only serve as a storage device from the perspective of operating system. This is why most existing PCM-based checkpointing mechanisms actually adopt the traditional storage-based checkpointing methods instead.

Unlike memory-based checkpointing, storage-based checkpointing does not share working memory and checkpointing memory, because of the volatile issue of traditional DRAM. As for a hybrid memory system, however, such a storage-based checkpointing design cannot make full use of the non-volatile feature of memory. What is even worse is that the isolation of working memory and checkpointing memory may result in an instant double size of the working memory in that a checkpoint generated during the execution is actually a snapshot of the working memory. This will definitely introduce a very huge burden on the memory capacity especially for the next-generation exascale applications with extremely large volume of data to process. On the other hand, redundant data incurs redundant writing cost, especially for the hybrid memory system because PCM suffers from poor data writing performance. Therefore, it is necessary to tailor a new checkpointing solution for the hybrid memory system, in order to maximize its checkpointing performance.

III. System Design

This section presents the overview of our checkpointing system first. Then we introduce Twins Page Mapping, which is able to resolve the checkpoint consistency problem with low overhead.

A. Overview

The goal of our checkpointing is to tolerate failures of hybrid memory system efficiently. We focus on the optimization of checkpointing, and adopt common methods for failure detection and recovery (cite). Theoretically, our checkpointing system can handle different failure models if it collaborates with appropriate failure detection technology. Hence we suppose a fail-stop model in this paper for the sake of simplicity.

Hybrid memory system has two typical architectures: 1) DRAM acts as the cache of PCM, 2) DRAM and PCM share address space. Using DRAM as the cache of PCM can mitigate the performance degradation caused by PCM's relative low performance, and latest work such as [**?**], [**?**], [**?**], [**?**], [**?**], [**?**] chooses this architecture as their target system, too. This architecture seems more promising so this paper focuses on it. In this architecture, all memory access requests are checked whether it hits DRAM cache first. If hits, DRAM will service theses requests. Otherwise, the request page will be read from PCM.

We split the application execution into multiple intervals. Each interval is composed by two periods. One is application execution period, and the other is checkpointing period. At the end of each interval, a checkpoint is generated.

Checkpointing period is to gather all checkpoint related information and construct a checkpoint. During this period, application will be suspended until checkpoint is established. Prior works propose to pre-copy checkpoint during execution period [**?**], [**?**], so that checkpointing could overlaps with execution partially and application will suspend for a shorter time. Such optimization also works for our checkpointing system and can be ported in easily. However, this paper tries to focus on using Twins Page Mapping to overcome the checkpoint consistency problem, thus we use a simple model without pre-copy optimization here.

B. Twins Page Mapping

Twins Page Mapping is a fine-grained address remapping mechanism aims to resolve the checkpoint consistency problem. It has a paired page to store It is a hardware mechanism that targets to translate physical address to hardware address. Physical address is the address space which can be observed by operating system, while hardware address is independent for different mediums. For example, if a PCM page is cached in DRAM, these two pages share same physical address, but their hardware addresses vary.

We first introduce two observations, which inspire Twins Page Mapping. 1) Fine-grained address remapping mechanism can resolve checkpointing consistency problem with little extra writes. As the analysis in Section **??**, logging has at least 1x extra writes. These extra writes have stable count and are hard to avoid. In contrast, the extra writes caused by copy-on-write is variable. Figure 2 shows that averagely 51.3% of the writes

are extra. In other words, page-level copy-on-write incurs about 1x more extra writes than cache line-level copy-on-write on average. So reduce the granularity of copy-on-write can reduce extra writes. Since the granularity of copy-on-write is determined by address remapping mechanism, we conclude that fine-grained address remapping mechanism could help reducing extra writes. 2) For fault tolerance, older checkpoints can be dropped when a new checkpoint is established[**?**]. Checkpoint used for fault tolerance is to recover application to latest executable state after failure happens. So we only need to keep two versions of data in memory, namely latest checkpoint and working memory.

The first observation inspires us to design a fine-grained address remapping mechanism. General-purpose fine-grained address remapping mechanism incurs large overhead because it needs to be able to map physical address to any hardware addresses. However, the second observation inspires us that a checkpointing-dedicated address remapping mechanism only needs to map a physical address to two specific hardware addresses, one of which stores checkpoint and the other one stores working memory. This gives us a chance to design a feasible fine-grained address remapping mechanism.

Based on these two observations, we propose a checkpointing-dedicated address remapping mechanism, called Twins Page Mapping. Twins Page Mapping maps one physical page with two hardware pages. It restricts that the n_{th} cache line of physical page can only be mapped to the n_{th} cache line of these two hardware pages. So every cache line of physical page only needs one bit to indicate which hardware page should it be mapped to. The metadata required by Twins Page Mapping is much smaller than general-purpose fine-grained address remapping mechanism and makes it more feasible.

Figure 4 presents an example of using Twins Page Mapping to keep checkpoint consistent. To explain the behaviors of Twins Page Mapping in different situations, we select four points in time from the first four consecutive periods respectively. We assume that a page is composed by four cache lines in this example. 1) The first one is arbitrary time during initial execution period. During initial execution, every physical page maps with one hardware page, which is called *Base Page*. The physical address of physical page should be the same as the hardware address of Base Page. At this time, Base Page stores working memory and services memory request as normal memory system. 2) The second point is the end of first checkpointing. All data is marked as checkpoint. So they have two roles now, namely checkpoint and working memory. 3) The third point is some time after application resumes execution. During execution, every read request is intercepted and redirected to working memory. Write request is also intercepted, but it needs more attention from Twins Page Mapping. Twins Page Mapping first checks whether this physical page has been mapped with two hardware pages. If not, Twin Page Mapping will allocate a new hardware page, which is called *Derivation Page*, and establishes the mapping relation. Then Twins Page Mapping translates physical address

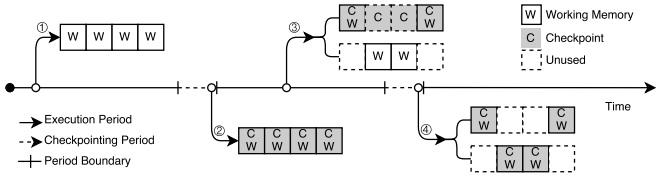


Fig. 4: An Example of Using Twins Page Mapping to Keep Checkpoint Consistent. We suppose a page with four cache lines. ① During initial execution period, all data is working memory. ② At the end of first checkpointing period, all data becomes checkpoint. ③ In the period of second execution, any write to checkpoint will trigger the allocation of Derivation Page. Write request will be redirected to corresponding cache line which does not stores checkpoint. Then such checkpoint data will have only one role of checkpoint, and the corresponding cache lines will be given a role of working memory. ④ When second checkpointing period ends, working memory becomes checkpoint and other data is dropped.

in terms of the role of data. Suppose a request tries to write the n_{th} cache line of physical Page. Twins Page Mapping finds that the n_{th} cache line of Base Page has the role of checkpoint, so it translates the address to the n_{th} cache line of Derivation Page. Thus checkpoint keeps unmodified. The role of data also changes. Before such write, n_{th} cache line of Base Page has two roles of checkpoint and working memory. But after write, n_{th} cache line of Base Page only has one role of checkpoint, while the n_{th} cache line of Derivation Page will be given a role of working memory. 4) The fourth point is the end of second checkpointing. All data only has one role of checkpoint is part of the checkpoint generated by last interval. So we drop them and mark them as unused. Except the data which is unused, rest data has the role of working memory. They are part of the checkpoint generated by this interval, so they will be given the role of checkpoint, too.

C. Coordinating with Hybrid Memory System

1) Avoid Intercepting All Memory Requests: Intercepting all memory requests to check or even translate address could potentially slow down system, especially for memory-intensive applications. Hybrid memory system helps to mitigate this problem. Coordinating with hybrid memory system, Twins Page Mapping only intercepts the memory requests happen on PCM, that is memory requests caused by cache miss and checkpointing. Because the high performance of hybrid memory system has been proved by prior works, we could expect a relative low probability of cache miss. So most of memory requests will hit DRAM and be serviced by DRAM directly without performance lost.

2) Mitigating Metadata Overhead: Twins Page Mapping restricts that every cache line can only be mapped to two candidates, which leads much less metadata than general-purpose address remapping mechanisms. However, as we will present in Section IV-A, every page still needs a relative large metadata. In our evaluation, 1.5MB of metadata is required.

Although DRAM and PCM can afford such storage overhead easily, it is still a burden for memory controller. To address this problem, we propose to select partial metadata and store them in memory controller.

Memory controller version of metadata contains essential data for two things. First, like full version of metadata, checking whether requested address has Derivation Page. Second, indicating the position of only the first several cache lines of page.

To cooperate with this shrinking metadata, we also modify the processes of flushing DRAM page and caching PCM page. To introduce the impact of our modified processes, we first present a simplified ideal page access process model in Figure 5a. If memory controller is large enough for all metadata, we could read metadata from memory controller, and then following the indication of metadata to read cache lines from DRAM or PCM. The whole latency of read multiple cache lines generally is not the sum of each cache line's read latency. Further cache line access process could be accelerated by several mechanisms, such as bank-level or rank-level parallel, accessing an opened row etc. To keep our page access process model simple, we mark all these mechanisms as parallel.

Flushing DRAM page in our system involves three steps. First, accessing memory controller to check whether we need more metadata. If the requested address has no Derivation Page, we could access DRAM normally. Otherwise we should perform second step, which is acquiring full version of metadata from DRAM. Third, reading cache lines from DRAM and writing them back to PCM. This modified process costs more time than ideal page access model because we need to read metadata from DRAM before performing real cache line read. However, this should not be a problem for our system. We explain it by analyzing two scenes of flushing DRAM page. 1) Cache miss during execution phase. As Figure **??** [not in this paper now] presents, because we use a quite small checkpointing interval, most of pages would be flushed by checkpointing instead of cache miss. Hence although flushing DRAM page incurs delay, the low frequency makes this overhead negligible. 2) Checkpointing phase. Flushing process contains two parts, namely read data from DRAM and write data to PCM. DRAM read latency is much smaller than PCM write latency, so before PCM finishes a write request, DRAM could have finished read and send this data to PCM's write queue. Figure **??** [not in this paper now. same figure as last missing one] proves this analysis. It shows that the critical path of checkpointing is composed by several DRAM read requests and lots of PCM write requests. As a result, checkpointing phase is also only delayed by several DRAM read requests.

Caching PCM page contains four steps in our system. First, we also access memory controller to get memory controller version of metadata. Second, following the indication of such metadata, we start reading the first several cache lines from PCM instantly. Third, at the time of reading cache lines from PCM, read DRAM to get a full version of metadata. Fourth, perform rest memory access when we got the full version of metadata. Because read latency of PCM is several times larger than DRAM, and DRAM and PCM could perform read operation concurrently, we could expect that the third step will finish before the second step. If so, the fourth step will start without delay. Hence our caching PCM page process has the potential to gain same performance as ideal page access model.

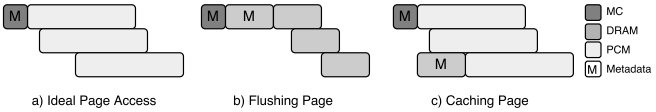


Fig. 5: Page Access Process

TABLE I: DPT columns

	Base Page Index	Checkpoint Position	Working Memory Position	Physical Page Dirty
MC	36 bits	-	2 bits	-
DRAM	36 bits	64 bits	-	64 bits
PCM	36 bits	64 bits	-	-

IV. Implementation

In this section, we propose the implementation details of our checkpointing system. At first, we introduce the metadata management of Twins Page Mapping, followed by the details of page movement between DRAM and PCM. Then Derivation Page management is presented. At last, we introduce the checkpointing process.

A. Metadata Management

Metadata of Twins Page Mapping is maintained in a table called Derivation Page Table (DPT). The columns of DPT is: 1) A 36 bits of Base Page Index which stores the higher-order bits of Base Page's physical address. 2) A 64 bits of Checkpoint Position indicating that whether checkpoint is stored in Base Page or Derivation Page. Every bit represents a cache line. 3) A 2 bits of Working Memory Position. Its usage is similar like Checkpoint Position, which is indicating whether working memory is stored in Base Page or Derivation Page. However, it only has two bits to represent the position of the first two cache lines. Its usage has been explained in Section III-C2. 4) A 64 bits of Physical Page Dirty indicating that whether the cache line of physical page has been modified since last checkpointing.

We maintain three versions of DPT and place them in memory controller, DRAM and PCM respectively. These three versions of DPT have different purposes, so the columns of them also vary, as listed in Table I. Memory controller version of DPT aims to be small so it can be hold by memory controller. It has two columns, namely Base Page Index, and Working Memory Position. For DRAM version of DPT, size is not the problem. It stores all columns except Working Memory Position and provides efficient access. PCM version of DPT is for recovery. Thus it only takes essential columns, i.e. Base Page Index and Checkpoint Position.

To reduce PCM writes, we also adopt a common optimization that only write dirty cache lines [**?**], [**?**]. This optimization

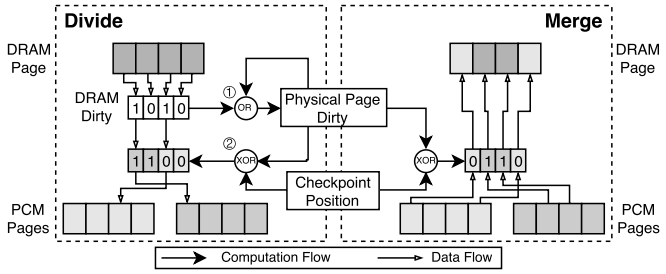


Fig. 6: Divide and Merge Processes of Twins Page Mapping

requires a 64 bits of tag, called DRAM Page Dirty, for every dram page. This tag indicates whether the cache lines of dram page have been modified since they were read into DRAM. Because such tag can be used for different purposes and is so common (cite), we expect it could become the standard of future hybrid memory system. Hence it is excluded from overhead caused by our design.

B. Page Movement Between DRAM and PCM

There are two possible ways to trigger page movement between DRAM and PCM. The first one is cache miss. A typical cache replacement process contains three steps. 1) Choosing victim page according to cache replacement algorithm. 2) Flushing the data stored in victim page to corresponding PCM page. 3) Copying data from expected PCM address to victim page. The latter two steps are replaced with the **Divide** and **Merge** process of Twins Page Mapping respectively. The second one is checkpointing. Checkpointing needs to flush DRAM pages to PCM. This process can also be replaced by **Divide** process.

Divide will divide DRAM page by cache line and write these cache lines to corresponding PCM pages. As Figure 6 illustrates, we first update tag Physical Page Dirty by bitwise OR it with DRAM Dirty. Then we calculate a temporary tag P by Equation 1. The value of P indicates the position of working memory. Suppose the n_{th} bit of P is 0, the n_{th} cache line of physical page will be translated to the n_{th} cache line of Base Page. Otherwise it will be translated to the n_{th} cache line of Derivation Page. In addition, we only write dirty cache lines to reduce writes.

$$P = Physical\ Page\ Dirty \oplus Checkpoint\ Posisiton \quad (1)$$

Merge selects cache lines from Base Page and Derivation Page according to the indication of metadata, and then merges them to an intact page in DRAM. Figure 6 illustrates this process. Merge process is simpler than Divide. We just need to calculate a temporary tag P by Equation 1. Then we could read cache lines following the value of P and write them to DRAM. At last, we get a normal DRAM page which can be accessed directly.

Note that, the tag update of Divide and Merge process all happens on the DRAM version of DPT. We do this for two reasons. First, update DRAM version of DPT can benefit from the high performance of DRAM. Second, and also the key reason, these update helps to locate the position of working memory but has no effect for checkpoint. So persisting them to PCM makes no sense.

C. Derivation Page Management

1) Derivation Page Pool: Derivation Page Pool is the structure used for storing Derivation Pages. It is a reserved large and continuous PCM area, and is managed by hardware. We split it into multiple pages and use bitmap to record whether a page a allocated.

The size of Derivation Page Pool is important. It determines the size of DPT, so a large Derivation Page Pool aggravates the overhead of metadata. However, small Derivation Page Pool leaves a high possibility of failing to find a free Derivation Page. So we perform an experiment in Section V-C to evaluate the influence of different size of Derivation Page Pool.

2) Allocating Derivation Page: To service Derivation Page Allocation request, we first scan the bitmap of Derivation Page Pool and try to find a free page. If a free page is found, we return this page to caller. Otherwise, we would try to release a Derivation Page and return it to caller. If we fail again, we should call checkpointing to get releasable pages.

Derivation Page Allocation is requested during execution. At the end of checkpointing interval, all PCM pages are marked as write-protect. In next execution period, any write to these write-protect pages will trigger a check of mapping state. If mapping is not established, we will send a Derivation Page Allocation request to Derivation Page Pool.

Note that, the write to DRAM page, but not the write to PCM page, will trigger Derivation Page Allocation. An integrate checkpointing process make all pages clean and ready to be released. So calling checkpointing is the last resort to meet the allocation requirement of caller, and we have to ensure that we can finish an integrate checkpointing process under any situation. This requirement results that allocating Derivation Page when a write happens on PCM is not appropriate. If we fail to allocate a Derivation Page to service a PCM write, we could not expect checkpointing, which will incurs more PCM writes, can finish smoothly. In contrast, allocate Derivation Page when we write to DRAM ensures that all dirty DRAM pages have allocated Derivation Page. So write back can be processed without failure. As a result, we can processing an integrate checkpointing at any time.

3) Releasing Derivation Page: To release space, we first scan Derivation Page Pool to find a releasable page. A releasable page is such a page that 1) its Physical Page Dirty tag is cleared, which means it has not been modified during this checkpointing interval. 2) if it is cached in DRAM, corresponding DRAM page should be clean. These two condition ensures that it only has one version of data stored in memory, namely last checkpoint. So we can extract the data belonged to last checkpoint from Derivation Page and write it to corresponding cache lines of Base Page. Then Derivation Page can be freed.

If Derivation Page Pool can not find such a page, which means all pages have been modified during this checkpointing interval, Derivation Page Pool has to tell caller that Page Allocation fails. Caller should call checkpointing instantly to get some releasable pages and request for Derivation Page Allocation again.

Derivation Page Release should never be requested by caller proactively because Derivation Page is released automatically when we can not find a free page for allocation. This design keeps operating system's page management transparent for hardware, which reduces the communication between operation system and hardware, and also simplifies system.

D. Checkpointing

Checkpointing needs to persist three kinds of data, namely working memory, CPU context information and DPT. In this section, we will explain the checkpointing processes of these data respectively.

During execution, cache eviction has moved partial working memory to PCM. Thus what left for checkpointing period is flushing data stored in DRAM to PCM. To be specific, we will call Divide process for all dirty DRAM pages.

Checkpointing is triggered by application or operating system proactively. So application or operating system is responsible to determine the composition and the store address of CPU context information. To avoid failure during checkpointing, such data should be stored in different address with previous checkpoint.

Before we persistent DPT, we need to update it first. This involves three steps. First, we update Physical Page Dirty by bitwise OR it with DRAM Dirty. Second, we update Checkpoint Position by bitwise XOR it with Physical Page Dirty. Third, the new PCM version of DPT, which is composed by Physical Page Dirty and Checkpoint Position are written to PCM. Certainly, they will also be written to a different address with previous checkpoint to avoid failure during checkpointing.

Besides of persisting data, we also need to prepare for resuming application. We need to reset Physical Page Dirty and DRAM Dirty. And the DRAM version of DPT should be overwritten by corresponding columns of PCM version of DPT. Eventually, we establish a checkpoint and are ready to resume application. Checkpointing period ends.

V. Evaluation

Our evaluation tries to answer following questions:

1) How does our system perform overall?
2) What is the storage overhead and how does different storage overhead affect system performance?
3) Does our system slow down if we only store partial metadata in memory controller?

A. Experimental Methodology

1) Experimental Platform: We build our experimental environment using MARSSx86[**?**], a full system simulation tool for x86-64 architecture. A quad-core system which has out-of-order core pipeline is conducted. Every core has a frequency of 1.87 GHz and 128KB of L1 I/D cache. A 2MB of L2 cache is shared by all cores.

Memory system is consisted of DRAM and PCM. DRAM is a 256MB DDR3 device simulated by DRAMSim2[**?**], while PCM is simulated by NVMain[**?**] with a size of 8GB. They both adopt the timing parameters presented by [**?**], such as tRCD, tCL, tRP etc. HybridSim[**?**] integrates DRAM and PCM. It takes DRAM as the cache of PCM, and uses classical LRU as cache replacement algorithm.

2) Comparison: We compare with two memory-based checkpointing systems, which use logging and copy-on-write to resolve checkpointing consistency problem respectively.

- *Logging* accesses data by the granularity of cache line. It copies old data to log area first before writes new data. To gain a better performance, we adopt a optimization from [**?**] that only log for the first modification per checkpointing interval.
- *Copy-on-Write* copies data to new page and redirects further memory access requests to new page. Such process happens only at the time of cache misses. We adopt two optimization for copy-on-write. First, address remapping mechanism is maintained by memory controller. Second, we reserve a large enough space to ensure that new page allocation can always be satisfied and can be finished by hardware.

These two memory-based checkpointing systems have the same configuration as our system. Memory requests are also handled by DRAM. And if write back does not incurs logging or copy-on-write, only dirty cache lines are written back.

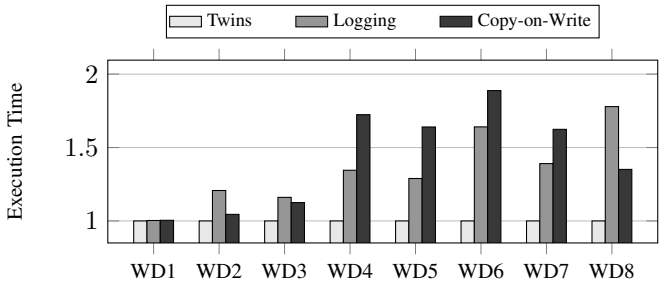
3) Workloads: To evaluate our system's performance, we carefully select applications from SPEC CPU2006[**?**] and list them in Table II. These workloads have evenly distributed L3 cache miss rate [**?**] and different footprint, so they can represent a majority of real workloads. Every workload contains four applications which start simultaneously. After a warm up phase, we start a simulation for 1.5 billion instructions and request for checkpointing every 30 million cycles.

TABLE II: Workloads

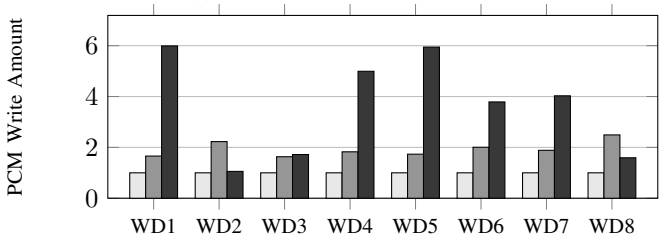
	Workload	L3 Miss Rate	FP (MB)
WD1	gamess h264ref h264ref povray	2.0%	34
WD2	bzip2 bzip2 gromacs gromacs	24.2%	1716
WD3	bzip2 gcc gromacs perlbench	40.4%	1842
WD4	gcc gcc mcf perlbench	57.3%	2471
WD5	astar gobmk mcf omnetpp	64.7%	2109
WD6	gobmk leslie3d omnetpp wrf	77.7%	986
WD7	leslie3d sjeng sphinx3 wrf	88.3%	1011
WD8	GemsFDTD lbm milc soplex	96.3%	1878

B. Overall Evaluation

Workload Execution Time. At first, we present the comparison of workload execution time in Figure 7a. All results

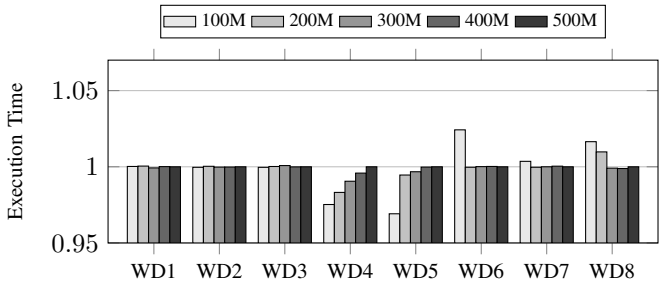


(a) Workload Execution Time

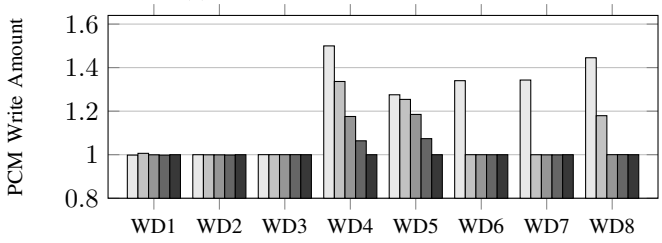


(b) PCM Write Amount

Fig. 7: Overall Performance



(a) Workload Execution Time



(b) PCM Write Amount

Fig. 8: Performance with Different Derivation Page Pool Size

are normalized by our checkpointing system, which is marked as Twins in figures. A speedup for all workloads are observed. The average speedup is 1.29x and 1.19x compared with logging and copy-on-write respectively. For specific workload, we even achieve at most 1.4x and 1.88x speedup.

PCM Write Amount. Our another concern is PCM write amount. Figure 7b illustrates how much PCM writes are requested during evaluation. All results are also normalized by our checkpointing system. As the analysis in Section **??**, writes incurred by logging is about 2x, while copy-on-write incurs various write amount for different workloads. The average PCM write amount of logging and copy-on-write is 1.89x and 4.11x on average.

The performance of Logging and Copy-on-write vary as workload changes. Logging performs better for WD1, WD3 to WD7, while copy-on-write wins for WD2 and WD8. The reason for that is the performance of copy-on-write depends on the access pattern of workloads. Copy-on-write has to copy all pages even if only a small part of it is modified. So workloads which modifies massive pages but only incurs a spot of modification for each page will perform bad. In contrast, Twins Page Mapping shows a stable performance for every workloads. Even for worst case WD1, in which Twins Page Mapping executes same time as logging and copy-on-write, we still observe a clear reduction of PCM writes.

We could find that PCM write amount reduction for different workloads does not contributes a same speedup. The reason is that the duration of checkpointing period for each workload is quite different. (data)

C. Storage Overhead

Twins Page Mapping needs to reserve a large size of PCM area as Derivation Page Pool. Such space can not be reused by other applications and should be treated as storage overhead. To know how much space should we reserve, we evaluate the performance of different size of Derivation Page Pool.

Impact on Workload Execution Time. Results presented in Figure 8a show that different size of Derivation Page Pool has slight influence on workload execution time. The smallest Derivation Page Poll, which is 100 MB in our experiments, only slows system at most 2.4%.

Impact on PCM Write Amount. Figure 8b presents a increase of PCM writes for some workloads. PCM writes increase as the size of Derivation Page Pool reduces, which conforms our expect. These extra PCM writes are produced by Derivation Page Release process. While we can not find available page for Derivation Page Allocation request, we have to find a page whose working memory and checkpoint are the same, so we can extract data from Derivation Page and write it back to Base Page. Such process incurs extra writes. For worst case, it produces 1.49x writes.

We observe that large increase of writes does not prolong execution much. This is because of two reasons. First, like we explained in Section V-B, checkpointing period is just a small part of the whole application execution, so extra PCM writes could not influence execution time too much. Second, Derivation Page Release process is requested when DRAM is modified, and it parallels with execution. Such parallel requires cold PCM to achieve better performance. Because we request for checkpointing in a high frequency, most of dirty DRAM pages would be flushed by checkpointing instead of being chosen as victim during cache miss. Our evaluate also proves that execution period only incurs negligible PCM writes. Hence PCM is cold during execution period and gives a chance for us to achieve parallel.

D. Metadata Overhead

VI. Conclusion

The conclusion goes here.

Acknowledgment

The authors would like to thank.