OPTIMIZING EMERGING MEMORY SYSTEMS FOR PERFORMANCE

A Dissertation

by

CHIH-CHIEH CHOU

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	A. L. Narasimha Reddy
Committee Members,	Paul V. Gratz
	Eun Jung Kim
	Krishna Narayanan
Head of Department,	Miroslav M. Begovic

May 2020

Major Subject: Computer Engineering

ABSTRACT

Emerging Non-Volatile Memories (NVM), such as phase-change memory (PCM), NVDIMM, and 3D XPoint, have byte-addressability and low latency, close to that of main memory, together with the non-volatility of storage devices. NVM can be treated as both memory as well as a storage device in the systems. These emerging technologies have great potential to improve the system/application performance as well as to provide scalability and reliability of applications and services. Much prior work focused on new system designs using NVM and has already shown promising performance improvements. In this work, we try to employ NVM with different directions and approaches.

First, we construct a user space library to virtualize and share NVM between multiple processes and applications. The reason to construct a user space library is because of performance, which can be significantly improved by avoiding context switch overheads from system calls. NVM has DRAM-like latency. So, accessing NVM through the kernel space as before would result in too much context switching overhead, and therefore it would squander the low-latency provided by NVM. Our library tries to access NVM mostly in user space, and only enters kernel space whenever necessary. We have shown that our novel user space library incurs less than 10% overhead while providing the properties of virtualization and sharing of NVM.

Also, recently emerging interconnect fabrics, such as Gen-Z, provide high bandwidth, together with exceptionally low latency. These concurrently emerging technologies are making possible new system architectures in the data centers including systems with Fabric-Attached Memories (FAMs). FAMs can serve to create scalable, high-bandwidth, distributed, shared, byte-addressable, and non-volatile memory pools at a rack scale.

We propose FAM-aware, checkpoint-based, post-copy live migration mechanism to improve the performance of application migration. We have implemented our prototype of mechanism in a Linux open source checkpoint tool, CRIU (Checkpoint/Restore In Userspace). According to our evaluation results, compared to the existing CRIU approach, our FAM-aware post-copy can improve the total migration time by at least 15%, the busy time by at least 33% and down time by at least 23%, and can let the migrated application perform at least 12% better during migration as well as our batching checkpoint can improve average checkpoint time by 15%.

We look at the problem of integrating low latency NVM tightly in the memory hierarchy. We consider the problem of reducing the costs of page faults for moving data between DRAM and NVM. Since NVM can provide much low latency, so, in addition to the overheads of context switches from system calls, the overheads of context switches from page faults seem too high, too. We approach this problem through a hardware, software co-design approach by extending the CPU hardware page walker and enhancing the Linux kernel. We develop a page pre-allocation mechanism to pre-allocate pages before a page fault happens, where the hardware page walker executes some required operations during the page faults, and a kernel background thread is triggered to periodically handle the post-page fault operations after page faults. We show that the critical path latency of a page fault, improved by our new hardware and kernel, can reduce to 2.1% for write and 12% for read, compared with that of existing kernel page fault exception handler. We can improve the execution time for some benchmarks by about 5.6%.

DEDICATION

To my mother in heaven

ACKNOWLEDGMENTS

First of all, I would like to thank Dr. Reddy for his guidance and support throughout my entire years at Texas A&M University. I truly appreciate his patience when I experienced some hard times for the first few years. I admire his solid knowledge about computer and memory architecture. Without his valuable suggestions and support, there is no way I can finish this degree.

I also thank Dr. Gratz, who is like a co-advisor to me, for his mentorship, teaching, and efforts of carefully revising my writing.

I thank Dr. Kim and Dr. Narayanan for their valuable comments and feedback to improve the content and quality of this work.

I appreciate Dr. Jung. During his time as a Postdoc at Texas A&M University, he shared his knowledge and experience of Linux kernel architecture wholeheartedly. Without him, I might still struggle for my first project now.

I want to thank Mr. Voigt for his feedback and suggestions and generous support from Hewlett Packard Enterprise.

I also want to thank Mian Qin, Ping Wang, and Fei Wen for their accompanying, as officemates, in the long path of pursing this degree. I truly hope they will all be able to finish their degrees successfully in the near future, too.

Finally, I want to thank my family for their continuous support and sacrifices to let me finish this degree. I hope, from now on, I can fulfill their dreams and provide them a better life.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supervised by a dissertation committee consisting of Dr. A. L. Narasimha Reddy, Dr. Paul V. Gratz, and Dr. Krishna Narayanan of the Department of Electrical and Computer Engineering and Dr. Eun Jung Kim of the Department of Computer Science and Engineering.

The modification of Gem5 emulator in Chapter 4 is implemented by Chandrahas Tirumalasetty, a PhD student of Dr. Reddy.

All other work conducted for the dissertation was completed by the student independently.

Funding Sources

The entire work was supported and funded by National Science Foundation and Hewlett Packard Enterprise through grant I/UCRC-1439722.

NOMENCLATURE

OS	Operating System
DBMS	Database Management System
NVM	Non-Volatile Memory
FAM	Fabric-Attached Memory
SSD	Solid State Drive
HDD	Hard Disk Drive
VM	Virtual Machine
CRIU	Checkpoint/Restore in User Space
YCSB	Yahoo! Cloud Serving Benchmark
TLB	Translation Lookaside Buffer
PTE	Page Table Entry

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

Emerging non-volatile memory (NVM) technologies, such as phase-change memory (PCM) [1], NV-DIMM [2], STT-RAM [3], and 3D-XPoint [4], have dramatically shaken up future system designs [5, 6, 7, 8, 9, 10]. In particular, these NVM technologies promise not only much faster access times than existing SSDs, within an order of magnitude of DRAM, but they also are "byte" addressable and will be placed directly on the memory buses. As a result, these NVM technologies could be used to replace the existing permanent storage devices or even volatile memory (single level system).

Similarly, recently emerging interconnect fabrics, such as Gen-Z [11], provide high bandwidth, together with exceptionally low latency. These concurrently emerging technologies are making possible new system architectures in the data centers including systems with Fabric-Attached Memories (FAMs) [12]. FAMs can serve to create scalable, high-bandwidth, distributed, shared, byte-addressable, and non-volatile memory pools at a rack scale, opening up new usage models and opportunities. These emerging technologies have great potential to improve the system and application performance as well as to provide scalability and reliability of applications as well as service.

In this dissertation, we would like to optimize the emerging memory systems for high performance; meaning that we will mainly focus on the access latency and system overall performance when employing NVM and FAM. To do so, we designed and implemented three novel system architectures from user space, application level, and kernel space respectively, to improve the emerging memory systems.

In Chapter 2, we build a user space library, vNVML, for applications to access NVM. Although NVM has appeared in the market for a while, in research community we have not reached an agreement on how to "correctly" use NVM. To date, there have been some significant works in this domain. Some prior works [13, 14, 15, 8, 10, 7] engineer novel file systems suitable for exploiting NVM. Other prior works [16, 9] employ NVM as the only media in their (single level) system and carefully design their data store manipulation mechanism to directly access data structures from NVM. Their aims are to maximize performance by eliminating unnecessary data movement between volatile memory and persistent storage devices. These prior schemes, however, currently present no way to virtualize and share persistent NVM among multiple applications and users.

Instead, vNVML provides the transaction-like interface for applications to access NVM efficiently while also offering the virtualization and shareability of NVM. Our method, the user space library, could provide better performance, compared to file system interface, because vNVML tries to execute everything in the user space as much as possible, and only goes down to kernel land if necessary. Thus, we can significantly reduce the context switching overhead, which the accessing NVM through file system interface must suffer and cannot avoid.

In Chapter 3, we explore the design area of application migration when employing FAM, a shared NVM pool in a rack scale environment. Migration is an important technique, which can redistribute the workload from a heavily loaded node to some other nodes with lighter loading to increase the overall system performance. Traditionally, migration can be non-live (with the shortest busy time) and live one (with the shortest down time). Employing FAM, we could redesign and reimplement post-copy live migration to be a migration with the shortest busy time and down time. Through evaluation, we prove that our new checkpoint-based post-copy migration can provide better metrics for migration, compared to the existing, networking based post-copy migrations.

In Chapter 4, we want to build the POE, a Page fault handling Offload Engine, when accessing DRAM and NVM. From Chapter 2, we have learned that the overhead of context switching is too high, compared to the low-latency provided by NVM. As the bus-attached NVM becomes available in the market, the existing memory system must be reviewed and be redesigned to provide better accessing latency to fully gain the benefits brought from NVM. We propose to co-design hardware and software to reduce the page handling overheads through enhanced page walker hardware and enhanced Linux kernel for handling page faults. This idea is similar to that some of the network processing has been offloaded through hardware to reduce the packet processing overheads.

What we do is simply to let the kernel (software) execute the required operations before and after the page faults, and only the hardware page walker executes some mandatory operations during a page fault. We could prove that our design can significantly reduce overhead of some types of page faults, and show that the page fault critical path latency can reduce to 2.1% for write and 12% for read of existing software exception handling. In addition, POE can improve the execution time of some benchmarks by about 5.6%.

2. VIRTUALIZE AND SHARE NON-VOLATILE MEMORY IN THE USER SPACE[*]

2.1 Introduction

Emerging non-volatile memory (NVM) technologies, such as phase-change memory (PCM) [1], NV-DIMM [2], and 3D-XPoint [4], will dramatically shake up future system designs [5, 6, 7, 8, 9]. In particular, not only do these NVM technologies promise much faster access times than existing NAND-based SSDs, within an order of magnitude of DRAM, but they also are "byte" addressable and will be placed directly on the memory buses. Furthermore, these NVM technologies could be used to replace existing permanent storage devices or even volatile memory (i.e. single level system).

To date there have been some significant works in this domain. Some prior works, such as [13, 14, 15, 8, 10, 7], engineer novel file systems tailored for exploiting NVM. Other prior works, such as [16, 9], employ NVM as the only media in their (single level) system and carefully design their data store manipulation mechanism to directly access some data structures stored in NVM. Their aim is to maximize performance by eliminating unnecessary data movement between volatile memory and persistent storage devices. These prior schemes, however, currently present no way to virtualize and share persistent NVM among multiple applications and users.

Traditionally, there are two common ways for applications to access data content in storage devices. One is through the file system `read/write` interface, the other is via the memory mapped file (`mmap`) interface. The cost of system calls incurred by accessing through the file system, however, would squander the low-latency as well as performance provided by NVM. Thus, to attain the maximum gain from NVM, in this paper, we focus on memory mapped file access form, which is also the recommended form by SNIA [17].

[*] ©2020 Springer Nature. Reprinted, with permission, from Chih Chieh Chou, Jaemin Jung, A. L. Narasimha Reddy, Paul V. Gratz, and Doug Voigt, "Virtualize and Share Non-Volatile Memories in User Space," *CCF Transactions on High Performance Computing*, vol. 2, pp. 16-35, Feb. 2020. ©2019 IEEE. Part of the data reported in this chapter is reprinted, with permission, from Chih Chieh Chou, Jaemin Jung, A. L. Narasimha Reddy, Paul V. Gratz, and Doug Voigt, "vNVML: An Efficient User Space Library for Virtualizing and Sharing Non-Volatile Memories," in *35th Symposium on Mass Storage Systems and Technologies (MSST)*, 103-115, May, 2019.

Currently, when files on storage devices are `mmapped` to volatile memory (DRAM), the POSIX interface can support *shared* `mmap`; that is, the same region of files can be shared/accessed between multiple processes. Also, thanks to the swapping mechanism provided by virtual memory, which writes dirty pages to the backend storage devices (swap space) and therefore produces clean pages for the future use, the impression of physical available DRAM is extended. These two attractive properties of existing volatile virtual memory, however, are not supported in the prior work for NVM.

This paper considers this problem of virtualizing and sharing byte-addressable NVM across multiple applications. Here we introduce "vNVML", an efficient library for virtualizing and sharing NVM in user land. What we mean by "sharing NVM" here is that not only can the same physical NVM pages be reallocated and reused across users, but the data content on NVM pages can also be seen/accessed by applications concurrently, exactly like *shared* `mmap` access form of virtual memory.

One of the main aims of vNVML is to provide the impression of larger NVM availability to applications, much like virtual memory allowing the use of more main memory than the actual physical memory in the machine. In order to virtualize NVM in this manner, this paper examines extending a smaller amount of byte-addressable NVM with larger, traditional storage devices[1].

Further, we examine mechanisms to safely leverage DRAM as cache to improve the performance of persistent memory access. There are certain advantages in employing DRAM even when the applications access virtual NVM. First, DRAM may have better performance (lower latency) than most types of NVM, except for NV-DIMM. Second, DRAM may alleviate lifetime issues of NVM in read-intensive workloads, since many NVM technologies have write endurance limits. In our design, some reads can be served by reading pages from storage devices to DRAM, bypassing the NVM entirely. This might be a better design choice compared to simply employing NVM as both read and write cache in terms of reducing the number of NVM write accesses.

We design and implement vNVML with the hope that programmers could access (virtual)

[1] Note: while our approach can be applied with magnetic disks as the backing stores, here we limit ourselves to SSDs.

NVM with a similar interface as for existing memory mapped files. That is, after a file on the storage device is `mmapped` as a virtual NVM region, a pointer to this region is returned. This pointer can be directly used in the programs as a typical `mmapped` pointer to virtual memory. However, when NVM is exploited as permanent storage by applications, the durability and ordering of writes must be assured. Write ordering as required by byte-addressable NVM has been discussed almost in every prior work [15, 16, 18, 19, 9, 20]. Many approaches have been proposed to address the write ordering problem within persistent memories, including hardware capacitors to ensure eviction order of all data from volatile memory to NVM [15], epoch-based writes [15, 18], transaction-like semantics [20, 19, 18, 21], versioning [16], and special data stores and algorithms designed for single level NVM [16, 9]. Here vNVML proposes to use transaction-like semantics to guarantee the write ordering, atomicity, and durability of NVM accessing.

To sum up, vNVML employs DRAM as cache, NVM as log buffer and write cache, and the backing storage device as the final destination of writes. From our evaluations, vNVML incurs less than 10% throughput overhead, if the cache system of vNVML can absorb the write traffic, compared to directly accessing NVM without the atomicity and write ordering guarantees.

The contributions of this Chapter are as follows:

- Propose a transactional interface for virtualizing and sharing persistent non-volatile memory.
- An implementation of this interface in our virtualized NVM Library, vNVML.
- This implementation leverages caching in DRAM, coupled with write logging and caching in NVM and lazy writeback to the backing storage to provide a high performance, virtualized, and shareable NVM to applications.
- We evaluate this proposed vNVML under not only synthetic but also realistic (YCSB+MongoDB) workloads and show that vNVML is competitive with prior techniques which do not support virtualization and sharing of NVM.

The remainder of this Chapter is organized as follows. Section 2.2 describes background and prior work in this area. Section 2.3 presents a design overview and discusses the design decisions of vNVML. Section 2.4 explains the implementation of vNVML in detail. Section 2.5 presents our results of evaluation of vNVML and section 2.6 concludes.

2.2 Background

Much of the early work to-date incorporating NVM in systems assumes basic hardware changes. New memory controllers are proposed [22, 23, 24]. Kiln [25] proposes a victim cache for buffering and Atom [26] deploys a hardware logging approach to eliminate software logging overhead. BPFS [15] develops a new "epoch" for write ordering. While these approaches show promise, they require significant hardware redesign, which may take several years to be reflected in commercial hardware.

In the more near term, we might prefer to the solutions requiring little or no change to the basic processor caching and memory management hardware because memory bus-attached NVM DIMMs have become available. For example, recently Intel has released its Optane DC Persistent Memory DIMM [4]. Near-term systems will incorporate this type of NVM attached directly on the memory bus, where it will be accessible via the system's physical address space. These system architectures argue for a pure-system software approach to management.

Existing work to-date looking at system software approaches to managing this (memory bus-attached) form of NVM primarily focuses on constructing new file systems to handle the underlying NVM, such as SCMFS [13], NVMFS [14], BPFS [15], PMFS [8], NOVA [10], STRATA [7], FRASH [27], and Aerie [28]. Some of those works have considered building file systems across multiple types of NVM and storage technologies. These include NVMFS [14] (NVM and SSD) and Strata [7] (DRAM, NVM, SSD, and HDD). The design concept of Strata is close to our vNVML; both of them contain DRAM and NVM as caches. However, the DRAM of Strata only caches the pages read from SSDs and HDDs and all updates would go directly to NVM only. Therefore, Strata needs to search for the up-to-date data locations. In addition, Strata does not support memory mapped files access form, which is the primary form used by our target applications.

Accessing NVM through file system APIs has fundamental drawbacks. First, accessing NVM via the file system interface is not suitable for random, small accesses to NVM due to the high (context switch) overheads incurred by system calls compared with the relative low-latency that NVM offers. Next, the file system naturally cannot support concurrent accesses of the same file by different threads or processes, even though those threads or processes access different objects/pages in the same file. For example, to access a certain location of a file, users must acquire a file lock first, and then `seek` to the location and `read/write` from and to that location of file. After `read/write` command is completed, the file lock can be released. Another drawback is that such (file system) software approaches require users and applications to deploy dedicated file systems.

Some works, such as NV-Tree [9] and CDDS-Tree [16], consider replacing DRAM and storage devices entirely with NVM to construct a single level, NVM-only system; their idea is to manipulate all data structure operations directly on NVM and therefore eliminate all data movements between DRAM and storage devices. Such approaches can improve performance significantly; however, they restrict themselves to only some specific data structures and cannot be easily applied to general memory access. Also, they totally ignore the lower latency offered by DRAM and do to further explore the potential performance improvements. For example, their approaches might be not suitable for the read-intensive, cache friendly workloads.

SPAN [29] proposes some new swapping enhancements in the operating system (OS) kernel to exploit NVM as extended system memory. Its concept is similar to the memory mode supported by Intel's Optane DC Persistent Memory [30]. NV-Heaps [18] provides some useful features, such as type-safe pointers and garbage collection, but it requires programmers to use its specific object framework and hardware must support epoch as BPFS [15] does.

Other approaches try to create user space libraries [19, 31, 20, 21]. PMDK [31] and KAMINO [21] employ only NVM and utilize undo logging to support in-place data updates. SoftWrAP [20] combines NVM and DRAM and employs DRAM for write/read accesses and NVM for redo logging. However, during the normal operations, the data content is "retired" from DRAM to the final locations in NVM, and the logs in NVM are referenced only after system failures. The most closely related work to our vNVML presented here is Mnemosyne [19], which also uses

Table 2.1: Summary of prior software approaches

Categories	Approaches
File system	SCMFS, NVMFS, BPFS, PMFS, NOVA, STRATA, FRASH, Aerie
Single level system	CDDS-Tree, NV-Tree
Persistent object system	NV-heaps
User space library	mnemosyne, PMDK, SoftWrAP, KAMINO

redo logging. While Mnemosyne provides both persistent region and persistent heap allocation methods, vNVML does not support heap style allocation. However, there are some fundamental differences between these two approaches. Mnemosyne achieves NVM virtualization by swapping, which is controlled entirely by the kernel and therefore suffers from context switching overhead in the page fault critical path; vNVML, on the other hand, always writes the dirty pages back to storage devices by a background thread. Further, Mnemosyne does not employ DRAM as read cache, so it requires an extensive search to find up-to-date data. Also, Mnemosyne cannot support true sharing of NVM between processes. Table 2.1 summaries above-mentioned software approaches.

In this work, we propose a system software-based, user space management approach, which makes NVM available directly to user applications, without the block-level semantics of traditional file systems. Furthermore, our work also provides write ordering, atomicity, and endurance guarantees while offering a larger than physical available NVM space to the applications, and allows them to safely share the virtual NVM regions while maintaining performance goals by leveraging caching in DRAM.

2.3 Design overview

In this section we describe our design and provide an overview of the decisions made in the design of the virtual NVM Library (vNVML), a user space library for virtualizing and sharing NVM. Our design decisions are guided by the following four observations.

First, *persistent memory is typically allocated and dedicated to an application*. For example, when file system writes data to a location in persistent memory, that location cannot be reused or reallocated by another application; otherwise, the data content of that location is corrupted. If NVM is similarly allocated and used, then NVM cannot be easily shared[2] across multiple applications if NVM is the only persistent storage device in the systems. In data centers, with dynamic workloads, there is a strong desire to share available resources across many applications. It is essential that we provide mechanisms to share precious resources like NVM across many applications.

Second, *simply replacing traditional storage devices, such as solid-state drives (SSDs) or hard drives (HDDs), with NVM is not good enough*. Although by doing so, all existing applications can benefit immediately from performance improvements provided by NVM without any modifications required; however, this ignores the byte-addressability of NVM, and requires accessing NVM in units of blocks, resulting in a suboptimal approach.

Third, *the access latency of NVM is very close to that of DRAM and is much faster than that of storage devices*. When storage devices are slow (for example magnetic disks), the overheads paid by accessing through system calls (or context switching) from file system APIs may not be a significant part of the entire access latencies. However, as devices get faster, like NVM whose latency is within an order of magnitude of DRAM, these system call overheads become much more significant and hence must be avoided. For example, Intel Storage Performance Development Kit (SPDK) [32] implements the whole NVMe device driver in the user space and thus improves the accessing Ultra-Low-Latency (ULL) SSDs, such as Intel NVM-based 3D-Xpoint or Samsung Z-NAND [33], performance significantly.

Finally, while it is possible (and even desirable) to continue running existing or older software on new hardware, *software may have to be redesigned/rewritten to attain the most of the hardware*. This can take the forms of new file systems, new data stores [16, 9] or new libraries [19, 34, 31, 20, 21]. In this paper, we take the approach of developing a user space library interface to NVM to achieve our goals.

Based on above four observations, we designed the vNVML user space library, integrating

[2] Note: the “share” here means the same physical NVM pages can be reallocated/reused by multiple applications.

DRAM, NVM, and backend storage devices to construct the abstraction of virtualized NVM. Like virtual memory, adopted almost universally in the modern computer systems, the main idea of our virtual NVM is to provide the impression that applications and users can treat (virtual) NVM contained in their system as large as the capacity of the storage devices in the system and as fast as the speed of NVM (or even DRAM).

Usually, applications have two means to access the data content stored on storage devices; one is through file system `read/write` commands, the other is through memory mapped files (i.e. `mmap`). Because traditional disks are very slow, accessing data content in disks through file system commands, which in turn trigger some system calls, may not incur too much performance penalty compared to the long access latency of the disks themselves. However, this is not the case when we deal with NVM because of its DRAM-like low latency.

By contrast, when accessing NVM, we must avoid expensive system calls triggered by the file system commands as much as possible. This is especially true for applications requiring abundant random memory accesses, such as tree manipulation operations, which may issue several system calls solely for modifying a few pointers and therefore results in significant system overheads.

In order to avoid context switch overheads and to expose the byte-addressability of NVM, in this project we primarily focus on memory mapped file accesses. Here, applications access NVM much like memory, through byte-level load/store interfaces without system calls. However, to utilize the byte-addressability of persistent memory, write ordering issue must be paid special attention [15, 35]. Therefore, we design vNVML such that programmers use it in much the same way as they employ existing POSIX `mmap` access for volatile memory, except that they must follow the transaction-like semantics for write ordering. Meanwhile, the benefits of performance improvement, atomicity, and durability are all provided by this transactional interface. Furthermore, we adopt the method of user space library hoping for executing NVM accesses in the user space as much as possible and using system calls only when necessary.

Since vNVML only provides the `mmap`-like transactional interface and only focuses on the file read/write accesses, meaning that vNVML still relies on file system to provide the file system

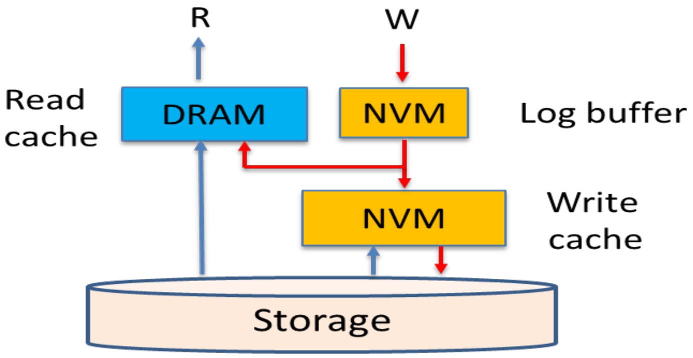


Figure 2.1: The read/write data flow of vNVML private mmap mode between DRAM, NVM, and storage device

related handlings, such as metadata management, directory management, file permission control, etc. vNVML places no limits on which file system may be used in the system and only requires that file system must support standard POSIX `mmap`. Here we want to highlight that vNVML does not try to place the file systems; instead, vNVML hopes to supplement and augment the current design limitations of file systems, especially when applications require the high performance computing.

From here, we focus on vNVML's *private* `mmap` access mode[3], meaning that virtual NVM regions can only be accessed by a single process. The mechanism of our *shared*[4] `mmap` mode is slightly different and, to avoid confusion, is explained in section 2.4.4.

Briefly speaking, vNVML utilizes NVM both as a log buffer and as a write cache and DRAM as a read cache. The reads can only be served by read caches. Modified data are first written to the NVM log buffer only and then copied to DRAM (read cache) when the logged data are committed. NVM (write cache) are updated by committed logs on the background. If pages containing accessed data are not already in NVM or DRAM, they (entire pages) are copied from the storage devices to NVM write cache or DRAM read cache. Data are evicted from the NVM write cache back to storage devices only when the usage of NVM write cache exceeds some threshold (30%) of the physical available NVM. Before programs are terminated safely, all data are completely

[3] Note: our (vNVML) *private* `mmap` mode is different from the standard POSIX *private* `mmap` mode. In our *private* `mmap`, written data can be reflected back to storage devices, but POSIX *private* `mmap` cannot.

[4] Note: the "share" here means that shared regions can be seen and accessed by multiple processes, exactly like the existing POSIX *shared* `mmap` method for volatile memory.

flushed from NVM (both log buffer and write cache) to files. The interactions between DRAM, NVM, and storage devices are shown in Fig. 2.1 for both read and write operations.

The key ideas behind our design choices are as follows:

Use NVM as log buffer: Like other NVM user space libraries, we also adopt transaction semantics as interface for vNVML to provide write ordering, atomicity and durability. All written data must be immediately stored at some temporary non-volatile storage locations before transactions commit. Since this logging process must be in the write critical path, employing NVM as temporary non-volatile log buffer provides significant performance advantage.

***Redo* logging and DRAM cache:** Typically, there are two approaches to logging: *undo* and *redo* logging. Both have pros and cons toward different workloads [36]. *Undo* logging requires that old data are persisted as logs before new data are updated in place. These two actions (both logging and in place update) must be in the write critical path. Alternately, with *redo* logging, all new data are persisted as logs to non-volatile media first, then new data can be updated in place. In-place updates, because they can be executed on the background, do not have to be in the write critical path, but they would affect the read critical path because reads have to be redirected to logs and have to search for the newest data from logs.

In vNVML, we augment *redo* logging by using DRAM as a read cache. Modified data are written to the NVM log buffer, and are also written to DRAM, during the commit command, for reads following this write. With the help of a read cache (DRAM), only persisting writes on the (*redo*) log and updating to DRAM are in the write critical path (from the perspective of the whole transaction), and reads do not need to be redirected to logs as usual (*redo*) log does. Updating data on the storage devices can be executed in the background, without it being in the write critical path.

Although *undo* and our *redo* logging both double the written data in the write critical path (*undo*: 2 NVM versus our *redo*: 1 NVM and 1 DRAM), using our *redo* logging still has three advantages. First, even though the access latency of NVM is close to that of DRAM, the write latency of DRAM is still shorter than that of NVM [37, 38]. So, writing to DRAM is still faster than writing to NVM. Second, writing to DRAM does not need ordering constraints, which use `clflush`, `clflushopt`, `clwb`, and `sfence` instructions and therefore are time-consuming [39]. Third, for read-intensive workloads, read cache can serve some reads without accessing NVM, which might potentially reduce the writes to NVM and alleviate the lifetime issues of NVM.

Only committed data are updated to read cache: Before logs of uncommitted transactions can be placed into true destinations, reading the data still in the logs requires parsing the logs to find the newest data, which can be time-consuming. This process is in the read critical path. In most workloads, the frequency of reads is much higher than that of writes. For example, Yahoo! Cloud Serving Benchmark (YCSB) [40] framework refers to workload A (50/50 read/write ratio) as update-heavy workload. In terms of the overall performance, shortening the read critical path is more important than write critical path. So, we simply use DRAM as a read cache to serve all read operations, and update the data into the DRAM in the commit command (through parsing the logs belonging to this transaction sequentially). By doing so, the following reads, after transaction commits, could read directly from DRAM. Our design doubles the written data on the write critical path, but it makes the reads faster as data can be directly read from DRAM, without having to search the entire log buffers. The section 2.4.1 will explain the detailed mappings of read cache, log buffer, and write cache into virtual address space of each process.

Two restrictions are related to this read cache: (1) reads can only be served by the read cache and (2) written data are copied to read cache only when the transaction commits to accomplish the isolation property; that is, only committed data are visible. This is sometimes referred as "read committed" transaction isolation level [41]. However, our transactions are defined differently from transactions of the traditional database systems. In database systems, the focus is on the consistency of transactions to ensure correct data are accessed between multiple concurrent transactions. In vNVML, we emphasize the persistency [42] of transactions. Here we define the *committed* (*uncommitted*, respectively) data are that the written data must be valid (invalid, respectively) after system crashes; meanwhile, the atomicity and durability of data are also guaranteed. We leave the consistency of transactions to the discretion of programmers/applications.

Employ NVM as write cache: All written data are at the log buffer when transactions commit.

Data need to be gradually moved from the log buffer to their true destinations on storage devices to avoid overflowing the log buffer. We utilize part of NVM as a write cache and committed data are moved to NVM write cache before they are moved to much slower storage devices. This allows us to migrate the logs quickly to more permanent locations and to prevent the log buffer from taking too much space.

A background worker (thread) is responsible for copying data from (NVM) log buffer to (NVM) write cache to avoid extra overhead in the write critical path. This design is also suitable for the cache-friendly workloads (or the write working set of workloads is smaller than write cache size) because logs could always be directly copied to NVM cache (where data are moved from NVM to NVM). Writing data to NVM allows us to maintain data safety, providing a better performance if future writes hit in the write cache.

Update to storage devices from NVM write cache (*private* `mmap` mode) or DRAM cache (*shared* `mmap` mode): Data are written to the log buffer using the write commands, and then moved to NVM write cache in the background. These data are also written to the DRAM read cache upon the commit command being issued. Therefore, the data can have two paths, from DRAM or from NVM, to reach the storage devices. Depending on whether the regions (or modes) of virtual NVM are to be shared across applications or not, we adopt different strategies.

For private virtual NVM regions (*private* `mmap` mode), We construct DRAM (read-only) cache by leveraging the existing POSIX *private* `mmap`, which adopts Copy-on-Write mechanism and all written data can only remain at DRAM. Therefore, the data can only be written back to storage devices from NVM cache. We choose this design choice with the hope that NVM write cache can absorb all write traffic (if cache size is larger than the write working set) and avoid all storage device accesses to gain the maximum performance.

On the other hand, when shared NVM regions (*shared* `mmap` mode) are required, the existing POSIX *shared* `mmap` is adopted to construct the DRAM (read/write) cache, and data are written from DRAM to storage devices. Here NVM cache is not used and logs in NVM log buffer are referenced only when recovering from system failures is needed. This design concept is similar to

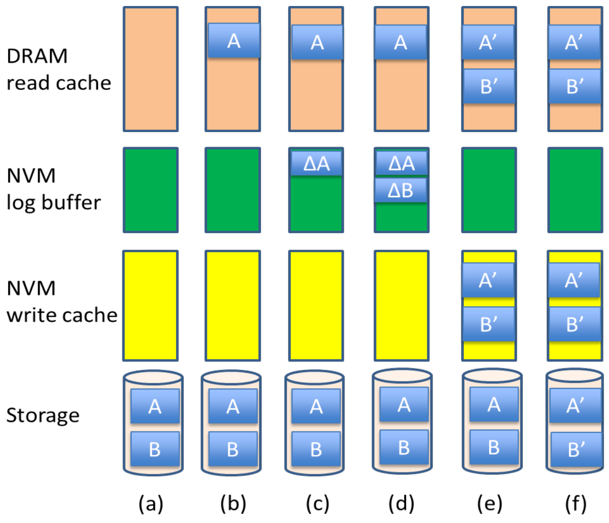


Figure 2.2: The read/write data flow and page movement of vNVML private mmap mode

SoftWrAP [20]. Further details are provided in section 2.4.4.

Fig. 2.2 illustrates the read/write flow of accessing private virtual NVM region and the page movement between DRAM, NVM, and storage device in detail. (a) A file on the storage with page A and B initially. (b) A read from page A lets page A is copied from the storage device to the memory and then the application reads page A directly from the memory. (c) A write to page A results in a log ΔA is appended to the log buffer. (d) Another write to page B also results in a log ΔB is appended to the log buffer. (e) The transaction commits. The page A in memory is updated with ΔA to page A', and page B is copied from storage to the memory by Copy-on-Write mechanism and is also updated with ΔB to page B'. Page A and B are read from storage device to NVM cache and are applied the logs ΔA and ΔB to be page A' and B' by a *redo* background thread. (f) Another writeback background thread writes the page A' and B' from NVM cache back to the storage device.

2.4 vNVML API and Implementation

In this section, we explain the implementation of vNVML in detail. We start from the introduction of the APIs that vNVML provides and describe their functions. Next, we describe the data structures that vNVML manages in the user space of applications, and then introduce two background workers (per process) for parallel processing in vNVML. Then, we explain the implementation of shared regions of vNVML. Finally, we discuss some general issues in vNVML implementation.

2.4.1 vNVML API

Algorithm 1 shows all APIs that vNVML offers and their brief implementation.

Every application (process) first needs to call `nv_init` once before it starts to utilize vNVML. The first caller creates (a log buffer, a cache, along with associated metadata) files in NVM and a shared memory object by calling `shm_open`. The first caller is also responsible for constructing one linked list for pages of NVM cache as a free list and one linked list for pages of the log buffer. Section 2.4.2 describes the linked list data structure in more detail. The shared memory object contains and provides global information accessible by all vNVML users such as number of total current vNVML users, unique application ids assigned to each application, and unique transaction ids for each transaction. Because NVM pages in the free list and in the log buffer do not contain useful information and therefore do not relate to recovery process as well as they also need to be accessible by all applications, their linked list heads are stored in this shared memory object, too.

All callers must *shared* `mmap` all files created in NVM using the `nv_init` command in their virtual address space. These files are mapped by vNVML and are invisible to applications. Therefore, applications have no information of mapped address regions of these files, and all accesses to NVM files from applications can only be through vNVML.

To allocate virtual NVM regions, applications call `nv_allocate` by passing a path *filepath* in the storage, a file size *n*, and the mapping mode (*private* or *shared*). If a file exists in the *filepath*, then that file is opened; if it does not, a new file is created at *filepath* and is `posix_fallocated` with size *n*. The file descriptor *fd* returned from `open` command, along with application id and *filepath* are stored as a file record entry (application id, fd, filepath) of the metadata file for recovery process if needed. Fig. 2.3 illustrates an entry of file record. Finally, A file pointer *fileptr* obtained by (*private* or *shared*) `mmapping` this file is returned to the caller.

Fig. 2.4 illustrates the virtual address space of a process after calling `nv_init` and

```
Function nv_init (void)
    if caller is the first caller then
        initialize vNVML;
        construct linked lists for NVM cache and log buffer;
    end
    mmap NVM files such as cache, log buffer, and metadata into caller's virtual memory
      space;
Function nv_release (void)
    wait for redo background worker to apply committed logs to NVM write cache;
    flush all dirty pages to the storage;
    munmap all NVM files;
    if caller is the last caller then
        release all resources allocated by vNVML;
        erase all NVM files;
    end
Function nv_allocate (path filepath, size n, mapping_mode mode)
    acquire fd by open(filepath);
    get fileptr from mmap(n, mode, fd);
    return fileptr;
Function nv_free (pointer fileptr, size n)
    munmap(fileptr, n);
Function nv_txbegin (void)
    generate a unique transaction tid;
    return tid;
Function nv_write (id tid, address dst, address src, length n)
    if log buffer is needed and no log buffer is available then
        return the number of written data;
    end
    Allocates a page from log buffer if necessary;
    Add written data from address src to src+n as log entries of tid to one of the open log
      lists;
    return the number of written data;
Function nv_commit (id tid)
    update the read cache by parsing logs of tid;
    move logs of tid from one of open log lists to the tail of a committed log list;
Function nv_abort (id tid)
    remove logs of tid from open log lists;
```

Algorithm 1: vNVML API.

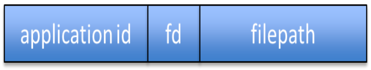


Figure 2.3: Structure of a file record for recovery process

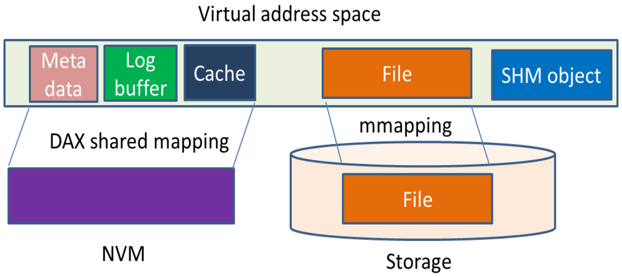


Figure 2.4: The mapping of virtual address space of a process after calling `nv_init` and `nv_allocate`

`nv_allocate` for a file. Only the mapping regions of files in the storage devices are known by applications.

After virtual NVM regions are allocated, applications can access virtual NVM like accessing real NVM through *fileptr* (virtual address returned from `nv_allocate`) for reading or the `nv_txbegin`, `nv_write`, `nv_write`, ..., `nv_commit` command series for writing. The `nv_txbegin` generates and returns a unique transaction *tid* for the following `nv_write(s)` and `nv_commit` commands to construct a single transaction.

The `nv_write` commands are used to write data into virtual NVM. Through `nv_write` commands, all data are written as *redo* logs in the log buffer. The first `nv_write` command must allocate a log page from log buffer. If a log page is needed but no log page is available, then `nv_write` returns the size of written data so far.

To write logs, a log object to store the log pages of this transaction is allocated from NVM and is put into one of 32 open lists of this process according to its transaction *tid*%32 (modulus operator). Log pages, allocated from the linked list of log buffer by the same transaction, are appended to the tail of the corresponding linked list of the log object in the open lists.

A single `nv_write` command may create several log entries. It first depends on the destination position and then depends on the left space of the current log page. This is because we want the data from a single log entry to be placed entirely within a single NVM cache page to simplify the design and implementation of redo background worker described at section 2.4.3.

```
nv_init();
ptr = nv_allocate(filepath, filesize, mode);
tid = nv_txbegin();
x = 100;
y = 200;
nv_write(tid, ptr, &x, sizeof(x));
nv_write(tid, ptr+sizeof(x), &y, sizeof(y));
nv_commit(tid);
nv_free(ptr, filesize);
nv_release();
```

Algorithm 2: Example of vNVML.

Inspired by some prior works [43, 39, 15], we know when dealing with byte-addressable, memory-bus attached NVM, write ordering is required. To do so, modern CPUs provide cacheline flushing (clflush, clflushopt, and clwb) instructions and memory fence (sfence and mfence instructions to help to enforce write ordering. For example, store-clflush pair combined with mfence (or sfence) is adopted by several prior works [8, 14, 13, 10]. Some other prior works [20, 10, 7, 39] further replace the store-clflush pair with non-temporal store (ntstore) instructions. The ntstore instructions can bypass the CPU caches and directly write data to DRAM or NVM. By doing so, ntstore eliminates the expensive cacheline flushing operations [39] and significantly improves the NVM persisting performance. Therefore, we also employ ntstore to write data to log buffer at nv_write commands.

When the nv_commit command is called, vNVML sequentially traces all log entries of log pages from the linked list of log objects for the committed transaction *tid*, and copies all committed data from log entries to the DRAM read-only cache. Because writing to DRAM does not require ordering, the standard memcpy function is sufficient for copying and therefore may be used to attain better performance. After copying to the DRAM cache, vNVML moves this log object (along with all log pages linked to this log object) from the corresponding open list to the tail of the only committed list of this process and persists all log entries and a log object in NVM. This committed list head is stored at the metadata of the applications in NVM. All log entries in the committed list are guaranteed to be preserved across power failures.

Finally, applications call nv_free (nv_release, respectively) if they do not want to access a certain file (do not want to access entire virtual NVM, respectively). nv_free is used to munmap a file mapped by the nv_allocate. After nv_release is called, the applications must wait for all committed logs, if they exist, to be applied to NVM cache by the redo background worker, actively flush all dirty cache pages back to the storage devices, and munmap all NVM files mapped at nv_init.

Algorithm 2 shows a typical example of using vNVML.

2.4.2 vNVML data structures

The NVM log buffer and NVM cache are partitioned into units of 4KB pages and organized as linked lists. The implementation of linked lists for pages is via metadata; that is, for each page, a corresponding page object is created from NVM metadata file and connected with each other as a linked list. Therefore, allocating a page object from the linked list also equals to allocating the corresponding page.

As mentioned in [44], however, constructing the linked list in NVM is not the same as constructing a typical linked list in memory. The virtual address cannot be directly used as a pointer to be stored in NVM because there is no guarantee that the NVM files can be mounted into the same virtual space regions (1) by multiple concurrent processes or (2) by a single process before and after system crashes. Thus, we replace the address with the index of the page starting from 0 to construct the linked lists in NVM. Similarly, we substitute the offset from the starting address to the current position, when an access needs to be made, for the address to be stored into NVM.

Page objects for the log buffer and cache are created by different metadata files at nv_init. After a page object of log buffer is allocated, the application id is stored into page object, and the log entries (from nv_write) can be written directly into the corresponding log pages. The first field of the log page is the total written bytes to this page, and log entries are appended sequentially. Fig. 2.5 illustrates the fields of a log page and log entries in the log page. The log entry contains log header, including length of this entry, file descriptor, and offset (from the first byte of this file to the destination location), followed by the *redo* raw data. The page object (application id), the log

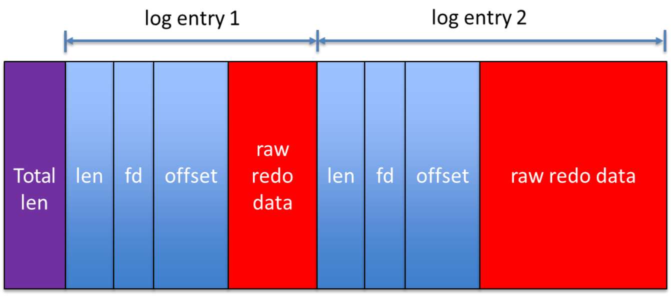


Figure 2.5: Structure of a log page and its log entries

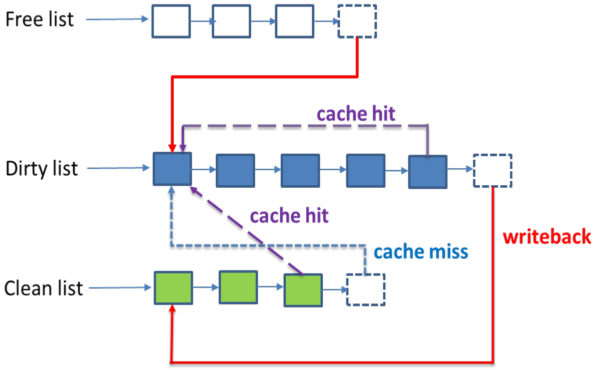


Figure 2.6: Page movements of LRU policy between free list, dirty list, and clean list

entry header (len, fd, and offset), and file record (application id, fd, and filepath) already contain all necessary information for the recovery worker to write the committed log entries directly back to corresponding files of storage devices.

To handle cache pages, one free list is created through the shared memory object, and the others, dirty and clean lists, are created within each application. The dirty and clean lists implement the LRU replacement policy.

Fig. 2.6 illustrates the page movements between the free list, dirty list, and clean list. Cache pages are always allocated from the free list, until the page share of an application is reached, and become dirty pages inserted to the head of dirty list after the redo background worker copies the corresponding pages from files in the storage devices and applies corresponding log entries on them. Dirty pages (of the tail of dirty list) become clean ones and are inserted to the head of the

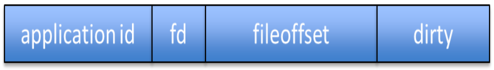


Figure 2.7: Structure of a page object for writeback and recovery process

clean list after the writeback background worker writes the dirty pages back to files in the storage device. When a cache is hit, regardless of whether the page is in the dirty or clean list, the page is applied the redo log and inserted into the head of the dirty list. When a cache miss happens, the tail page of clean list is always picked and is filled with the corresponding page from file in the storage devices. This page is inserted into the head of the dirty list after writing the redo log on it.

For the individual cache page, besides the application id of the page owner, some extra information is also stored into the corresponding page object, such as the fd (file descriptor), file offset, and dirty flag. The file offset is the offset which is used to `seek` the file and to access the page of files in the storage devices. The fd and file offset can be known from the header of log entries by redo worker when it redoes. The dirty flag is set only if this page is dirty (in the dirty list). This flag is cleaned after writeback worker writes this dirty page back to files and puts it into the clean list. Thus, the recovery worker only needs to handle the pages whose dirty flag are set. Also, the information contained in the page object (application id, fd, file offset, and dirty flag) and the file record (application id, fd, and filepath) are enough for recover worker to write the dirty pages back if system crashes. Fig. 2.7 illustrates an entry of page object.

Partitioning the log buffer and cache page at a 4KB page-size granularity and organizing them as linked lists has some advantages. First, the allocation and deallocation of pages from log buffer and free list are both *O(1)*. Second, the management of log buffer and cache space becomes easier since the space is managed in terms of pages, rather than bytes or variable size segments. Third, it makes it easier to share pages of the log buffer and free list across applications through linked lists maintained at the shared memory object.

To prevent a single application from allocating all log pages and all cache pages, vNVML adopts the equal share policy through the shared memory object, containing the total number of

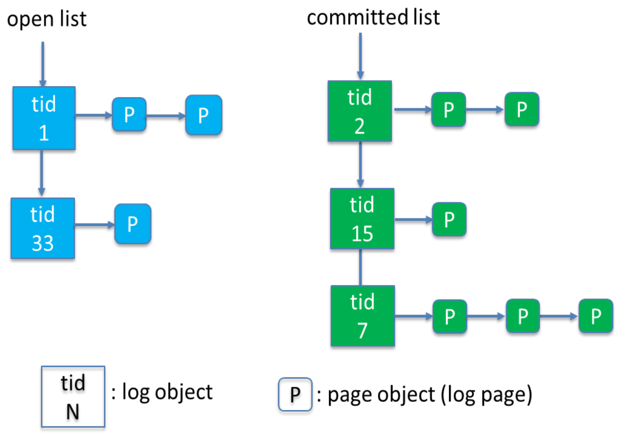


Figure 2.8: Open list contains log objects of transactions. Each log object may link several page objects (of log pages). After transaction commits, the log object (along with its log pages) of the transaction is appended to the tail of the committed list. Redo worker always redoes from the head of the committed list; therefore, the transactions which is committed early would also be replayed early

current applications. Applications can allocate log pages from the log buffer or cache pages from the free list if and only if the number of allocated pages does not reach their shares. A new joining user of vNVML may result in all current users exceeding their shares. Two background workers, described in the following section, help to return extra pages back to log buffer and free list.

Fig. 2.8 illustrates the relation between open list, committed list, log object, and page object.

2.4.3 Background workers

Two background workers (threads) are created for each process at `nv_init`. The redo background worker keeps checking the committed list. If the committed list is empty, the worker goes to sleep for a while (10us in the current configuration) and then checks the committed list again after it wakes up. If the committed list is not empty, the redo worker obtains the first log object (of some transaction) from the head of the committed list, and replays all the log entries sequentially from log pages of this log object to NVM cache pages. Since in this redo operation, data are moved from NVM to NVM, write ordering is also required; therefore, we also employ `ntstore` instructions here for writing to NVM cache pages. If a cache miss happens, redo worker is also responsible for reading this page from files in the storage device to NVM cache page. All log pages can be discarded and returned to the log buffer pool only after the entire logs of a transaction are completely replayed by the redo worker.

Here we want to highlight the limitation that, since the committed logs are always appended to the tail of the only one committed list (per process) at `nv_commit` (mentioned at section 2.4.1), and redo background worker always redoes logs from the head of this committed list; therefore, programmers should keep in mind that the transaction which writes to an object first should also be committed first for the consistency of DRAM cache and the file on the storage device when multiple threads write to the same objects, but there is no constraint when threads write to different objects. This is a much weaker constraint compared to accessing a file by file system, which requires locking the whole file, even if different objects of the same file are being accessed.

The other writeback background worker is responsible for writing the dirty NVM cache pages back to the storage devices. To avoid accessing storage devices too frequently, we employ a threshold on dirty NVM pages accumulated in the dirty list (we use 30% of cache page share). Dirty pages are written back to the storage device by the writeback worker after the number of dirty pages is more than threshold. The dirty pages are inserted to the head of the clean list after writing back to storage devices. Furthermore, if the number of allocated cache pages exceeds the share due to new joining applications, the writeback worker may further release some clean pages back to the free list. After the number of the dirty pages drops below some threshold (we set 10% of cache page share), the writeback background worker is stopped and dirty pages may accumulate again.

Both background workers are killed upon the `nv_release` command.

2.4.4 Sharing NVM between processes

The implementation of shared NVM regions between processes is slightly different from what we have implemented for private regions. What we mean by "shared region" here is that a memory region can be accessed concurrently by multiple processes and all processes could see the same view of this region. This is exactly the same as existing POSIX *shared* `mmap` for volatile memory.

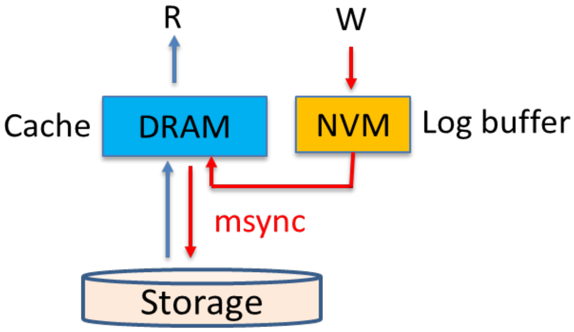


Figure 2.9: The read/write data flow of vNVML shared mmap mode between DRAM, NVM, and storage device

Shared regions are constructed by `nv_allocate` command. When `nv_allocate` is called, a file is *shared* `mmapped` to construct the DRAM read/write cache and a committed list head is created and maintained as metadata in NVM for each *shared* `mmapped` region. By DRAM cache constructed from *shared* `mmap` of the file, processes which require to access this region can share the same view.

Because a committed list is required for each shared region, the same limitation, the transaction writing to an object first should also be committed first, must also be obeyed. In addition, one more limitation is required when writing to shared regions; that is, all true destinations of writes from a single transaction must lie within the same single shared region. Without this limitation, if a transaction can write to multiple shared regions, then multiple log objects, each log object is for a shared region, should be created to be appended to multiple committed lists when transaction commits. This would cause a serious problem if system crashes while a transaction commits, resulting in some log objects have been appended to some committed lists but some not.

When we write to a shared region, data are still written to log buffer first. At transaction commits, the data from log buffer are replayed to the DRAM of a single *shared* `mmap` region, and all logs of this transaction are moved from one of 32 open lists of a process to the committed list of this shared region. We do not utilize NVM cache here to simplify our design and implementation. Fig. 2.9 illustrates the read/write data flow of accessing a shared virtual NVM region.

Since *shared* mmap is employed to create our shared regions, the msync system call is required to write the modified data back to storage devices. Different from fsync, however, calling msync requires to know all the modified virtual address regions (i.e. start addresses and their lengths). To avoid parsing all logs when flushing (msync) data back to storage devices later, a global bitmap of dirty pages dedicated to this shared region is maintained in DRAM and is updated at the end of each nv_commit command by the local bitmap of each transaction; the local bitmap is constructed when logs are parsed and replayed to DRAM at nv_commit.

After logs have been accumulated to a certain threshold, upon a call to nv_commit, a background thread is triggered and atomically copies the global bitmap of this shared region to a "copied global bitmap" variable, zeroes this global bitmap, and marks the tail transaction of the committed list. Then several msyncs might be issued to flush dirty pages back to storage devices according to the copied global bitmap. Only after flushing is completed, can logs of transactions (from the head to the marked tail) be removed from the committed list.

During the flushing procedure, new transactions can still proceed, be committed, and update the "zeroed" global bitmap. Because their logs are always appended after the marked tail transaction and "read committed" policy is employed in vNVML, unexpectedly flushing some data of transactions committed after "marked tail" transaction during the flushing procedure will not harm since all data in DRAM must have been committed and therefore must be written back to storage devices eventually.

The logs from head transaction to marked tail transaction are discarded only after flushing is completed. If system crashes, recovery procedure always replays all logs from the head of committed list of this shared region.

2.4.5 Transaction aborts and long running transactions

For some extreme cases, the log buffer may run out of space if too many long running transactions, which keep writing data before commitment, execute concurrently. This situation can be detected when pages cannot be allocated from the log buffer pool for a while. vNVML could actively abort long running transactions by recording the timestamp into the log objects when log objects are allocated by transactions. The redo worker can periodically check the log objects from the head of each open lists and can abort the transactions whose elapsed time exceed some predefined threshold. Applications can also abort transactions for various reasons.

When a transaction is aborted, since all its logs are still in an uncommitted state (in the open list), these logs can be discarded directly and log pages are returned back to the log buffer pool. Moreover, because transactions of vNVML support the "read committed" isolation property, when one of the nested transactions needs to be aborted, aborting all transactions involved in the nested transactions may not be necessary and it should depend on the discretion of users.

2.4.6 Data recovery

Systems or applications may crash due to an unexpected failure at any moment such as power shortage, bugs of applications, or inadequate kernel resources. The mandatory function any NVM solution should provide is to ensure the data persistency after systems or applications crash. In vNVML, we handle this by a recovery program run by *root*. After systems reboot, a recovery worker (process) first `mmaps` the all NVM files (log, cache, and metadata) into its virtual memory space. From section 2.4.2 we know the recovery worker already has all necessary information to recover the dirty pages and log entries back to files in storage devices by tracing the page objects, file records, and committed lists.

We always recover/write the dirty pages (by checking if dirty bit is set) back to files before we recover the committed logs back to files because the committed logs contain the newest data. Reversing this order might result in that the new data are covered by older data from dirty pages.

The order of objects in the committed list is important and we should replay the objects sequentially. With the help of 8-byte atomic update feature natively supported by processors, the order of objects can be maintained correctly by carefully handling the order of pointer updates between objects of linked lists.

Fig. 2.10 illustrates the process of insertion and deletion of an object to and from a linked list at NVM. For object insertion (the correct sequence is from (a) to (b) to (c)), we could assume object C has been inserted into linked list only when system crashes after (c); otherwise, we assume object

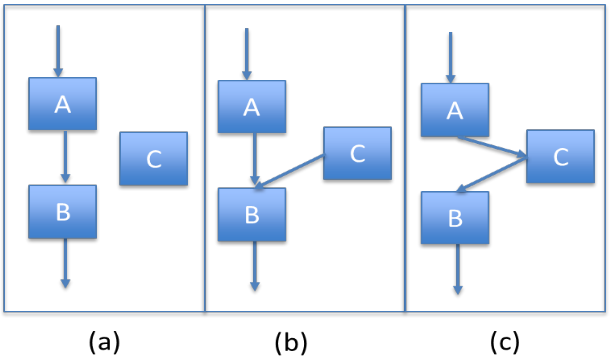


Figure 2.10: The correct order of pointer updates for the objects of linked lists in NVM. From (a) to (c) is for the object insertion; from (c) to (a) is for object deletion

C is not inserted yet. On the other hand, for object deletion (the correct sequence is from (c) to (b) to (a)), when system crashes after (c) we would assume object C has been deleted from the linked list.

After the recovery process finishes, all NVM files are erased, and vNVML can be restarted again. This recovery process may be re-executed as many times as needed if the system ever crashes again during the recovery process since all the required data and metadata are conserved in NVM and are erased only after a successful recovery. Thus, all data have been written back to their true destination of files.

2.4.7 Security

Security is a major concern in the modern computer systems, especially in the data center, where infrastructure has to protect against any attacks from third party applications. In vNVML, the security is guaranteed in two aspects. First, the private regions are produced by *private* `mmap`. Due to the Copy-on-Write mechanism brought from *private* `mmap`, all the direct writes within this private address region will remain within the memory (virtual address space of the user process) and cannot impact the contents at the storage device.

Second, all the writes to private regions must be executed through `nv_txbegin`, `nv_write`, `nv_write`,..., `nv_commit` command series. Those APIs are entirely controlled by vNVML and accessing NVM (log buffer, cache, and metadata) files, which are invisible to applications, is not allowed outside vNVML. When applications try to write beyond the mapped regions (or outside allocated virtual address regions), the protections within the existing memory system will detect these violations. In addition, the vNVML bound checks will not allow these writes to proceed.

2.5 Evaluation

In this section, we conduct experiments to answer fundamental questions about vNVML as follows:

- What are the characteristics of the vNVML?
- How does vNVML impact the performance when used by real applications?
- How to decide the size of the log buffer and the cache given a fixed and limited size of NVM in the platform?
- How does the vNVML perform when multiple processes concurrently access the NVM through vNVML?
- What is the impact of using vNVML within the container environment?
- What is the impact of different log buffer sizes, total cache sizes, and single cache page size on life span of backend SSD?
- How does the vNVML perform compared to other user space libraries?

2.5.1 Experimental setup

Due to the absence of real NVM, we emulate NVM by DRAM for all our experiments. We mount the NVM with the Ext4 file system in order to utilize the DAX (direct access) feature provided by Ext4.

We evaluate vNVML on a platform with 16GB DRAM, 12GB emulated NVM, and Intel i7-4770 four-core 3.4 GHz processor with hyperthreading enabled. Samsung enterprise PM863

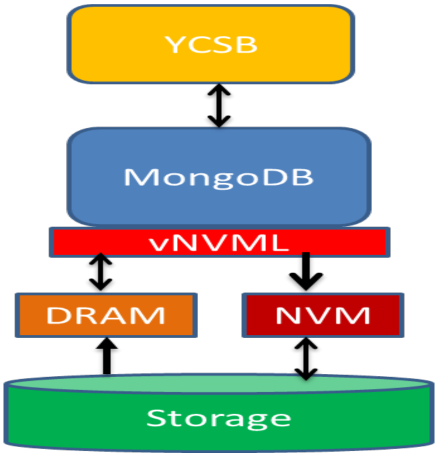


Figure 2.11: The experimental setup of YCSB, MongoDB, and vNVML

480GB SSD (SATA 6.0 Gbps) is adopted as our example of the storage devices. We implement vNVML on the Linux kernel 4.13 version.

2.5.2 MongoDB and YCSB

In this subsection, we explore and analyze the impact of accessing NVM through vNVML by real applications. We adopt a popular open-source database MongoDB version 3.6.0 [45] as our target application because its MMAPv1 storage engine uses memory mapped file form to access the data in the storage devices, which is perfect for our vNVML to employ. We modify part of the source code of MongoDB for our transactional interface to deploy vNVML.

We choose YCSB [40] to generate the read/write traffic of MongoDB. The setup of experiment is delineated in Fig. 2.11. To simplify our analyses, we configure the size of all records' fieldcount as 128 and fieldlength as 512 and readallfields and writeallfields are both set as true in the configuration file of YCSB workloads, meaning that each read/write request will access exactly 64KB data, which is also the data written per transaction. 100K operations (read/write requests) are executed for all experiments. We deploy the different read/write ratio and two request distributions (zipfian or uniform) to observe the impact of performance.

YCSB has two phases: one is inserting records into the target data store, the other is accessing (read or write) records in the target data store. To avoid polluting the NVM cache of vNVML

Table 2.2: Throughputs of single MongoDB instance with different number and distribution of inserted records when NVM is the only storage device

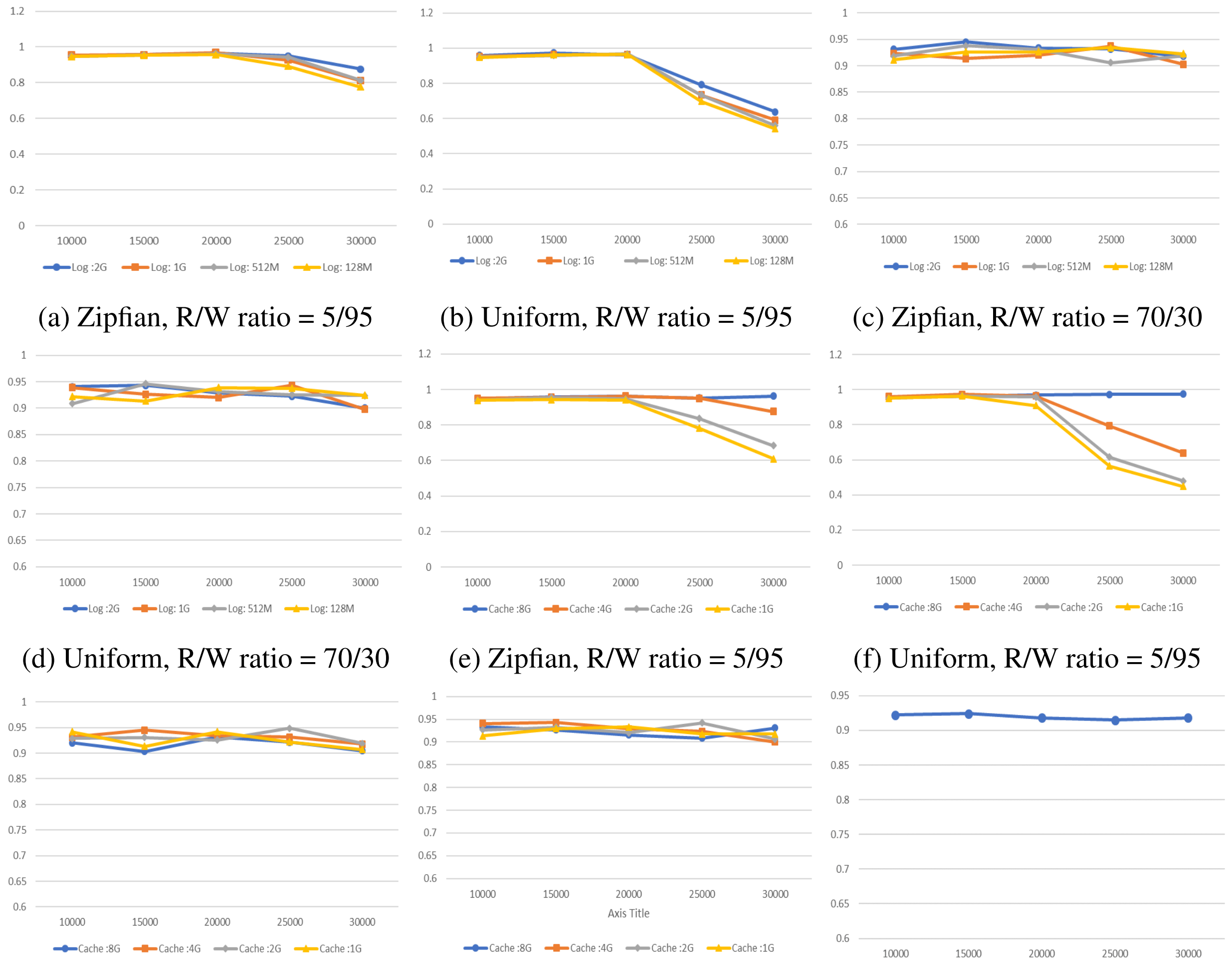
# of records	uniform	zipfian
30K	1500 (op/s)	1505 (op/s)
10K	1509 (op/s)	1498 (op/s)

before the accessing phase, in the insertion phase MongoDB employs `nv_allocate` to *private* `mmap` files in the storage devices, and then, instead of using `nv_write` vNVML commands, MongoDB only adopts its original (unmodified) insertion functions to access the memory mapped regions, meaning that all records are only inserted into the memory (due to the Copy-on-Write mechanism provided by *private* `mmap` used by `nv_allocate`).

All experiments are conducted by accessing four MongoDB instances concurrently in a single OS. However, since the MMAPv1 storage engine uses padding and the power of two sized allocation mechanisms [46], four instances would generate total 8.8GB files in the storage devices when each instance is inserted 10K records, and total 33GB files after 30K records are inserted into each of four instances. Therefore, 12GB emulated NVM in our platform can only accommodate files created by four MongoDB instances inserted at most 13K records, respectively. However, from table 2.2, we find that the YCSB throughputs of one instance are very close to each other even with different numbers and different distributions of inserted records if all files generated are stored only in NVM. We assume this observation still holds in the 4-instance case. Therefore, we insert 10K records to each of four instances, remove the periodic `msync` calls by MongoDB, disable journaling with *nojournal* option, and employ NVM as the only storage device of MongoDBs as our baseline[5].

Fig. 2.12 shows the normalized throughputs of different request distributions (zipfian and

[5] Note: our platform does not have enough NVM to accommodate all files generated by eight instances inserted 10K as baseline, respectively. Also, when eight MongoDB instances adopting vNVML are running concurrently, some of instances would crash because of out-of-memory (OOM) error from *private* `mmap`. Therefore, executing at most four instances concurrently is the limitation of our platform.



(a) Zipfian, R/W ratio = 5/95 (b) Uniform, R/W ratio = 5/95 (c) Zipfian, R/W ratio = 70/30

(d) Uniform, R/W ratio = 70/30 (e) Zipfian, R/W ratio = 5/95 (f) Uniform, R/W ratio = 5/95

(g) Zipfian, R/W ratio = 70/30 (h) Uniform, R/W ratio = 70/30 (i) Uniform, R/W ratio = 100/0

Figure 2.12: Normalized total throughput of four instances. Numbers of X-axis stand for inserted records to each database, and numbers of Y-axis stand for normalized throughput. (a) to (d): Fix 4GB cache size and adjust log buffer size from 2GB to 128MB. (e) to (h): Fix 2GB log buffer and change cache size from 8GB to 1GB. (i) 100% read uniform request

uniform) and different read/write ratios (5/95, 70/30, and 100/0). The results (normalized throughputs) are the summation of throughputs[6] (op/s), generated from YCSB when YCSB accesses one of four MongoDB instances employing vNVML, divided by the summation of throughputs of four MongoDB baseline instances with the same request distributions and the same read/write ratios.

From these results we can make some useful observations. First, the case that cache size is 1GB, log size is 2GB, and four MongoDB instances with 30K inserted records (4 * 30K *

[6] Note: since all four throughputs from YCSB are almost the same (with usually less than 1% difference).

64KB = 7.32GB) already proves that vNVML can provide virtualization and shareability of NVM successfully.

Second, not only can vNVML achieve over 90% of the throughput of baseline (if the log buffer and cache can "absorb" the input write traffic, such as the line of 8GB cache in Fig. 2.12 (e) and (f)), but vNVML can also provide the guarantees of atomicity, persistency, and write ordering, which are our baseline, the MongoDB without journaling and write ordering, cannot. This less than 10% overhead results from writing data to NVM log buffer and from redoing logs to DRAM read cache.

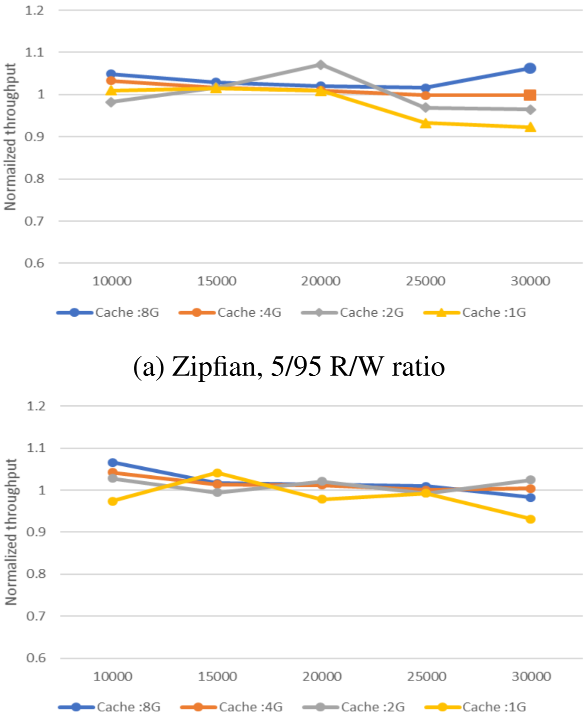
Third, larger write working sets (more inserted records or uniform access requests), and more write requests (lower R/W ratio) degrade the throughput of vNVML. Larger write working sets require more NVM cache to store all data at run-time. Furthermore, if the write working sets are even larger than the capacity of NVM cache owned by applications, then cache pages are frequently written back to storage devices. Finally, when all cache pages become dirty, the overall performance would deteriorate to write throughputs of storage devices.

Fourth, through (a) to (d), when cache sizes are all fixed, the adjustment of log buffer only affects at most 10% normalized throughput[7] of baseline in all these cases.

Fifth, from (e) to (h), when sizes of log buffer are fixed, their throughputs vary highly, especially in the case of (f): read/write ratio is 5/95, uniform request, and 30K inserted records. In (f), the throughputs differ by almost 50%, meaning that cache size impacts vNVML throughput more significantly than that of the log buffer. For the bottleneck of vNVML performance is the access throughput of storage devices, the cache size can impact vNVML performance significantly. As more NVM caches are able to store more write traffic, less writes (if NVM cache hits) to storage devices will be needed. The actual throughput of vNVML should be a function dominated by factors of NVM cache size and access performance of storage device.

On the other hand, unlike NVM cache, NVM log buffer only temporarily stores the write

[7] Note: At (a), the normalized throughput is 0.876 (0.776, respectively) when log buffer is 2GB (128MB, respectively) and inserted records are 30000. At (b), the normalized throughput is 0.638 (0.542, respectively) when log buffer is 2GB (128MB, respectively) and inserted records are 30000.



(a) Zipfian, 5/95 R/W ratio

(b) Zipfian, 70/30 R/W ratio

Figure 2.13: Normalized throughput of four instances inside Docker container

traffic as logs before logs are written to NVM cache pages. As a result, log buffer can only impact/improve performance slightly until log buffer is full; usually the log buffer will be full much quickly than NVM cache if the write working sets are huge and the storage devices are frequently accessed in order to fill the NVM pages before applying the corresponding log entries to cache pages.

Finally, (i) shows at 100% read, uniform distribution request case, vNVML can achieve around 92% throughput regardless of the number of inserted records. It matches our expectation of vNVML since the read is entirely handled by the read cache (memory) and 16GB memory is enough to handle the 30K records working set since 30K $\times$ 64K $\times$ 4 $\sim$ 7.32GB.

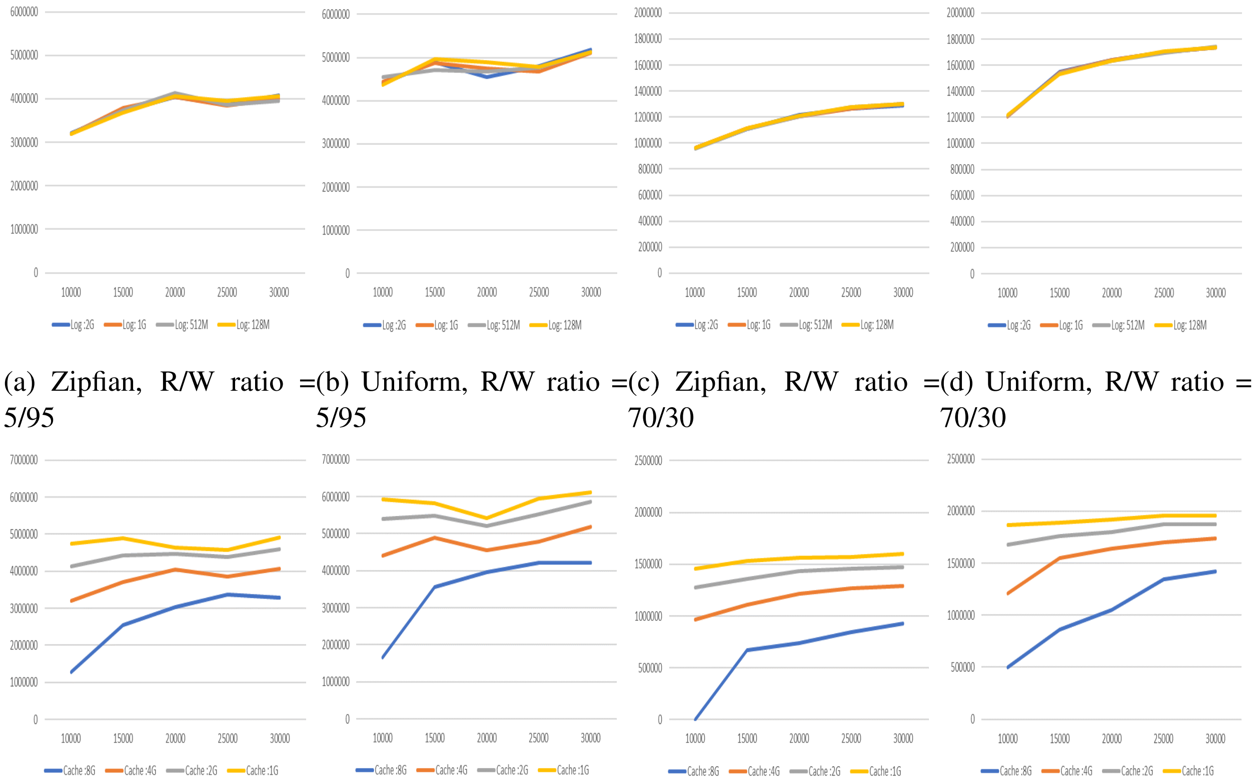
Therefore, from aforementioned observations, we suppose that under limited NVM resources, only some reasonable amount of NVM should be allocated as log buffer, and the rest should be cache to achieve higher performance of vNVML.

Next, we would like to examine the impact of using vNVML within docker containers [47]. Docker is a popular virtualization technique in data centers and recently has drawn significant attention from industry and academia due to its lightweight execution environment compared to traditional virtual machines. In this experiment, we launch four docker containers, use **bind mount** [48] to mount 12GB emulated NVM into each container so all containers can access and share content in NVM, and run single MongoDB instance within each container. Log buffer is configured as fixed 2GB, the cache size as well as read/write ratio are adjusted to various settings. Each data point is normalized with individual counterpart, which is the same configuration without using containers. Fig. 2.13 shows that all the data are close to 1; that is, using vNVML within docker containers does not affect the performance.

Moreover, we want to examine the impact of different sizes of NVM log buffer and NVM cache on life span (or write counts) of backend SSD. We measure the number of total NVM (cache) pages written to storage devices after all logs have been replayed from log buffer to NVM cache pages, and cache pages would then be written to backing SSD if the percentage of dirty pages are over 30%. Fig. 2.14 shows the results, from which our conclusions are drawn.

First, larger write working sets usually have more numbers of written pages (more write counts), but some exceptions can also be found. For instance, cache size is 1GB at (e) and (f). This phenomenon is because writeback background worker starts to write NVM cache pages back only when the percentage of dirty pages is over 30%. So, it is possible that even larger write working sets make all instances with higher percentage of dirty pages, but none of them is more than 30%. On the other hand, smaller write working sets cause instances with lower percentage of dirty pages, but once the percentage of one instance is over 30%, then smaller write working sets might result in more write counts than larger ones.

Second, from (a) to (d), all results (of various log buffer sizes) are almost the same. This also proves that log buffer only temporarily stores the write traffic and cannot influence the access frequencies and patterns of storage devices. It also matches the conclusion made from results of Fig. 2.12.



(a) Zipfian, R/W ratio = 5/95

(b) Uniform, R/W ratio = 5/95

(c) Zipfian, R/W ratio = 70/30

(d) Uniform, R/W ratio = 70/30

(e) Zipfian, R/W ratio = 5/95

(f) Uniform, R/W ratio = 5/95

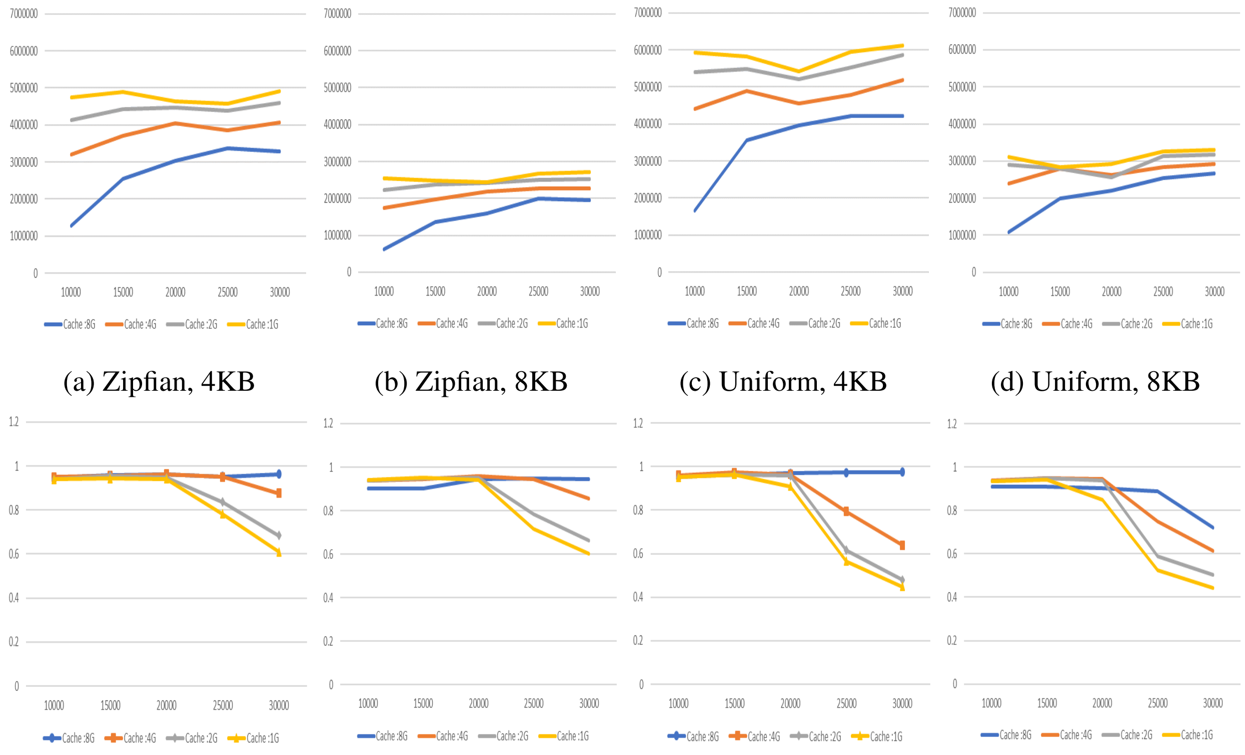
(g) Zipfian, R/W ratio = 70/30

(h) Uniform, R/W ratio = 70/30

Figure 2.14: Total number of NVM cache pages written to SSD of all four instances. Numbers of X-axis stand for inserted records to each database, and numbers of Y-axis stand for total number of pages written to SSD. (a) to (d): Fix 4GB cache size and adjust log buffer size from 2GB to 128MB. (e) to (h): Fix 2GB log buffer and change cache size from 8GB to 1GB.

Third, one interesting point is the case of 8GB cache at (g). We can find out that when number of inserted records is 10000, the number of write count is zero; that is, all writes are stored entirely on NVM cache and none is written to SSD. This means that if we have enough NVM cache to store incoming write traffic, vNVML can achieve not only better performance but also longer life span of SSD.

Next, we consider the impact of different SSD page sizes on write counts and performance. We assume the size of NVM cache page should be the same as the page size of backend SSD; otherwise, the write amplification must be considerable. As a result, we only change the page size of NVM cache as 8KB and compare the write counts and performance with NVM cache of 4KB



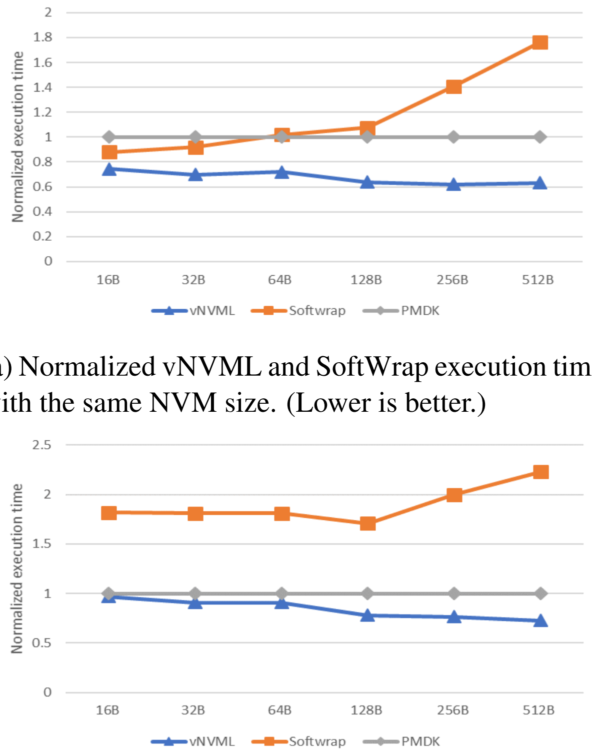
(a) Zipfian, 4KB (b) Zipfian, 8KB (c) Uniform, 4KB (d) Uniform, 8KB

(e) Zipfian, 4KB (f) Zipfian, 8KB (g) Uniform, 4KB (h) Uniform, 8KB

Figure 2.15: Comparisons of 4KB and 8KB size of NVM cache page. All read/write ratios are 5/95. The log buffer size is 2GB and cache sizes are from 8GB to 1GB. Numbers of X-axis stand for inserted records to each database. (a) to (d): Total number of NVM cache pages written to SSD of all four instances (numbers of Y-axis) with different request distributions and page size. (e) to (h): Normalized throughputs (numbers of Y-axis) with different request distributions and page size.

page size. Since we have learned from Fig. 2.14 that the size of log buffer cannot impact the write counts, we only employ different cache size here. R/W ratios of all experiments are fixed as 5/95. Fig. 2.15 shows the result.

From our experiments, doubling the page size (as 8KB) will cause the write counts slightly more than 50% of those of 4KB page size at all experiments, and has almost no impact on the performance of vNVML. This is reasonable because larger cache pages can absorb more write logs than smaller cache pages before they are written to SSD and therefore require less writes to backend SSD.



(a) Normalized vNVML and SoftWrap execution time with the same NVM size. (Lower is better.)

(b) Normalized vNVML and SoftWrap execution time assuming unlimited NVM size. (Lower is better.)

Figure 2.16: Normalized vNVML and SoftWrap execution time. Numbers of x-axis stand for the amount of written data per page

2.5.3 Microbenchmark

We use a simple microbenchmark to compare the performance between Intel's PMDK [31], SoftWrAP [20], and our vNVML. In this experiment, we create a 2GB array in NVM (virtual NVM, respectively) for PMDK and SoftWrAP (vNVML, respectively), and write different amounts of data (from 16B to 512B) per page sequentially. Each transaction contains 32 page writes.

To use PMDK, we use `pmemobj_create` to create a 4GB NVM pool because 2GB NVM pool is not enough to accommodate 2GB array. We always set **PMEM_IS_PMEM_FORCE=1** when executing PMDK to avoid unnecessary `msync` or `fsync` when accessing NVM. For fairness, we use 2GB log buffer and 2GB cache when running vNVML. We only use default setting for SoftWrAP since it does not provide API for internal buffer size adjustment.

Fig. 2.16(a) shows the result. We use the total execution time of PMDK as our baseline, and show the total time of writing the 2GB array for once. The result indicates that among others our vNVML performs better as the total written data keeps increasing. Fig. 2.16(b) shows another experiment, which we enlarge the NVM to 8GB and want to compare the upper bound of each library. We write the 2GB NVM array 16 times. Its result is similar as Fig. 2.16(a).

2.6 Conclusion

In this paper we presented vNVML, a byte-level user space library to access NVM that provides transaction-like semantics for applications, ensures write ordering, and provides persistency guarantees across failures. Our system employs NVM as a write log and a write cache, while also employing DRAM as a cache.

We implemented vNVML and evaluated it with realistic workloads to show that our system allows applications to share NVM, both in a single OS and when docker-like containers are employed. The results from the evaluation show that vNVML incurs less than 10% overhead while providing a larger than available physical NVM space to the applications and allowing them to safely share the virtual NVM.

3. OPTIMIZING POST-COPY LIVE MIGRATION WITH SYSTEM-LEVEL CHECKPOINT USING FABRIC-ATTACHED MEMORY*

3.1 Introduction

The emerging Non-Volatile Memories (NVM), such as phase-change memory (PCM) [1], NVDIMM [2], and 3D XPoint [4], have byte-addressability and low latency, close to that of main memory, together with the non-volatility of storage devices. Similarly, recently emerging interconnect fabrics, such as Gen-Z [11], provide high bandwidth, together with exceptionally low latency. These concurrently emerging technologies are making possible new system architectures in the data centers including systems with Fabric-Attached Memories (FAMs) [12]. FAMs can serve to create scalable, high-bandwidth, distributed, shared, and byte-addressable NVM pools at a rack scale, opening up new usage models and opportunities. These technologies have great potential to improve the system/application performance as well as to provide scalability and reliability of applications/service. Much prior work focused on new system designs using NVM [19, 18, 9, 29, 5, 6] and has already shown promising performance improvements.

As the VLSI technology continues advancing and the scale of transistors continues to shrink, shrinking transistors and therefore components made from them become more unreliable [49]. Similarly, shrinking per-bit area in memory technologies is leading to greater memory errors [50], as bits are corrupted randomly or damaged permanently due to one of many compounding failure mechanisms. These hardware errors are further compounded by numerous software bugs, operator/human error, and/or power outages that could interrupt running programs or even crash the entire system. Due to these vulnerability in existing hardware and software, checkpoint and restart (CPR) is an important technique for fault tolerance; that is, a given application's state is periodically saved (checkpointed) into persistent storage devices such that it may be restarted in the event

*©2019 IEEE. Part of the data reported in this chapter is reprinted, with permission, from Chih Chieh Chou, Yuan Chen, Dejan Milojicic, A. L. Narasimha Reddy, and Paul V. Gratz, "Optimizing Post-Copy Live Migration with System-Level Checkpoint Using Fabric-Attached Memory," in *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, 16-24, November, 2019.

of a failure.

Beyond fault tolerance, non-live migration leveraging CPR is also another crucial technique in data centers, especially for load balancing. Migration allows system administrators to remove some applications from stressed physical nodes in order to redistribute load, and therefore to increase overall system performance. In addition, migration can also provide power saving [51], and online maintenance. Traditional approaches to migration are non-live; that is, they require the application to be taken off-line while the migration occurs. Non-live migration can be divided into three steps: 1) the program is checkpointed at source, 2) the checkpointed data are copied from source to target, and 3) the program is restarted at target. The main drawback of non-live migration is that its application downtime (off-line time) is too long. To reduce this downtime, live migrations [52] are proposed to migrate most of (or all) pages before (pre-copy) or after (post-copy) "real" migration, and therefore to reduce the number of migrated pages during the downtime. However, the side effect of live migrations is that they require longer, compared to non-live migration, busy time of source node.

Here, we define the busy time as the duration from the beginning of the migration to the time that migrated applications can be killed at source node.[1] As far as we know, all prior works of post-copy focused on total migration time. However, we would like to make another point here that the busy time might be more important because it has direct impacts on the system performance; it decides when computing resources, such as CPU and memory, occupied by migrated applications to be released and be reallocated to remaining applications in source node. For example, when applications are suffering from swapping due to the lack of memory in a "hot" node, simply migrating an application with large memory footprint might be able to stop (after busy time) all remaining applications at source from swapping and therefore to improve the overall system performance.

In this work, we consider that migration and checkpoint both can greatly benefit from FAM and NVM. Although the non-live migration techniques can be easily improved with FAM by the removal of the copy phase (step 2), the optimal post-copy page fault handler, however, requires

[1] Note: after migration completes, the migrated applications would continue to execute at target node.

significant redesign. This new page fault handler should rely on the both non-volatility and shareability of FAM to provide the optimal (shortest) busy time of source node as well as total migration time. In particular, we introduce FAM-aware, checkpoint-based, post-copy live migration, which checkpoints the entire application to FAM in the beginning and transfers all pages through FAM, rather than through the network connection, which the traditional migration techniques use.

In our new page fault handler, the performance of checkpoint to FAM (or NVM) plays a key role toward the busy time of source. While some prior work has already studied exploiting NVM in CPR, unlike these works which are predominantly high-level, architectural improvements [53, 54, 55, 56, 57], we explore instruction level and accessing methods when optimizing system-level checkpoint with NVM.

We observe that checkpoint and migration both have very similar spacial access patterns. In particular, they both typically write data to the storage devices only once and there is a high chance that the written data will not be read at all (checkpoint for fault tolerance) or will only be read from other nodes (migration). Due to this spacial pattern and the conclusions from prior work [39], which claims that the main performance penalty of writing to NVM does not come from the longer-than-DRAM access latency, but comes from making data persistent (i.e. costly cache line flushing operations), we find that instead of using cache flushing mechanisms, like `clwb`, `clflush`, or `clflushopt`, employing `ntstore` (non-temporal store) instruction [58] might be more suitable for checkpoint and migration programs.

We further investigate two main access approaches when utilizing `ntstore` instructions on NVM: file system `write` (batching) and memory mapped file (i.e. `mmap`). From our evaluations, we conclude that the batching is more suitable for writing larger portions of data, and `mmap` is better for smaller data sets. This is because checkpoint only dumps data once, so one `write` only costs a single expensive system call to write all data without any extra page faults. On the other hand, the `mmap` requires no system call at all, but results in a page fault per page.

We also examine employing FAM as a large shared, distributed NVM pool and exploit batching (write to FAM) and the memory mapped file (read from FAM) methods as our primary mechanism to transfer data and eliminate a lot of networking protocol overhead in post-copy. Our results show that we have improved the total migration time by at least 15%, the busy time by at least 33%, the down time by at least 23%, and migrated application performs at least 12% better during migration.

The contributions of this Chapter are as follows:

- Propose a new, checkpoint-based, page fault handler for post-copy migration using FAM to provide the best busy time and better total migration time.
- Evaluate the different access approaches of writing to NVM and propose a batching write mechanism for NVM checkpoint and migration.
- Implement our FAM-aware post-copy migration and batching NVM-aware checkpoint on a Linux open source checkpointing tool, CRIU (Checkpoint/Restore in Userspace).
- Evaluate our enhancements of CRIU with synthetic and realistic (YCSB+REDIS) workloads and show significant performance improvement.

The remainder of this Chapter is organized as follows. Section 3.2 describes the background and related work. Section 3.3 presents the motivation and design overview of our new FAM-aware post-copy live migration and NVM-aware checkpoint. Section 3.4 explains the implementation of our work by modifying and enhancing existing CRIU in more detail. Section 3.5 presents our results of evaluation of post-copy live migration and checkpoint using some macro benchmarks and realistic workloads. Finally, section 3.6 concludes.

3.2 Background

3.2.1 Migration

Traditionally, in non-live migration of applications, at first the migration program needs to stop the applications. It then checkpoints their state (mostly as files) into local storage devices, copies these checkpointed data from local storage devices to the remote storage devices (at the target machine), and finally restarts/resumes them back to the checkpointed state at the target. The downtime is mostly proportional to the amount of migrated memory.

Unlike non-live migration, live migration means that application is (or appears to be) responsive during the entire migration process. There are two main categories of live migrations:

- Pre-copy [59]: Pre-copy transfers all pages at the first round, and only the dirty pages since the previous round are sent at the following rounds until the number of dirty pages is smaller than some threshold or the number of rounds reaches some threshold. During the last round, all dirty pages and the processor state, register, etc., are sent.

- Post-copy [60, 61]: Post-copy transfers the processor state, register, etc., to target and resumes immediately at the target host. When resumed application accesses some pages which have not yet been transferred, page faults are triggered, and those pages are retrieved from source node through network.

Usually because every page is sent only once, the post-copy has better total migration time than that of pre-copy. Although post-copy has the almost minimum downtime, it would suffer from the network page fault overhead, and therefore degrade the migrated application performance during migration. Also, post-copy usually requires the longer migration time (which depends on page access pattern of application), compared with non-live one.

Sahni et al. [62] proposed a hybrid approach combining pre-copy and post-copy to take advantage of both types of live migration. The post-copy could suffer less page faults if the pages of the working set have already been sent by the pre-copy phase. Ye et al. [63] investigated the migration efficiency under different resource reservation strategies, such as CPU and memory reserved at target or source nodes. CQNCR [64] considered an optimal migration plan to migrate massive virtual machines in the data center by deciding a migration order to have less migration time and system impact.

These works, however, largely employ traditional networking as the only transfer media, thus they incur high access latencies. Besides, they only emphasize the total migration time, not busy time.

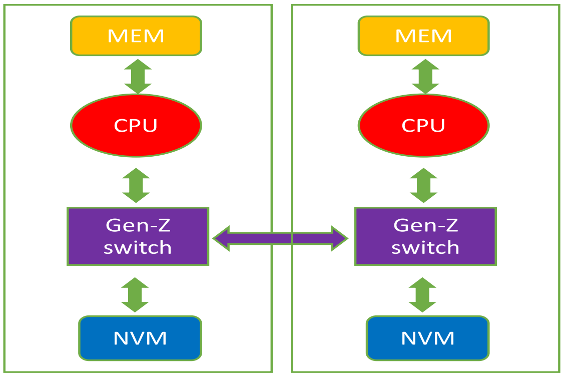


Figure 3.1: Fabric-Attached Memory.

3.2.2 Fabric-Attached Memory

Fabric-Attached Memory (FAM) is a system architecture component in which high performance networking fabrics are used to interconnect NVM between multiple nodes in a rack to create a global, shared NVM pool. In our system model, we consider a cluster consisting of many nodes, each of which contains both DRAM and NVM. DRAM can only be accessed locally by local memory controller and serves as fast, local memory in each node. NVM and the processor (in each node) are connected to a switch fabric and these switches are interconnected to each other. Here we focus on Gen-Z [11] as one such fabric because it supports hundreds gigabyte per second bandwidth and memory semantics; however, any other fabric of similar latency and bandwidth could be used. The CPU can access NVM at other nodes with memory-semantics through the fabric interface and libraries. Therefore, all interconnected NVM can be treated as the slow, global memory pool in a rack scale [12]. This byte-addressable, shared, high-bandwidth, global NVM pool is so-called Fabric-Attached Memory (FAM) in this work. Fig. 3.1 shows the overall memory, NVM, CPU, and Gen-Z interconnections.

3.2.3 Checkpoint with NVM

Checkpoint and Restart (CPR) is an important technique for fault tolerance wherein the state of a given program is saved such that it can be restarted in the event of some fault or failure. CPR also has wide applications in system management, such as application suspension and resumption, job migration, job progress saving in preemptive scheduling [65, 66, 67], instant OS update [68], load balancing, service startup boost, and debugging.

Two main types of checkpoints are as follows:

- Application-initiated (application-level): What data to be checkpointed are decided by application, so the application needs to be modified in advance. This kind of checkpoint usually dumps less data but its main drawback is that it requires significant work of modifying each application to checkpoint.

- Application-transparent (system-level): Unmodified programs can be checkpointed directly; however, nearly all the application's state needs to be checkpointed. Therefore, this kind of checkpoint usually dumps a considerable amount of data.

The emerging NVMs, such as PCM, due to their low-latency, byte-addressability, and non-volatility, have become ideal candidates for use in checkpointing. Some previous work has studied CPR exploiting NVM. Mona [53] proposed a dynamic partial checkpoint, utilizing incremental checkpoint, to checkpoint some pages before the final checkpoint. This allowed it to reduce the downtime of final checkpoint. Kannan et al. [54] proposed shadow buffering and pre-copy mechanisms to deal with the long write latency (compared to DRAM) and bandwidth limitation of PCM. Dong et al. [55, 57] proposed a local/global hybrid checkpoint technique to overcome the scalability issues of checkpoint since most of failure can be recovered locally. Phoenix [69] aggregated the bandwidth of DRAM and NVM to provide a higher total bandwidth during checkpoint and then "de-staged" the checkpoint logs from DRAM to NVM.

Leveraging the system-level checkpoint mechanism, our proposed post-copy migration can optimize busy time, close to that of non-live migration, as well as optimize total migration time. We further improve the system-level checkpoint by batching since system-level checkpoint usually has to dump a considerable amount of data, which is perfect for batching.

3.2.4 Checkpoint/Restore in Userspace (CRIU)

The Checkpoint/Restore in Userspace (CRIU) [70] is a Linux open source checkpoint tool which executes mainly in the user space, rather than kernel space. CRIU is application-transparent, i.e. applications do not have to preload some specific library or modify their source code before employing.

CRIU saves the current state of a running application, or a parent process and all its child processes, into the local storage devices and restarts the application whenever necessary. CRIU has recently received considerable attention from industrial and academic community because of the grand popularity of Docker container [47], a lightweight virtualization technique from which all containers share the same underlying operating kernel so it significantly reduces the overhead and requires much less memory footprint compared to the traditional virtual machine technique. CRIU, because its operating unit is processes, is suitable for using with Docker container and has also been integrated within Docker container now.

After checkpointing, CRIU generates a collection of files (including processor register state, memory content, fdinfo, etc.) per process to the local storage devices. Among these saved files, the biggest one is the memory content (or page image) file which contains the raw data of virtual memory areas (VMAs) of the process. To reduce the entire checkpoint time, optimizing and speeding up the dumping of page image file is the most critical and efficient. However, pages of VMAs can only be accessed by checkpointed application itself (kernel will prevent all illegal accesses from outsider). CRIU overcomes this problem by using `ptrace` system call to inject a piece of parasite code into the checkpointed application (a similar method is adopted by GDB [71], which also utilizes `ptrace` to inject software breakpoints). Through the injected parasite, CRIU daemonizes the checkpointed application and lets it begin to accept commands sent by CRIU. Hereafter, CRIU starts the checkpoint process.

The most time-consuming part of checkpoint process is to dump pages of application. The page-dumping request asks the application to execute `vmsplice` system call, which maps the pages of VMAs of the process into pipes (kernel buffers) [72]. Finally, after all pages have already

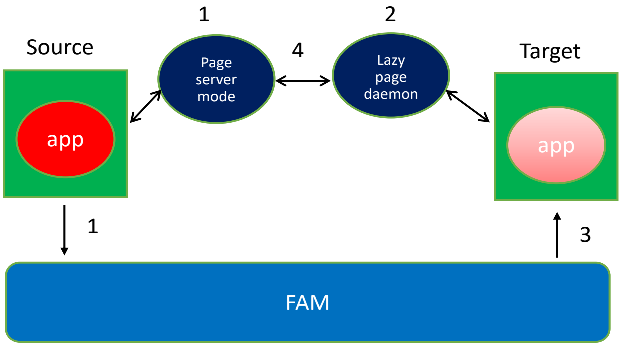


Figure 3.2: Existing CRIU post-copy using FAM. (1) CRIU checkpoints all files except for page image file to FAM, and transforms itself as a page-server. (2) At target, CRIU creates a lazy-page daemon. (3) After (1) is completed, CRIU restores application immediately at target. (4) If restored application at target accesses a page and causes a page fault, it would notify the lazy-page daemon, which then requests to and obtains from page-server at source that faulting page.

been mapped to the pipes, CRIU can access them directly (without the help of parasite) through `splice` system call, which copies dumped pages from pipes to a page image file in the storage devices.

CRIU also supports migration features, including non-live, pre-copy, and post-copy live migrations. For post-copy, the page fault handling is the main challenge. Like on-demand paging used in virtual memory systems, CRIU's post-copy employs the `userfaultfd` system call [73] to allow paging in the user space. Fig. 3.2 shows the sequential steps of the existing CRIU post-copy implementation. To achieve the post-copy, like checkpoint, at first CRIU maps the pages of VMAs of applications, at the source node, into pipes and then places itself into a page-server mode. Then a lazy-page daemon at the target node is created to handle the page fault and other events requested during the following restore operations. Finally, the migrated application is resumed at the target node. When the restored application accesses a page which still remains in the source, the application is halted temporarily, and sends the page fault request to the lazy-page daemon, which in turn communicates with page-server at the source node to obtain that faulting page from pipes through network transfer. Although all other checkpointed state can be sent through FAM. The page transfer still needs to rely on socket interface if page fault handling is not redesigned.

We note that when using pure on-demand paging, migration process may never complete, since some pages may not be ever accessed. CRIU thus employs a timer to trigger the active pushing; that is, sequentially dumping all remaining pages from source to target. The timer is kept reset whenever a page fault event happens. Before the timer is expired, the lazy-page daemon only handles the page fault event, and it would start to actively push all remaining pages only after expiration.

3.3 Motivation and System Design Overview

In this section, we describe the motivation and design overview of our optimized, FAM-aware post-copy live migration and batching system-level checkpoint.

3.3.1 Fabric-Attached Memory-Aware Post-Copy Live Migration

We propose an optimized FAM-aware post-copy migration, which exploits the properties of FAM to achieve low application[2] downtime, low total migration time, low application degradation, low resume time, and especially low busy time (of source node).

To simplify the understanding of readers, we first briefly restate some key metrics of live migration proposed by Hines et al. [61] and we further introduce the concept of busy time (of source node).

- **Preparation Time:** The time spent for the pre-copy to transfer most of its pages before migrating the state of processors. The number of rounds depends on the workloads executed. This duration is negligible for the post-copy.

- **Downtime:** The time that the application is stopped and cannot respond while the state of the processors and some pages are transferred. Pre-copy needs to transfer remaining dirty pages after the last round of preparation stage. Post-copy, depending on implementation, might transfer a few (or no) pages. Non-live migration transfers all pages here, so its downtime is the longest.

- **Resume Time:** The time from the application starts executing to the end the entire migration

[2] Note: although we only use the term "application" in this work, our approach can also be applied to virtual machines.

process. This time is mainly required for post-copy to handle the page faults happening at the target node, and is near negligible to the pre-copy and non-live migrations.

- **Busy Time (of Source Node):** The time from the beginning of the migration to the time that migrated applications can be killed and their resources (especially CPU and memory) can be released at source node. This may be the most important metric for migration since system administrators can only alleviate the loading of "hot" nodes (for load balancing) after this time.

- **Total Migration Time:** The total time including above-mentioned preparation time, downtime, and resume time. It is also the major index since the migration program (CRIU in our case) can be killed at both sides only after the total migration time. Usually the total migration time (of live migration) equals to the busy time; however, this is not the case if we employ FAM for migration. We will discuss this later.

- **Application Degradation:** The extent that application (or application in virtual machines) is slowed down during the operations of the migration. The pre-copy could affect the application performance is mainly because the need of tracking the dirty pages, which might trigger extra page faults. As to the post-copy, because it handles the page fault at target node and needs to get the faulting pages from the source node, its application degradation might be the most severe compared to pre-copy.

3.3.2 Motivation and Design

Intuitively, the performance of non-live migration, especially total migration time, could benefit from adopting the FAM simply because the copy phase can be eliminated through direct FAM accesses. Therefore, the entire migration process is now simplified as (1) checkpoint application to FAM at the source node and (2) restart application at the target node. Similarly, for pre-copy implementations employing FAM, one could simply access all the pages generated during the preparation time by FAM and therefore its copy phase can be removed.

Post-copy migration is much more complicated than the two above-mentioned methods because the most critical part of post-copy is page fault handling (or network fault as termed by Hines et al. [61]) in the target node. The behavior and implementation of the page fault handler will greatly impact the resume time (application performance degradation) and busy time. Simply removing the copy phase, if applicable, is obviously not good enough. Therefore, our work focuses on optimizing FAM-aware post-copy migration based on the following three guidelines.

Checkpoint-Based Migration: The most apparent drawback of non-live migration is its very long downtime (which equals to the total migration time). However, the busy time (of the source) of non-live migration is the best (compared to the live migrations) and is also much shorter than its total migration time because non-live migration first checkpoints everything to storage devices, and therefore, since a complete snapshot is saved in persistent media, the checkpointed application can be killed at the source.

Traditional live migrations have a long busy time (which equals to the total migration time) because they utilize memory to temporarily store pages and transfer pages by network. So, killing the application must wait until the completion of entire migration. Otherwise, if target crashes before migration completes, then the application will crash, too.

Due to the low-latency and non-volatility of FAM, storing migrated pages to FAM directly only slightly impacts performance (compared to DRAM), but it also saves the entire migrated state to persistent media. Therefore, application can be killed after being checkpointed and its busy time could be very close to that of non-live migration.

Accessing FAM as Shared Memory: Most existing live migration techniques [63, 62, 74, 64] still migrate their data content through communication network because they do not have a shared memory across multiple nodes. Therefore, their approaches will suffer the network overhead and network bandwidth. On the other hand, with the help of FAM, serving as a shared memory pool within the same rack, data could be simply migrated through memory semantics (`load/store` instructions), which bypass the significant networking protocol overhead. This could improve the critical page fault latency and therefore resume time and application degradation.

Retrieving Faulting Pages Synchronously: The page fault handler of the existing shared memory within a machine is controlled by the central operating system. The OS only needs to setup the page table of each process, then faulting pages can be mapped to virtual space of processes correctly. Our migration scheme, however, differs because a central controlling OS across multiple hosts does not exist. The FAM across machines can only be employed as a connecting media between nodes.

Besides, our page fault handler must contain two steps: first pages are written from source to FAM and then pages are read from FAM to target. So, page fault handling must be executed asynchronously through communication between the source (writing to FAM) and target nodes (reading from FAM); that is, the faulting address of pages must be sent to source first and then target must wait for the response to read that page. This communication impacts the page fault latency (even though all pages are already transferred by FAM) as well as migration performance, and must be avoided as much as possible by leveraging the information of the dumped pages. The source could notify target of the information of all dumped pages, and therefore the following faulting pages (if they have been dumped to FAM) can be accessed synchronously at target from FAM without the need of communication.

Thus, our post-copy optimization is mainly based on non-volatility and shared-ability of FAM, and can be divided into three parts.

- *to achieve the shortest downtime*, we checkpoint the processor state, registers, etc., (excluding pages of VMA) of the victim application to FAM and resume victim application at the target.
- *to achieve low busy time (of source node)*, at source, after checkpointing the necessary information, all the remaining pages continue to be dumped to FAM on the background. The victim application can be killed right after the page dumping is finished. This provides near optimal busy time, which is the checkpoint time of non-live migration.
- *to achieve low resume time and low application degradation*, all faulting pages at target will

be served/received from FAM directly and the need of asynchronous communication is also tried to minimize. Therefore, with of help of FAM, the networking overhead as well as the page fault latency can be reduced significantly.

3.3.3 Batching System-Level Checkpoint

Checkpoint performance plays an important role in the busy time of our post-copy. We propose a batching checkpoint mechanism by exploring the different accessing methods of NVM from the instruction level as well as from the file level.

3.3.3.1 Motivation

Due to the non-volatility and byte-addressability of NVM, NVM can be exploited in memory *or* storage devices, encouraging the further integration of memory and storage. This observation has been explored in several prior works [28, 34, 20]. Some prior work has shown that directly dumping data from DRAM to NVM can improve the performance of checkpoint significantly. Although their results are promising, these works focus on high level software [53, 54, 55, 57, 69] and hardware [56] architecture improvements and enhancements. Broadly they do not consider how NVM is accessed to gain the most performance. In particular, due to the non-volatility of NVM, accessing NVM should be different from accessing DRAM. Therefore, those works may result in only suboptimal solutions.

Inspired by prior work [43, 39], we find that persisting data to NVM requires costly order enforcement, which not only can provide persistency [42] of the written data, but is also particularly important toward application checkpointing because checkpoint program must make sure that checkpoint is finished only after all dumped data have already safely arrived at non-volatile storage media (NVM in our case).

Modern CPUs employ cache flushing (`clflush`, `clflushopt`, and `clwb`) and memory fence (`sfence` and `mfence`) instructions to enforce write ordering. For example, `clflush` combined with `mfence` or `sfence` is adopted by several prior works [8, 14, 13, 10]. To order writes/stores to NVM, after `stores` are issued, written cache lines should be flushed and a memory barrier is placed at the end of those flushing instructions to ensure that all the memory access operations issued after the barrier will not proceed until all store instructions received before the barrier have completed[3].

Some prior work [20, 10, 7, 39] replaces the store-clflush pair with non-temporal store (ntstore) [58] instructions, to enforce writing to NVM. The ntstore instructions force the stored data to bypass the cache and be directly written to DRAM or NVM. By such means, ntstore improves the NVM persisting performance significantly by eliminating the need for expensive cache line flush operations [39]; however, as a side effect, ntstore deteriorates the read-after-write performance due to the nature of cache bypassing[4]. That is, all writes are not be cached.

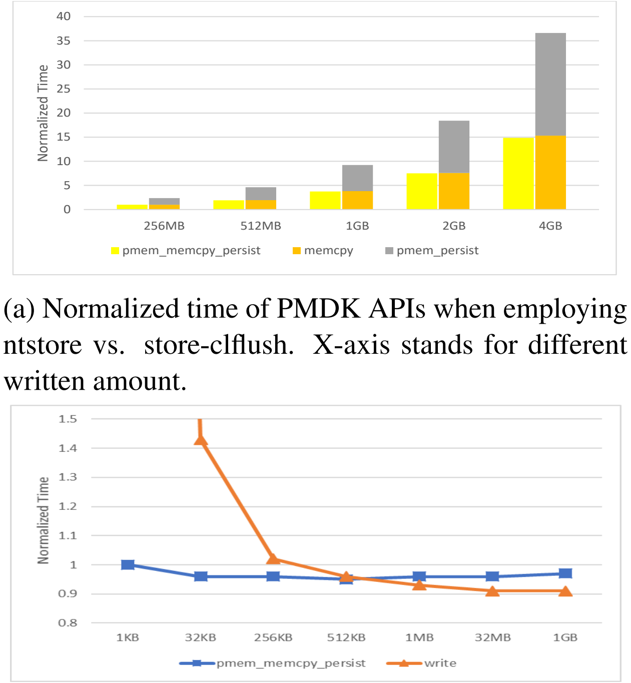
3.3.3.2 Evaluation of NVM accessing methods

Before proceeding, we compare the performance of ntstore with that of the store-clflush when writing to NVM. Our platform is described in section 3.4 and the libpmem library from Intel PMDK version 1.4 kit [31] is used.
Fig. 3.3(a) shows the normalized time of pmem_memcpy_persist (ntstore) and memcpy (store) combined with pmem_persist (clflush in our platform) when writing different amount of data as a new file in NVM. Surprisingly, the time of ntstore is already slightly less than store; not to mention that using store relies on clflush to persist data, which takes even longer time.

Next, we further evaluate two essential accessing methods of utilizing ntstore instructions; in particular, we examine the performance difference of ntstore between file system write command (batching) and memory mapped files (pmem_memcpy_persist) when accessing new files like checkpointing. Since the emulated NVM is mounted as an EXT4 file system with direct access (DAX) option enabled, all write commands through EXT4 will automatically issue

[3] Note: since checkpoint only emphasizes store instructions, so we limit ourselves to employ sfence throughout the whole work.

[4] Note: only clwb instruction does not throw data out of cache; the clflush and clflushopt both invalidate all the corresponding cache lines so their read-after-write operations do not have performance benefits (might be even worse because they pollute all cache lines and then invalidate them immediately) compared to ntstore.



(a) Normalized time of PMDK APIs when employing ntstore vs. store-clflush. X-axis stands for different written amount.

(b) Normalized time of memory mapped file vs. file system write

Figure 3.3: The comparison of different writing methods to NVM.

`ntstore` instructions [75]. The main difference of these two methods is that the file system `write` command incurs an expensive system call, but it can handle all the following memory management and requires no further context switching at all. On the other hand, the memory mapped file method costs no system call, but it will trigger a page fault (as well as context switching) per page.

Fig. 3.3(b) shows the normalized total execution time when dumping a 2GB file in NVM with 2 methods, and the x-axis stands for the written data amount per function call. For example, `write` writes 32MB each time until 2GB is finished. The time of (1KB, `write`) is 9.05 and is not shown in the figure. From the result, we can find out that the batching can outperform the mmap accessing form around 7% when enough amount of data is written at once.

From above experiments, we can conclude that: (1) the checkpoint performance of `ntstore` is much faster than that of the `store-clflush` by eliminating the time-consuming ordering op-

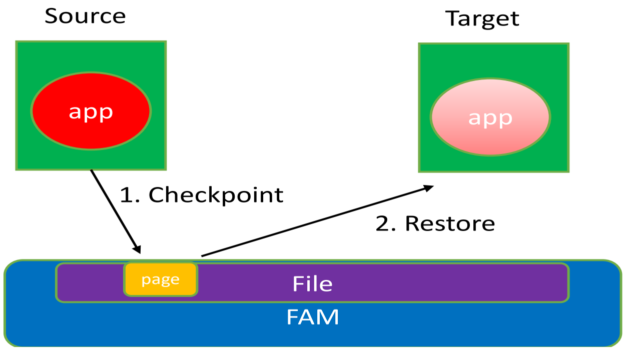


Figure 3.4: Ideal migration method using FAM.

erations. (2) Utilizing the file system `write` command enabling DAX is more suitable for writing larger data ($>$ 512 KB) at the cost of a single system call, so it might be perfect for the application-transparent checkpoint mechanism. On the other hand, writing by `ntstore` instructions through memory mapped files performs better when dumping smaller amount of data at a time. Therefore, it might be more appropriate for an application-initiated checkpoint.

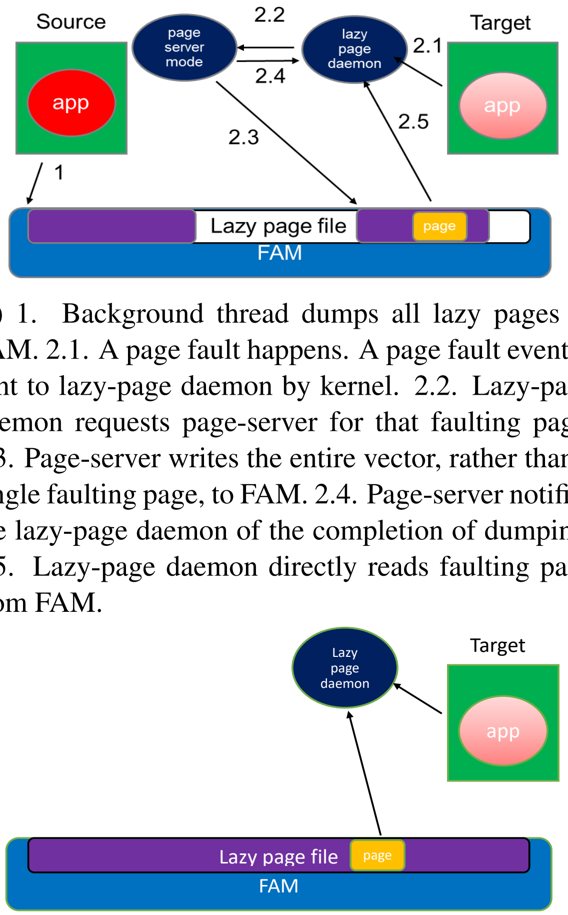
3.4 Implementation

In this section, we explain our implementation of FAM-aware, checkpoint-based, post-copy live migration and batching checkpoint in detail.

The existing CRIU [70] checkpoint and migration tool is used as a baseline implementation, and augmented with our FAM-aware post-copy and batching checkpoint techniques in this work. In particular, our case study implementation is based on modifying CRIU-dev branch version 3.2. Although our implementation is based upon CRIU, the techniques developed are universal and may be easily implemented in other checkpoint and migration schemes.

3.4.1 FAM-Aware Post-Copy Live Migration

Fig. 3.4 shows an ideal migration between nodes with FAM. First, an empty file is created in FAM, and the migration program at target node can `mmap` this whole file and directly read dumped pages without having to wait until the whole file is dumped from the source node. Ideally, (if the future can be predicted,) the source node would write a certain page to FAM each time before



(a) 1. Background thread dumps all lazy pages to FAM. 2.1. A page fault happens. A page fault event is sent to lazy-page daemon by kernel. 2.2. Lazy-page daemon requests page-server for that faulting page. 2.3. Page-server writes the entire vector, rather than a single faulting page, to FAM. 2.4. Page-server notifies the lazy-page daemon of the completion of dumping. 2.5. Lazy-page daemon directly reads faulting page from FAM.

(b) The page-server and application at source are killed after all lazy pages have been dumped. All remaining pages can be directly/synchronously accessed from target.

Figure 3.5: FAM-aware post-copy live migration.

the page is needed at target node. From the perspective of migration, which means that not only the restored application does not have to wait for the completion of page-dumping (that is, live migration), but it also avoids much of the network transmission (required by the existing CRIU and other previous implementations) through memory-semantics.

We have implemented the concept shown in Fig. 3.4 in our optimized post-copy migration in CRIU. Fig. 3.5 illustrates the main difference between our post-copy design versus the existing CRIU implementation (in Fig. 3.2).

Like the existing CRIU post-copy, we also employ a lazy-page daemon and page-server mode in our implementation. To migrate, first all required data are checkpointed as files to FAM except for the page image file, as the existing CRIU does. After that, CRIU at source enters a page-server mode, and launches a background thread to dump (checkpoint) all remaining "lazy" pages to FAM as a single lazy-page file (per process) (arrow with number 1 in Fig. 3.5(a)). Without having to wait for this background thread to finish its dumping job, system administrator can launch a lazy-page daemon and then can begin to restore the migrated application at the target node. To improve the checkpoint performance by batching (in section 3.3.3.2) and to avoid too long critical latency of page fault, we partition the VMAs of application as several I/O vectors with the maximum size of 2MB. Each I/O vector contains pages with contiguous virtual addresses. Therefore, the background thread writes the pages to FAM with at most 2MB of data at a time. When the restored application encounters a page fault, it sends the page fault event to the lazy-page daemon, which in turn sends page fault request to page-server. If the faulting page has not been dumped to FAM, the page-server waits for background thread to finish the dumping of current I/O vector, stops the background thread (by a spin lock), and dumps a specific I/O vector containing the requested faulting page. After dumping that (2MB) I/O vector, the page-server acknowledges page fault request and resumes the background thread. Arrows with number 2.1 to 2.4 in Fig. 3.5(a) illustrate this process. Alternately, if the faulting page has been dumped, the page-server acknowledges immediately.

The responses (arrow with number 2.4 in Fig. 3.5(a)) sent by the page-server not only indicate the "completion of dumping" of faulting pages, but they also contain some extra information for lazy-page daemon: the background thread's dumping progress (the largest virtual address of dumped pages) and the virtual address space of this entire (2MB) dumped I/O vector. The lazy-page daemon employs such information to construct a lookup table. (Actually, we build an LRU linked list). If the following faulting pages whose addresses can be found at the lookup table or are smaller than the dumping progress, they can be read from FAM synchronously without requesting to page-server. This can eliminate a lot of communications and overheads between source and target.

Once the background thread dumps all the lazy pages, the page-server actively notifies the lazy-page daemon of the completion of the checkpoint. Hereafter, the lazy-page daemon can synchronously read all faulting pages from FAM without any communication with page-server. Both migrated application and page-server can be killed at the source now as Fig. 3.5(b) shows.

Our implementation has some advantages: (1) The page-server only needs to handle faulting pages until the checkpoint of all "lazy" pages is finished. After that, all pages can be synchronously read from FAM. A lot of network transmissions of pages and protocol overhead can be eliminated. (2) The information of dumped pages is utilized to further reduce the communication between target and source nodes. It significantly minimizes the page fault latency and therefore improves performance. (3) After the background thread finishes dumping the lazy files, the application and page-server can be killed and their resources can be released; that is, the busy time would be much shorter than total migration time.

3.4.2 Batching Checkpoint

Based upon the discussion section 3.3.3.2, since CRIU typically generates considerable amounts of dumped data, we expect that employing `write` system calls should be higher performance than alternate techniques. Our optimized batching checkpoint is achieved by modifying the injected parasite code, which replaces the `vmsplice` with the `writev` system call; by doing so, CRIU directly dumps the content of virtual memory of checkpointed victim from its parasite to the NVM and avoids accessing kernel buffers (`vmsplice` and `splice`). In addition, unlike `write`, the `writev` supports the "struct iovec" data structure and does not require the written virtual memory address space to be contiguous. Thus, we can dump as much data as possible with one single system call. The `writev` automatically issues `ntstore` instructions after mounting EXT4 file system on top of NVM with "-o dax" option.

This implementation not only has the benefit of employing `ntstore`, as described in section 3.3.3.2, but it also has an extra advantage: A single `write` system call can dump up to 0x7ffff000 bytes (i.e. 2GB) [76]. The existing CRIU uses one pair of `vmsplice-splice` system calls per 2MB data to prevent the failure of pipe allocation. Therefore, our improvement can eliminate up to 2047 system calls per 2GB data dumped.

3.5 Evaluation

In this section, we experimentally evaluate the performance improvement of our optimized batching checkpoint and FAM-aware, checkpoint-based, post-copy live migration.

Our platform contains 20GB DDR3-1600, 12GB NVM (emulated with DRAM [77]), and Intel i7-4770 four-core 3.4 GHz processor with hyperthreading enabled. Linux 4.15.0 is used in our platform.

3.5.1 Performance of Batching Checkpoint

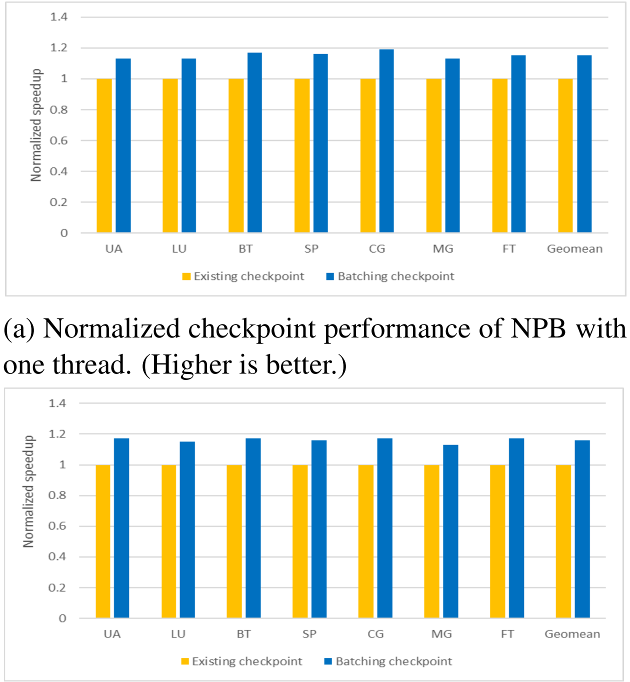
The NAS Parallel benchmarks (NPB) 3.3.1 [78] class C are used to evaluate the performance of our checkpoint scheme. Here each benchmark is checkpointed after running ten seconds. Fig. 3.6 (a) and (b) show the normalized checkpoint performance with one thread and four threads. The benchmarks are arranged according to their dumped memory size, increasing from left to right (from 475MB to 5.1GB). The results show that our optimized batching checkpoint scheme is better for all NPB benchmarks, with a performance improvement averaging 15.12% and 15.99% for one and four threads respectively.

Fig. 3.6 also indicates that the checkpoint time depends mostly on dumped memory size and is not related to workload behavior too much. That is because `ptrace` is employed by CRIU and would freeze the checkpointed victim before starting to dump; therefore the page access patterns of workloads have little impact on checkpoint process.

3.5.2 Performance of FAM-aware Post-Copy Live Migration

3.5.2.1 Workloads and Experimental Setup

To examine the performance of our FAM-aware post-copy migration scheme, we leverage the PARSEC 3.0 [79] and SPLASH-2X [80] (SPLASH-2X is SPLASH-2 with enlarged native input) benchmark suites, in addition to NPB. NPB Class C as well as PARSEC and SPLASH-2X native inputs are applied, and all workloads are migrated after executing twenty seconds, with the



(a) Normalized checkpoint performance of NPB with one thread. (Higher is better.)

(b) Normalized checkpoint performance of NPB with four threads. (Higher is better.)

Figure 3.6: The comparison of batching and existing CRIU checkpoint performance.

exception of “NPB IS” which migrated after five seconds for one thread and two seconds for four threads, and “SPLASH-2X radix” is migrated after five seconds for four threads.

We further examine the migration of REDIS [81] to investigate application performance degradation during live migration. REDIS is an in-memory data structure store and can be employed as database or in-memory cache. Through loading YCSB (Yahoo! Cloud Serving Benchmark) [40] records into REDIS, migrating REDIS to the target, and immediately accessing REDIS with YCSB at the target before the migration is finished, we can observe the impact of page fault overhead on REDIS performance with different YCSB workloads.

To evaluate FAM-aware migration, two limitations must be overcome. First, CRIU requires migrated/restored applications to use the original pid they were checkpointed with. This means application cannot be “lively” migrated within the same physical host. Next, we do not have real FAM hardware, so we do not have a global, shared NVM across physical nodes. These two limi-

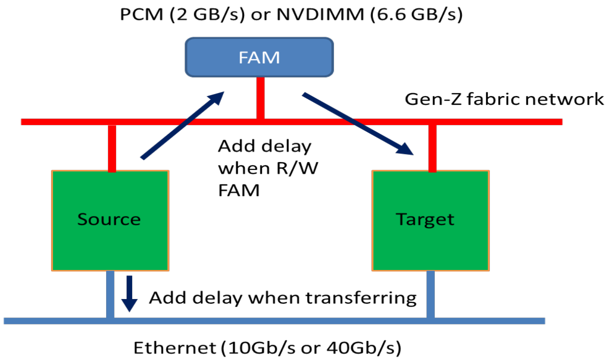

PCM (2 GB/s) or NVDIMM (6.6 GB/s)
FAM
Gen-Z fabric network
Add delay
when R/W
FAM
Source
Target
Add delay when transferring
Ethernet (10Gb/s or 40Gb/s)

Figure 3.7: The delay model of our evaluation.

tations seem to contradict with each other. Fortunately, with the help of a container virtualization technique, FAM can be emulated within a host machine by means of container and NVM: Two docker containers [47] are treated as target and source nodes; the emulated NVM, which is bind-mounted [48] into containers so applications in containers can access NVM concurrently, can be emulated as FAM in our platform.

To compare the performance of our FAM-aware with existing CRIU post-copy, we assume all process states, except for memory pages, are migrated through FAM, so post-copy migration contains only 2 steps: checkpoint from source container and restart at target container. However, the methods of page transfer are different: one is by FAM, the other is through socket interface.

Furthermore, since NVM (FAM) is emulated from DRAM, to correctly emulate the “slow” NVM, we use the same method proposed by Volos et al. [19] to add extra delay (via busy waiting) when accessing slow FAM (or Ethernet). Fig. 3.7 illustrates our delay model. The delay is added both when writing to and reading from FAM. The latency resulted from Gen-Z fabric is negligible compared to that of PCM [82]. Therefore, we only consider the access latency and bandwidth of NVM (PCM or NVDIMM). The delay calculation formula is as follows:

$$Delay_{R/W} = Latency_{R/W} + (data/bandwidth)$$

The parameters of NVM (and 10 Gigabit Ethernet used later) are shown at Table 3.1. The

Table 3.1: Parameters for extra delay.

Media	Read lat. (us)	Write lat. (us)	BW. (GB/s)
PCM	1	1	2
10Gb Ethernet	0	0	1.25

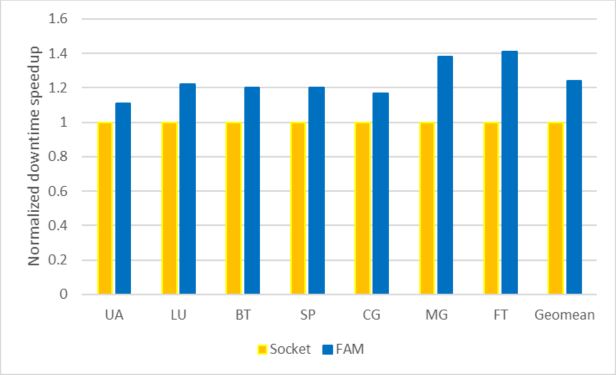


Figure 3.8: The comparison of downtime performance. (Higher is better.)

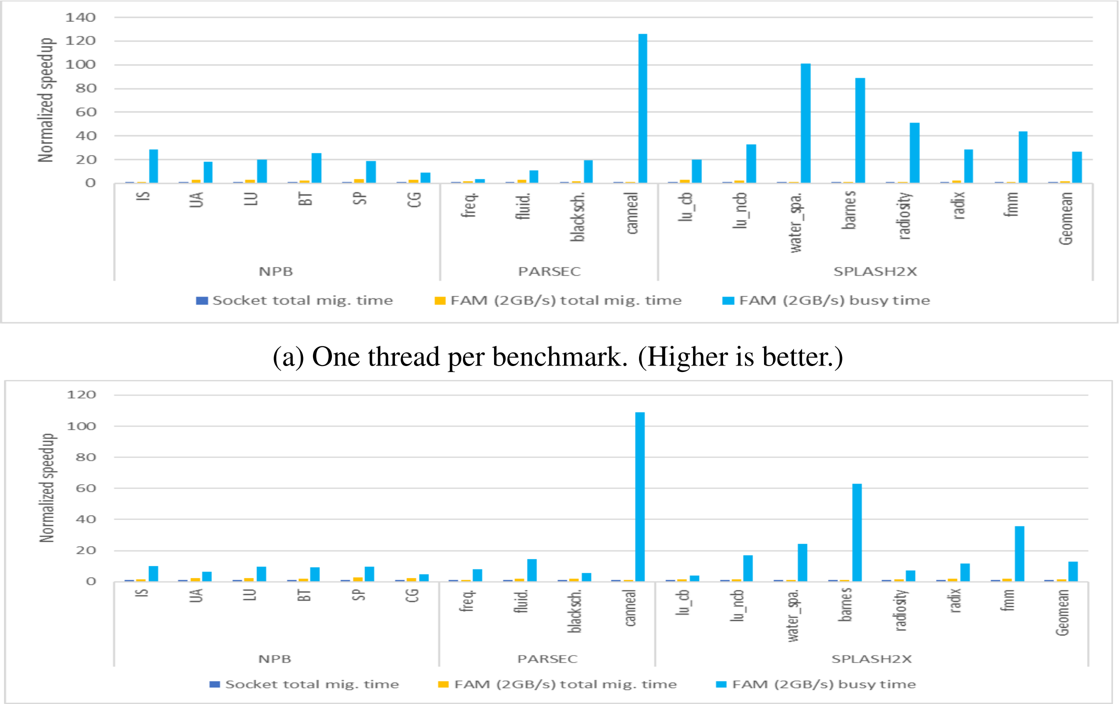
bandwidth of NVM (PCM) is assumed to be 2GB/s [83, 54]. Therefore, to access a single 4KB page from FAM, we have to wait around 3us and then can access FAM.

3.5.2.2 Evaluation Results

Fig. 3.8 shows the normalized downtime performance. All benchmarks are migrated after twenty seconds of execution. Our FAM-aware post-copy performs better at all benchmarks (23.73%) and improves more as the migrated data increases (MG: 3.4GB; FT: 5.1GB). The reason is that during migration, existing CRIU post-copy must map all pages into pipes for later page transfer by `vmsplice` system calls. Thus, when more pages need to be migrated, more system calls incur. Alternately, FAM-aware post-copy uses background thread to dump pages directly to FAM so it bypasses the system calls entirely.

Fig. 3.9 (a) and (b) show the normalized total migration time and busy time performance of our FAM-aware vs. existing[5] CRIU post-copy with one and four threads. All the measurements are

[5] Note: for existing CRIU post-copy, the busy time equals to the total migration time, so only total migration times are shown.



(a) One thread per benchmark. (Higher is better.)

(b) Four threads per benchmark. (Higher is better.)

Figure 3.9: The busy time and total migration time performance of FAM-aware and existing CRIU post-copy migration using FAM. Expiration time is 100ms.

normalized to the total migration time of the existing post-copy (socket) of the same benchmark. FAM and socket (in Fig. 3.9) stand for the mechanism of page transfer by FAM and by socket (network). We will keep using these terms hereafter. FAM (2GB/s) means the extra delay is added based on the 2GB/s bandwidth of PCM. The expiration time of active pushing timer is set as 100ms. Remember this timer would be reset whenever a page fault happens. We think 100ms should be a reasonable duration for workloads to access all working set pages before timer expires. So, the total migration time here should be a good indicator toward the different page fault overhead of these two mechanisms.

Tab. 3.2 summarizes the performance improvements of our FAM-aware post-copy. The number is the geometric mean of all benchmarks. From those results, we could conclude some useful observations.

First, instead of workload behavior, the busy time (also checkpoint time) of post-copy is im-

Table 3.2: Average improvements of FAM-aware post-copy vs. existing CRIU post-copy.

# of threads	Mig. time imprvmt	Busy time imprvmt
1	2.00X	26.74X
4	1.63X	12.72X

pacted mostly by the dumped memory size. Thus, they are relatively small compared to total migration time.

Second, the improvements of total migration time (2X and 1.63X) come from faster demand paging handling, proving that our FAM-aware migration incurs less page fault overhead.

Third, the improvements of both total migration time and busy time reduce as the number of threads increases. For migration time, that is mainly because of the available bandwidth. We limit the FAM bandwidth as 2MB/s (PCM). On the other hand, although the existing CRIU acquires pages from source through socket, we do not apply any bandwidth limitation on it. Docker containers, in a single host, utilize Linux bridge component and bypass the NIC (network interface card) to communicate with each other. To estimate the bandwidth between docker containers, we use iperf3 [84] tool to measure and get the average 7.0 GB/s throughput (compared to 2GB/s at FAM-aware case). So, we could conclude that the reduction of improvements of total migration time when more threads are executed comes from the bandwidth limitations of FAM.

For busy time, the busy time of FAM-aware post-copy is the checkpoint time, which is almost the same if the memory footprint does not change, regardless of the number of executing threads. As to the existing post-copy, since the busy time is the total migration time, which would be improved as the number of threads increases because more threads will access pages more quickly. Therefore, the decreasing the busy time improvement results from the total migration time improvement of existing post-copy as more threads are executed.

Fig. 3.10 shows a similar comparison of FAM-aware and existing CRIU post-copy except that the expiration time is set as 0ms. Only the workloads whose migrated memory sizes are larger than 1GB are selected. All workloads are executed with one thread and are migrated after exe-

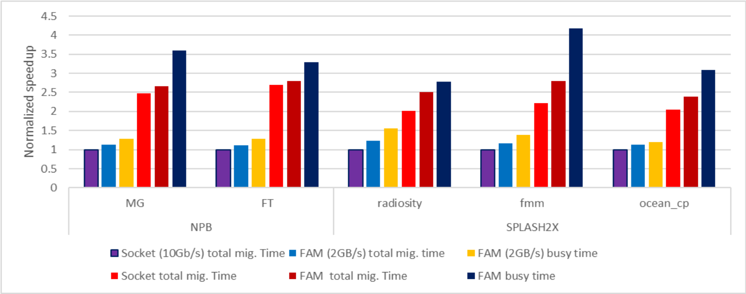


Figure 3.10: The comparison of busy time and total migration time performance with the expiration time is 0ms. (Higher is better.)

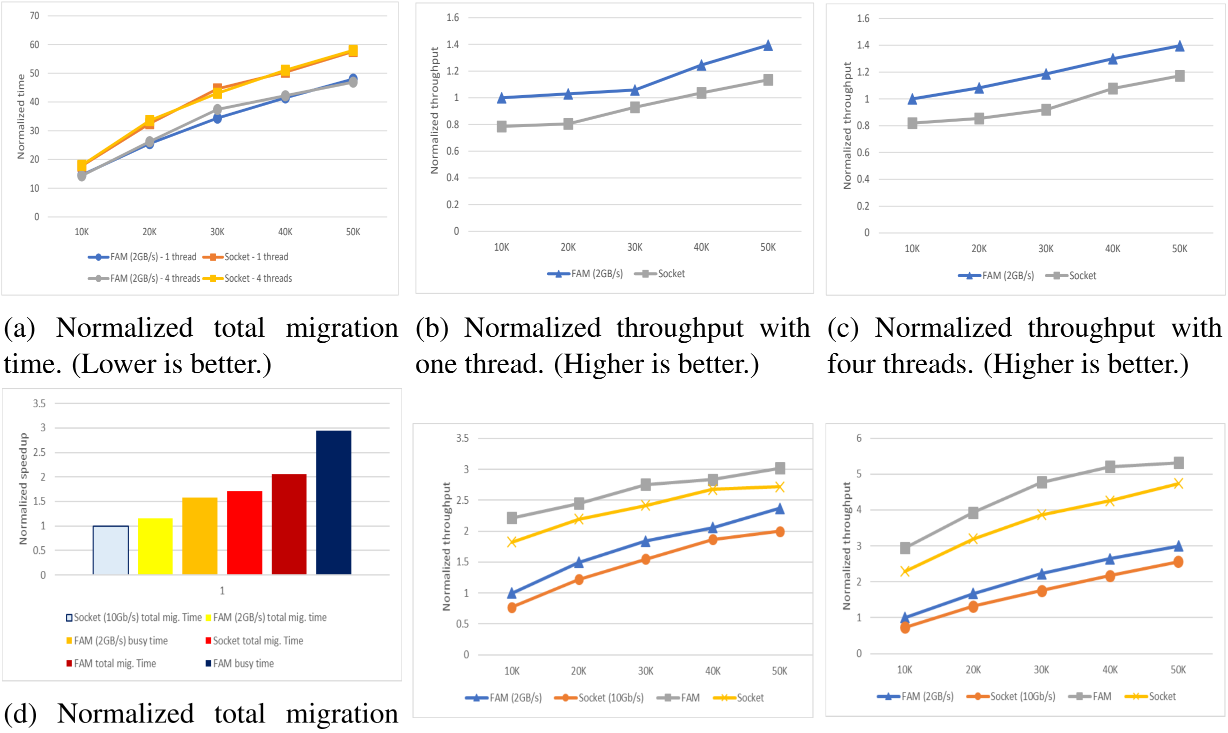
Table 3.3: Average improvements of FAM-aware post-copy for realistic case (FAM (2GB/s) v.s. socket (10Gb/s)) and ideal case (FAM v.s. socket).

	Mig. time	Busy time
FAM (2GB/s) v.s. socket (10Gb/s)	15.44%	33.68%
FAM v.s. socket	15.16%	47.08%

cuting twenty seconds. All the measurements are normalized to the total migration time of socket (10Gb/s) of the same benchmarks. Socket (10Gb/s) is assumed that the employed underlying network is 10 Gigabit Ethernet. FAM (without bandwidth limitation) means no delay is added when accessing NVM, which can be treated as the case of NVDIMM and whose performance is also the best among all cases. In our platform, the average throughput of accessing DRAM via file system `write` is around 6.6 GB/s (a little lower than 7 GB/s of socket throughput between containers).

The 0ms expiration time means that the lazy-page daemon would actively push the pages from source from the beginning. Meanwhile, if a page fault happens, the daemon will also try to handle page fault event at best effort. From Fig. 3.10, we could conclude that (A) the bandwidth provided by the transmission media dominates the migration performance; (B) if the bandwidth is close to each other, FAM-aware is better than existing post-copy due to the lighter overhead. Tab. 3.3 summarizes the improvements.

Finally, REDIS and YCSB are utilized to investigate the migration performance, especially for performance degradation. 500K records are loaded by YCSB into REDIS at the source node, and



(a) Normalized total migration time. (Lower is better.)

(b) Normalized throughput with one thread. (Higher is better.)

(c) Normalized throughput with four threads. (Higher is better.)

(d) Normalized total migration time and busy time performance. (Higher is better.)

(e) Normalized throughput with one thread. (Higher is better.)

(f) Normalized throughput with four threads. (Higher is better.)

Figure 3.11: Migration performance of REDIS accessed by YCSB. (a) to (c): The expiration time is 150ms; (d) to (f): The expiration time is 3ms.

each record is of 100 fields and the size of each field is fixed to 10B. This configuration will result in 939MB of pages to be migrated. After downtime, YCSB accesses REDIS at target immediately by different operation records (from 10K to 50K). The YCSB workloads are all configured as uniform distribution, readallfields and writeallfields are false, and R/W ratio are 70/30. The YCSB employs one and four threads to access REDIS. Fig. 3.11 shows the results. (a) to (c) employ 150ms expiration time and (d) to (f) employ 3ms.

Fig. 3.11 (a) shows the measured total migration times normalized to the busy time of FAM (2GB/s). The busy times of one and four threads are almost the same since the same amount of data (939MB) are checkpointed. FAM-aware post-copy improves average total migration time 23.92% and 22.48% with one and four threads respectively. Fig. 3.11 (a) also indicates that the total migration time is not related to the number of threads of YCSB. The reason is that the REDIS

Table 3.4: Average REDIS throughput improvements of FAM-aware post-copy of real case (FAM (2GB/s) v.s. socket (10Gb/s)) and ideal case (FAM v.s. socket).

	1 thread	4 threads
FAM (2GB/s) v.s. socket (10Gb/s)	19.8%	25.69%
FAM v.s. socket	12.7%	21.79%

is single-threaded, so more requests (threads) from YCSB cannot make the pages of REDIS be accessed faster. Fig. 3.11 (b) and (c) show the REDIS performance during migration. FAM-aware post-copy lets the REDIS perform 22.3% and 23.4% higher with one and four threads. respectively. Fig. 3.11 (a), (b), and (c) could prove that the total migration time and application degradation have some correlations because they are impacted mostly by the latency of page fault handling if the expiration time is larger enough.

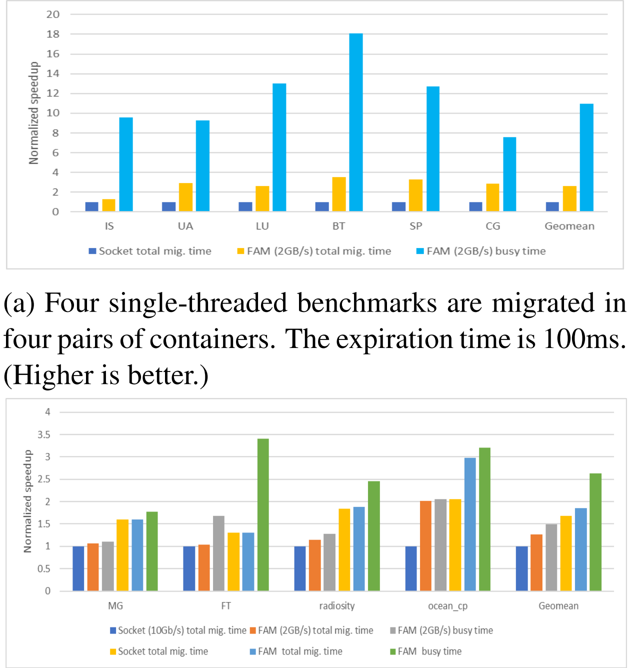
Fig. 3.11 (d) shows the normalized total migration time and busy time performance with 3ms expiration time. Because those times at different operation counts (from 10K to 50K) of YCSB are almost the same, we only take the average. This result also looks like Fig. 3.10. Tab. 3.4 summarizes the results of Fig. 3.11 (e) and (f).

Again, Fig. 3.11 (d), (e), and (f) also show that the total migration time and performance degradation are both influenced by the bandwidth of transmission media if the expiration time is too small and active pushing is triggered soon enough. The throughput of REDIS increases as the number of operations increases; this is because the chances of page fault reduce as more pages are migrated.

3.5.3 Performance of Multiple Concurrent Post-Copy Migrations

Here we would like to explore the impact of multiple concurrent migrations. Instead of only creating one pair of docker containers as a source node and a target node, we launch multiple independent pairs of containers and conduct the similar experiments as section 3.5.2.

Fig. 3.12 shows the results of multiple concurrent benchmark migrations. All benchmarks are configured to be single-threaded. Fig. 3.12 (a) shows the results that four NPB programs are



(a) Four single-threaded benchmarks are migrated in four pairs of containers. The expiration time is 100ms. (Higher is better.)

(b) Two single-threaded benchmarks are migrated in two pairs of containers. The expiration time is 1ms. (Higher is better.)

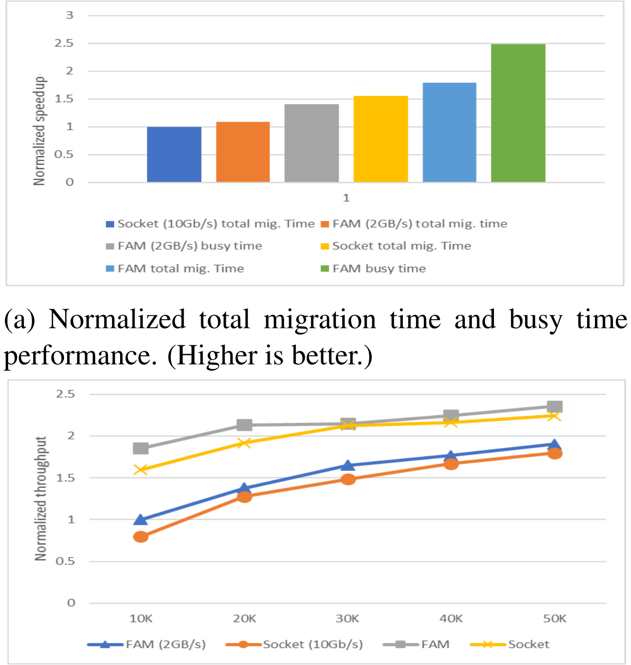
Figure 3.12: The results of multiple concurrent migrations

migrated at the same time and their outcomes are averaged. The expiration time is set as 100ms. Fig. 3.12 (b) shows the results that two programs are migrated and the expiration time is 1ms.

We also conduct the concurrent migrations of two REDIS instances in two pairs of containers. YCSB is configured to access REDIS with one thread. The expiration time is 5ms. The results are shown at Fig. 3.13. In general, the multiple concurrent migrations have little difference from single migration.

3.6 Conclusion

We presented FAM-aware, checkpoint-based, post-copy migration. Through FAM, a global, shared NVM pool in a rack scale, we can map the entire migrated memory space onto FAM. So, the data migration can be simplified as memory-semantics to achieve a much lower page fault latency path. We have implemented our prototype at CRIU, and shown that our approach has lower down time (at least 23.73%), lower busy time (at least 33%), lower total migration time



(a) Normalized total migration time and busy time performance. (Higher is better.)

(b) Normalized throughput. (Higher is better.)

Figure 3.13: The concurrent migration performance of two REDIS instances are migrated in two pairs of containers.

(at least 15%), and migrated application can perform at least 12% better (i.e. lower application degradation) during migration process.

We also presented batching system-level checkpoint, which utilizes `ntstore` instructions and file system write command to dump data into NVM. We also implemented this concept into CRIU, and showed that the checkpoint performance improves by 15%.

4. POE: PAGE FAULT HANDLING OFFLOAD ENGINE

4.1 Introduction

Virtual memory is a useful and critical technique in the modern computer systems. Virtual memory makes the applications be able to “own” the memory even more than physical available memory in the platforms. To do so, virtual memory basically relies on two other fundamental techniques: (1) lazy allocation and (2) swapping.

Lazy allocation is that the memory or page is actually allocated only when it is being accessed. By doing so, the memory subsystems of operating systems (OS) do not have to give applications any memory before applications start to execute; instead, memory is offered when memory is indeed used/needed. This (lazy allocation) can work because it is very rare for an application to touch/access all pages it requires immediately after executing. Usually the working sets of programs would be much smaller than their whole memory footprints. Based on this characteristic of programs, the memory subsystems can let multiple applications concurrently execute and therefore improves the system overall performance without running out of memory.

However, lazy allocation tries to make the memory usage as less as it can, but it is still possible that all memory is allocated if too many applications are executing concurrently in the systems. If the memory is run out, the swapping mechanism is adopted to store the content of some pages in the non-volatile storage devices and therefore those pages can be reused and re-allocated by other applications.

These two mechanisms usually work as part of the kernel page fault exception handling. When the memory is “allocated” (such as `mmap` or `malloc`) by programs, actually only a region of the virtual address space is created by kernel for the calling programs, and none of the physical pages is allocated. So, the following memory access within this new created region would trigger a page fault exception by hardware page walker, and then the (software) exception handler will check and confirm this access as legal and in turn allocates a page for it.

If the kernel exception handler cannot obtain any page for this access because of lack of memory, then exception handler will write some dirty pages (by LRU order) to storage devices and therefore make pages available for current and the following faulting accesses.

The exception handler must do a lot of operations (overheads) to handle a page fault, from checking validity of faulting address, acquiring a page from available free memory pages, filling the page with the corresponding data content from storage device (major page fault) or zeroing the page (minor page fault), to creating some data structures for the memory management. These overheads of a major page fault might not be a serious problem if the underlying storage devices are SSDs or HDDs due to their several orders of access latency compared to that of DRAM. In these cases, the context switch overhead only occupies a very small portion of entire accessing latency. However, it might not be the case when we deal with emerging Non-Volatile Memories (NVM) as storage devices.

Emerging Non-Volatile Memories (NVM), such as phase-change memory (PCM) [1], NVDIMM [2], STT-RAM [3] and 3D XPoint [4], have byte-addressability and low latency, within an order of that of main memory [85], together with the non-volatility of storage devices. These bus-attached NVMs can be seen as potential candidates of next generation of storage devices in the near future.

However, if we simply treat the NVM the same way as the traditional storage device, we will squander the benefit of much lower latency (around 5X slower than DRAM [85]) provided by NVM. For example, when a file stored in NVM is accessed and a major page fault happens, kernel does not have to block the faulting programs to trigger an I/O request for accessing the slow traditional storage devices; instead, kernel could directly use `memcpy` function to copy the data content from (bus-attached) NVM to DRAM. Further, in such cases with NVM, the context switch overhead (of page fault) can be too high, compared to the access latency of NVM, and must be further avoided.

In this work, we consider that some types of page fault overheads can be significantly reduced. In particular, we want to minimize the critical path latency of page faults resulting from accessing the memory regions created by `malloc` and anonymous private `mmap` commands. Our goal is achieved by a page pre-allocation mechanism and background thread post-page fault handling, together with the execution of enhanced hardware page walker during the page fault. The operations of our new page fault exception handling can be divided into three parts: (1) kernel (software) page pre-allocation and legal address indication; (2) hardware page walker execution at page fault; (3) kernel (software) post-page fault handling by a kernel background worker. By doing so, the critical path of page fault exception can be reduced to only a few memory accesses.

The contributions of this Chapter are as follows:

- Propose the POE, page fault handling offload engine, to reduce overheads of page fault exception generated from accessing `malloc` and anonymous private `mmap` of user programs.
- Implement POE on Linux kernel and Gem5 emulator.
- Evaluate POE using Linux kernel and show significant critical path latency improvement.

The remainder of this Chapter is organized as follows. Section 4.2 describes the background and related work. Section 4.3 presents the central design concepts of our POE hardware and software modifications. Section 4.4 explains the implementation of POE by modifying Linux kernel and Gem5 emulator in more detail. Section 4.5 presents our results of evaluation of POE with some benchmarks. Finally, section 4.6 provides a conclusion.

4.2 Background

4.2.1 Page Fault and Context Switch

In modern computer systems, virtual memory is a common technique and is employed almost in every operating system. Virtual memory is useful and helpful because it assists OS to launch multiple applications without running out of memory at once. Before the virtual memory is universally adopted, an application must allocate/obtain all needed memory before starting to execute; this limitation significantly restricts the number of concurrent executing applications in a computer system as well as the overall system throughput the computer systems can provide.

In theory, virtual memory can let applications start to execute immediately without allocating any memory for themselves at all. The memory is dispatched only when memory is really needed; this is so called lazy allocation or on-demand paging. Since memory is one of the valuable resources in the computer systems, so memory is managed and can only be dispatched by kernel.

One way user space processes can acquire memory is through the kernel page fault exception handling. When user programs need memory, they usually call `mmap` or `malloc` functions first. But `mmap` and `malloc` functions do not allocate any memory for applications when called; instead, they only create and validate a region of virtual address space for the calling processes and return the start address of this region.

Later, when applications access some address within this new created region, (if no page has been allocated for this address yet,) an exception would be triggered by hardware page walker, and the page fault exception handler starts to run in the kernel space on behalf of the faulting applications. Now, the mode of application is transferred from user mode to kernel mode. Exception handler will allocate a page for this faulting address.

After page fault exception handler completes its job, the mode is changed back to the user mode and the faulting load/store instruction is re-executed. This mode changing (from user mode to kernel mode and then back to user mode) requires some "states" (such as local variables, hardware registers, program counter, etc.) of the user mode to be stored (or pushed) into the stack. The last few operations of the exception handler is to pop those states from stack back to original places so that the faulting instruction can continue to execute. In addition to the overhead of pushing into and popping from stack, context switch might also result in some extra impacts of cache and TLB misses.

The other way memory can be allocated by applications is through the file system read/write interfaces. This method does not incur page faults, but it needs to rely on system calls, which also result in context switching. Recently, some works have observed that context switching overhead incurred by file system APIs is too high when accessing emerging storage devices, such as NVM, and they try to reduce the number of system calls. For example, Intel Storage Performance Development Kit (SPDK) [32] provides the whole NVMe driver in the user space for applications to access Ultra-Low-Latency (ULL) SSDs based on NVM. Similarly, vNVML [86] implements a user space library for applications to access bus-attached NVM. Their idea is to access the emerging storage devices (ULL SSD and NVM, respectively) from the user space as much as possible and therefore to reduce the number of system calls to improve the system performance.

On the other hand, Alam et al. [87] adopt hardware helper thread to reduce the number of context switches incurred by page fault. However, their work requires a pair of registers to indicate a single region of virtual address space (one is for start address and the other is for end address). Therefore, their approach would be better for the virtual machine workloads (because a guest OS will allocate a huge amount of contiguous virtual memory region from hypervisor as its physical memory), but might be not suitable for the general workloads.

4.2.2 Hardware Page Walker

The hardware page walker [88] is popularly employed in the modern CPUs. After TLB misses, the hardware page walker will "walk" the page table with the faulting load or store address and CR3 register in Intel x86 architectures (page directory base register (PDBR)). If the page walker can reach the lowest PTE (Page Table Entry) of the faulting address and finds the present bit (bit 0) of PTE is set; meaning that a page has been allocated for this virtual address and its physical frame number is also stored at the PTE, then page walker would simply update the corresponding TLB entry and re-execute the faulting instruction again. Otherwise, the page walker must trigger the page fault exception and let the kernel handle the page fault.

Employing the hardware page walker can reduce the number of page fault exceptions because if it, when reaching PTE, can find the present bit is sit, then page walker could directly update the corresponding TLB entry (by faulting address and page frame number found at PTE) without the intervention of the kernel.

4.3 Motivation and Design Overview

In this section, we describe the motivations and design overview of our POE (Page fault handling Offload Engine) architecture.

4.3.1 Handle Page Fault Entirely from User Space or Kernel Space?

To avoid context switching of the page fault, intuitively, there are two methods: one is the program being executed only in user space, the other is the program being executed entirely in kernel space, and in both of them since mode switching does not exist, nor does context switching.

First, can we try to execute programs and also handle page faults solely in user space? That is, we implement a user space library, which might create some new data structures in user space and also map tons of existing kernel internal data structures[1], such as struct task_struct, struct mm_struct, struct vm_area_struct, struct page, and several data structures related to swapping, into user space of the library calling process.

When a page fault happens, the hardware page walker would usually generate a page fault exception, which, instead of jumping to a usual kernel exception handler, will `call` a dedicated user space handler/function and therefore would continue to handle the page fault in the user space. This method could be implemented in almost the same way as a typical function call. Thus, after this user space exception handler finishes its page fault handling (in user space), the program can return to the faulting instruction and re-execute this instruction.

The above idea should be feasible, but it is complicated (since we have to map and maintain many data structures in user space) and it also violates the existing security policy imposed by Linux kernel. The reason why all user processes in Linux require to have both user space stack and kernel space stack is because the security policies of Linux ask that a process must utilize its user space stack (kernel space stack, respectively) when executing in user space/mode (kernel space/mode, respectively). This is actually the main reason of context switching: before mode switching, all local variables/registers/states must be saved/pushed to the corresponding stack.

[1] Note: we must map those data structures from kernel space since page fault can also happen in kernel space, so we need them to be accessible from kernel space, too.

However, user space page fault exception handler, which is still executed in user space regardless of mode switching, will violate this policy (because it has to map some kernel internal data structures into user space) and makes security even more vulnerable.

Due to the above-mentioned drawback, the only candidate left seems to be: programs should be executed in kernel space. However, it is impossible for user programs because they must launch in the user space in the first place.

Therefore, because of two straightforward methods are not viable, we consider a more complicated approach, and try to rely on the assistance from hardware since the pure software solutions are not acceptable.

A reasonable approach might be to enhance the hardware page walker, which is the only hardware having the chance to execute during the page faults, to operate part of page fault handling for kernel. This is also our proposed solution in this work. The goal of our POE approach is to coordinate software (kernel) and hardware (page walker) to complete page fault handling. With the help of enhanced page walker hardware, and by reordering the operations of page fault handling software, we can execute some operations before and after the page fault by kernel, and execute remaining operations during "real" page fault by hardware. By doing so, at the page fault only hardware is running without the intervention of software and without requiring a context switch.

Based on our proposal, we have the following design decisions to achieve our goal.

4.3.2 Page Pre-Allocation

The existing page fault exception handler always obtains a page from buddy system; this action may result in blocking the faulting program if no page is available and kernel must try even harder to get pages either from the page cache, or as a last resort, from writing some dirty pages to swap space. These two means usually require I/O requests to access backing storage devices (i.e. process will be blocked), so they cannot be simply executed by hardware.

What we consider here is to pre-allocate pages by a background kernel thread far before page faults even happen. When a page is allocated, usually a pointer of struct page (or the virtual address in kernel of that page) is returned by the buddy system in the kernel. This pointer (of struct page) can be translated into physical page frame number of that page easily by `page_to_pfn` macro. If those frame numbers of pre-allocated pages can be saved beforehand in a certain format at a page pool (here we employ a pre-allocation table per core) and the POE/hardware page walker can access them easily when page fault occurs, then the operation of page allocation of the page fault handling can be moved out of the page fault critical path.

This design choice has several benefits as follows:

- Since allocating pages might block the faulting programs, page pre-allocation by a kernel background thread only blocks this kernel thread and will not hurt (slow down) the user applications.
- In the existing Linux kernel, to allocate pages for `malloc` or anonymous private `mmap`, kernel always "zeros" the entire page (4KB) or entire huge page (2MB) in the page fault critical path for the concern of security. Our pre-allocation mechanism also moves this time-consuming operation out of the page fault critical path.
- It keeps the required hardware enhancements simple.

4.3.3 Legal Virtual Address Space Indication

A major and also the first part of the existing page fault operations is to check whether the faulting address is legal or not, i.e. the address lies within the "good_area" or "bad_area"? This checking operation is very complicated and is almost unlikely to be handled by hardware. However, the faulting virtual address can only be known at the moment that the page fault is about to happen; how can POE/hardware know it is a legal or illegal address?

It seems to be a very challenging problem, but, **by thinking in the opposite direction**, we come out a very simple solution: instead of finding out the validity of faulting address by hardware, we let software tell hardware if the faulting address is legal or not.

First, we would like to narrow down the scope of a page fault that POE can handle. We limit POE to handle the page faults happening only in the user space; in particular, our POE mechanism currently only deals with the page faults of accessing the virtual address space regions created by `malloc` and anonymous private `mmap` functions within user programs.

When user applications call `malloc` or anonymous private `mmap`, a virtual address space area is created and a pointer pointing to the start virtual address of this area is returned to applications, meaning that kernel can know which address is legal when these two functions are called. The only thing left is that we need a mechanism to inform POE of this information. Since POE/hardware page walker always walks through the page tables and corresponding page directories, such as Page Middle Directory (PMD) and Page Global Directory (PGD), of the current running processes after TLB misses. It is then straightforward that we can put an "indicator" in this page table walking path to tell POE that whether the current accessing address is legal. Page Table Entry (PTE) seems to be a reasonable candidate for accommodating this indicator.

Therefore, our proposed flow is as follows. When applications call `malloc` or anonymous private `mmap`, and before the system call is about to return, a background thread is created by kernel. This kernel background thread, like hardware page walker, continues to walk the page table path for all pages within this new created virtual address region in the background, until all PTEs[2] have been reached, and then kernel background thread sets a "POEable" bit as a legal address indication for PTEs of all pages belonging to this virtual address region.

4.3.4 Hardware Page Walker Enhancement

Our central idea is to move most of operations out of page fault critical path and to execute them before and after page fault by software, and only the mandatory operations are tackled by hardware during page fault, so we could keep POE as simple as possible by enhancing and extending the function of existing hardware page walker.

From above-mentioned design choices, we have a pre-allocation page pool containing page frame numbers of available pre-allocated pages and POEable bits, as indications, have been set at PTEs of legal address space. Therefore, the operations of POE should be straightforward: When

[2] Note: If the paths to PTE of some addresses have not been constructed yet, kernel will construct these paths as exception handler does. So, this path construction process can also be moved out of the page fault critical path.

TLB misses, POE (hardware page walker) starts to walk the page table until it reaches PTE of faulting address. If POE finds out that the POEable bit is set (meaning the faulting address is legal), but the present bit is not set yet (meaning that this address has not be allocated a page, so POE needs to allocate a page for it), then POE requests a free page from pre-allocation page pool, stores its page frame number as well as sets some flag bits into PTE as software exception handler does, refills this missing TLB entry, and continues to execute the faulting instruction without the need of kernel help.

However, for all other cases different from the above-mentioned scenario, the POE just simply triggers a typical page fault exception as existing approach and lets software (kernel) to handle it.

4.3.5 Post-Page Fault Handling by Kernel Background Thread

Besides the page allocation, the address validation, and PTE update, there are still some operations left to be executed by the page fault handling, such as increasing counter of mm object (by calling inc_mm_counter function) and put page into LRU list for swapping later. Those operations can be delayed and executed later by another periodic background kernel thread if programs are not terminated, and we will actively scan and process the unprocessed pages in pre-allocation tables only if programs are being terminating. Thus, this part can also be moved out of page fault critical path.

4.4 Implementation

In this section, we explain our implementation of POE hardware and corresponding kernel components in detail. We first discuss the case of single-threaded applications, and address the case of multi-threaded applications at the end of this section.

4.4.1 POE Enable and Disable

To enable POE, applications should call the `POE_enable` system call in the beginning of their source code, and no other extra change is required to use POE. The POE will be automatically disabled by `POE_disable()` function called inside __mmput() function by kernel when applications are about to be killed.

We have also implemented the POE_disable system call, which would directly call POE_disable() function and allows applications to disable POE if necessary[3].

4.4.2 POE_Enable System Call

The POE_enable system call sets the POE_enable, a boolean type member of struct_mm, as true for the caller's mm object. The first POE_enable caller also has to construct the pre-allocation page pool. Here, we implement this pool as a pre-allocation table per core. What we mean by "table" here is actually the physical contiguous pages allocated from kernel's buddy system. How many pages should be allocated (to construct a single table of a core) depends on how many pages we want to pre-allocate/be contained in a table, which is configurable. To shorten the time spent in POE_enable system call, only the pages to construct pre-allocated tables are allocated here, and the pre-allocated pages themselves contained within these tables are allocated later by a background thread.

After that, schedule_on_each_cpu() kernel function is executed to set the page frame number (34 bits) of the first page of the pre-allocation table, the number of entries of pre-allocation table (16 bits), and POE_ENABLE bit (1 bit) to a new 64 bits register (here, the CR9 control register is employed in this work) per core in order to enable POE hardware of all cores. In this work, the numbers of entries of all pre-allocation tables are configured as sixty-four, but they (entry number per table) are configurable and different numbers can be applied from table to table.

Finally, at the end of the POE_enable system call, a kernel thread (here we implement it with Linux work queue) is triggered to pre-allocate all pages, in the background, for all pre-allocation tables of all cores. Because allocating pages per core could be very time-consuming, considering platforms with hundreds of cores, adopting a kernel background thread can avoid blocking the execution of the first POE_enable caller program. Although it is possible that the POE hard-

[3] Note: 1. POE can be implemented as a kernel module and operations of POE_enable and POE_disable system calls can be executed when the POE module is loaded and unloaded, respectively. By doing so, all user applications will automatically enable POE by default. However, we prefer to use system calls method here because it makes debug much easily (we can limit POE users within some caller applications, rather than all user applications in the systems). 2. What we mean by "POE is disabled" is that the new created virtual address spaces, after POE is disabled, would not support POE, so page faults in those regions must be handled by typical kernel exception handler. However, all previous POE enabled regions are still "POEabled."

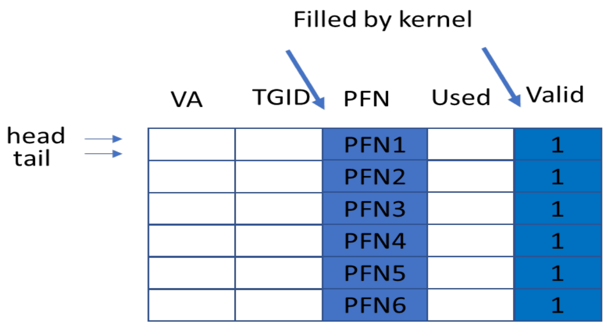
ware may request (pre-allocated) pages from some pre-allocation tables even before the above-mentioned asynchronous kernel thread is executed and has a chance to allocate any page, it is still fine since the POE can always trigger a typical page fault exception and can hand over the task to kernel to handle this page fault if POE could not obtain pages from the corresponding pre-allocation table.

4.4.3 Pre-Allocation Table

Our pre-allocation table adopts a lockless ring buffer architecture [89] with one producer (the kernel, which produces/pre-allocates pages) and one consumer (the POE, which consumes/requests pages). Each entry of pre-allocation table has sixteen bytes, and the first entry (entry number 0) stands for the table header, which contains the head index (ranging from one to the number of entries), tail index (also from one to the number of entries), number of table entries, and locks, and the size of each of them is four bytes. Except for the table header, all the entry's format is delineated in Fig. 4.1. Fig. 4.1 shows the pre-allocation table and its fields, which contains faulting virtual address, TGID (thread group id), page frame number, used bit, and valid bit.

The producer only looks at the head index, and the consumer only checks the tail index. That is, when kernel (producer) wants to pre-allocate a page, first it looks up the corresponding entry of the head index from table header. If this entry is not valid (the valid bit is not set), meaning that this entry does not contain a valid pre-allocated page, then kernel allocates a page and put its page frame number into the corresponding field of this entry, increments the head index by one, and checks the next entry. Kernel would continue this page pre-allocation process until it reaches an entry whose valid bit has already been set. This also means that all entries of this table are valid.

Since a table is dedicated to a core, so page pre-allocation (by kernel) can follow some policies/rules. For example, if a platform contains multiple NUMA (non-uniform memory access) nodes, then pages can always be pre-allocated from near memory first. When near memory is exhausted, we can decide either to pre-allocate pages from far memory, or not to pre-allocate pages at all. After all pages (from near memory) are used from this table, POE triggers a typical page fault exception and lets kernel handler to decide the next step. We would not plan to discuss this



VA	TGID	PFN	Used	Valid
		PFN1		1
		PFN2		1
		PFN3		1
		PFN4		1
		PFN5		1
		PFN6		1

Figure 4.1: The pre-allocation table and all its fields after pages are pre-allocated. To avoid confusion, the table header is not shown here.

policy further since it is out of the scope of this work.

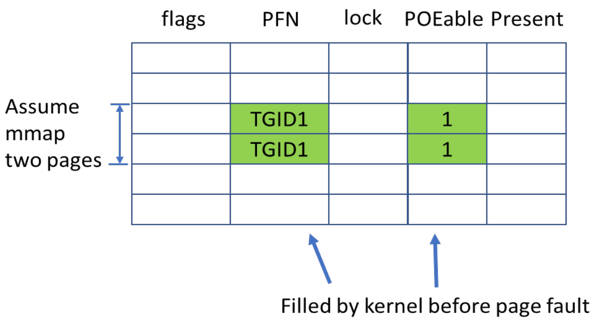
4.4.4 Pre-Page Fault Software Handling

When applications call `malloc` or anonymous private `mmap`, if kernel finds the POE_enable member of mm object is set as true (by `POE_enable` system call), then kernel will add a new VM_POE flag to the vma (virtual memory area) object (of the struct vm_area_struct) when this new vma is created.

By the end of `mmap` or anonymous private `malloc` functions, the kernel creates a background kernel thread by kthread_run() function. This thread, since it knows the start and end addresses of the newly created vma, can walk the page table of the calling process until it reaches all the corresponding lowest level Page Table Entries (PTEs); if the paths to PTEs have not been constructed yet, this kernel thread will construct them as page fault exception handler does.

When reaching a PTE, kernel thread sets the POEable bit (bit 2) and RW bit (bit 1, if the region is writeable) of PTE only if the present bit of this PTE is not set yet. Here we borrow bit 2 (the user bit) of PTE as the POEable bit, which is safe since this bit is not involved in the swap entry computation. Besides these two bits, kernel background thread also writes the TGID (thread group id) of current process into the field of PFN. Fig. 4.2 shows these pre-page fault operations.

This kernel thread is employed here because we want to execute this pre-page fault operation asynchronously to avoid blocking programs too long since this operation could be time-consuming



flags	PFN	lock	POEable	Present
	TGID1		1	
	TGID1		1	

Figure 4.2: The context of the lowest level PTEs of a POE enabled process. The `mmap` function maps two pages and sets the TGID and POEable bit.

if applications `malloc` or `mmap` a huge, say 1GB, region. Also, if POE reaches a PTE of a page which is "pre-allocatable" but its POEable bit has not been set by the kernel background thread, then POE can simply treat this page as usual and trigger a typical page fault exception.

4.4.5 Page Fault Hardware Handling

The POE, since it is an enhanced hardware page walker, only starts to execute after TLB misses. The basic operations of POE are as follows:

1. If POE cannot reach (the lowest level) PTE, then it triggers the page fault exception as usual.

2. If POE can reach (the lowest level) PTE and the present bit of PTE is set, then POE updates the TLB entry and re-executes the instruction again.

3. If POE can reach (the lowest level) PTE and the present bit of PTE is not set, and the POEable bit of PTE is not set, (meaning that this page cannot be pre-allocated,) then POE also triggers the page fault exception as usual.

4. If POE can reach (the lowest level) PTE and the present bit of PTE is not set, but the POEable bit of PTE is set, then POE stores the TGID (written by kernel at section 4.4.4) from PFN field of PTE, obtains a page (by looking up the corresponding entry indicated by the tail index and checking the valid bit of this entry is set[4]) from the pre-allocation table of the

[4] Note: If the valid bit (of tail entry) is not set, meaning that not only this entry does not contain valid page, but

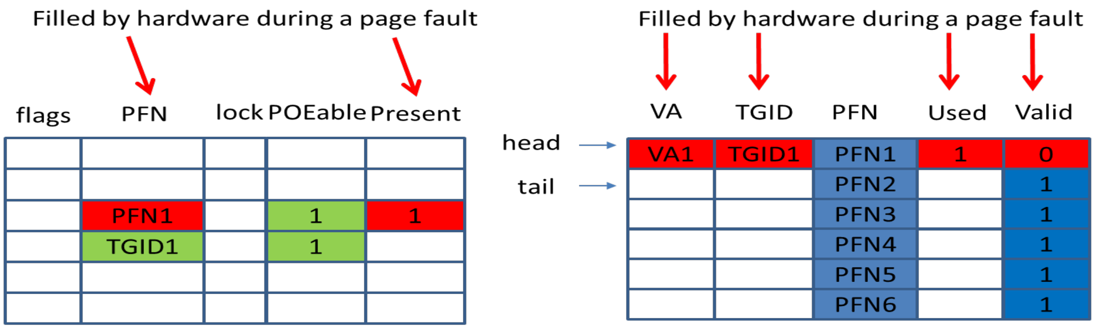


Figure 4.3: The operations of POE. The left table stands for PTEs and the right table is pre-allocation table.

current executing core, fills its page frame number into the PFN field of PTE, and sets some corresponding flags (for read: the present, accessed, and nx bits are set; for write: besides the three bits mentioned above, dirty bit and soft dirty bit are also set) into PTE[5]. The POE also updates the TLB entry, writes the faulting virtual address, stores TGID, and used bit for the tail entry of the pre-allocation table, cleans the valid bit of the tail entry, and increments the tail index by one.

Fig. 4.3 shows the operations of POE during a page fault.

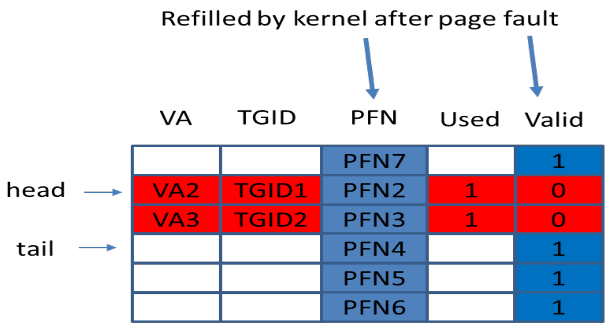
POE makes page fault critical latency quite small: except for regular memory loads to walk the page table and the PTE entry update, which are also required operations of the existing page fault exception handling, POE only incurs one four bytes load (read the tail index), one eight bytes load (read PFN field and valid bit of the tail entry), one sixteen bytes store (update the entire tail entry), and one four bytes store (update the tail index).

4.4.6 Post-Page Fault Software Handling

Linux delayed work queue is adopted to periodically execute post-page fault processing. After all pages of all pre-allocation tables are pre-allocated by a background thread triggered by `POE_enable` system call, a delayed work queue function is scheduled, with a delay timer is

also the whole pre-allocation table of this core is empty, then POE would trigger a typical page fault exception.

[5] Note: the user bit (bit 2) and RW bit (bit 1, if applicable) are already set by the pre-page fault kernel thread.



VA	TGID	PFN	Used	Valid
		PFN7		1
VA2	TGID1	PFN2	1	0
VA3	TGID2	PFN3	1	0
		PFN4		1
		PFN5		1
		PFN6		1

Figure 4.4: The operations of post-page fault handling.

set as 2ms. When timer expires, a delayed work function is executed to do the post-page fault handling.

This post-page fault handling is straightforward. First, it uses the head index to get the corresponding (head) entry. If the head entry's used bit is set, since the TGID, faulting virtual address, and page frame number can be found at the used entry, then they are enough to execute the following functions: `anon_vma_prepare`, `inc_mm_counter`, `page_add_new_anon_rmap`, and `lru_cache_add_active_or_unevictable` for a single page. After all functions are executed, the used bit is cleared.

Besides above-mentioned operations, the delayed work function is also responsible for refilling the used entries of pre-allocation tables. That is, if the valid bit is cleared, then delayed work function will pre-allocate a page, write its page frame number and set the valid bit into head entry, and increment the head index by one. The delayed work function will continue to process and refill the next entry until it meets an entry whose valid bit is set. Fig. 4.4 illustrates the operations of post-page fault processing.

If pages cannot be pre-allocated due to the lack of memory, delayed work function will still continue to do the post-page fault handling for the following used entries, without increasing the head index. After all entries are processed, the delayed work function is re-scheduled for the next time (2ms).

4.4.7 Error Handling

Since delayed work function is only executed every 2ms, what happens during this period if an application is terminated and some pages are still not processed by the delayed work function? An error handling function is implemented to handle those pages. This error handling function will scan all pre-allocation tables and finds out the used entries (with used bit is set) whose TGID is the same as the terminated application. If such entries exist, then the error handling function executes the same functions as delayed work function does. Also, since error handling function and delayed work function might execute concurrently, we need a lock to avoid race condition between them. The bit 0 of locks field in the table header is employed as a `test_and_set` lock bit here.

Because error handling function needs to scan all pre-allocation tables and it is better that we only scan whenever necessary. Therefore, error handling function is only called at the zap_pte_range() function and when the page's page_mapcount() returns zero as well as the vma's VM_POE flag is set. This can significantly reduce the frequency of calling error handling function since `page_add_new_anon_rmap` will not be called for those unprocessed pages, so the values of their _mapcount of struct page are still -1 as well as the page_mapcount function will return zero.

4.4.8 POE_Disable Function and POE_Disable System Call

The `POE_disable` function sets the caller's POE_enable member of mm object as false. The last caller of `POE_disable` first waits until the delayed work function is completed or cancelled. Then, like `POE_enable`, this last caller disables POE hardware of each core by calling `schedule_on_each_cpu()` function to clear the POE_ENABLE bit for all cores, and it also frees/releases all valid pages (whose valid bit is set) of all pre-allocation tables.

As we have mentioned at section 4.4.1, POE is implemented to be automatically disabled by `POE_disable()` function from __mmput() function by kernel when applications are terminated. So disabling POE within the source code of applications is unnecessary. But we still implement `POE_disable` system call for applications to disable POE if they want. The `POE_disable` system call directly calls `POE_disable` function. Also, a protection has been implemented so that calling `POE_disable` function twice accidentally will not harm. The second `POE_disable` system call will directly return.

4.4.9 Huge Page Support

POE can also support huge (2MB) page easily. Another pre-allocation table per core is needed to contain all pre-allocated huge pages for a core. The kernel background thread at section 4.4.4 will set the POEable bit at PMD if huge page can be allocated for certain virtual address regions. When POE reaches PMD and finds the POEable bit is set but present bit is not set, POE will obtain a huge page from the pre-allocated table.

4.4.10 Multi-Threaded Process

The previous sections all describe the interactions between POE and kernel for a single-threaded process. How about a multi-threaded process? Two or more threads might access the same page and therefore encounter the page fault (of the same page) at the same time. Furthermore, we should consider a more complicated race condition case between POE hardware and kernel software page fault exception handler.

In section 4.4.4, a kernel background thread is used to set the POEable bit for all PTEs. Consider a case, the POE hardware reaches a PTE before the background thread sets its POEable bit. Therefore, POE hardware will treat this page as “non-POEable” and triggers a page fault exception. Before the page fault exception handler executes, our kernel background thread could be scheduled and set the POEable bit of this PTE. POE hardware from another core could reach this PTE (since its present bit is not set but POEable bit is set) and execute the POE page fault handling. Meanwhile, the page fault exception handler triggered from the first core executes and starts to handle the page fault for the same faulting page. This might be a serious problem, so we have to avoid it.

To solve this problem, we employ a `test_and_set` lock at PTE. That is, when the POE hardware and kernel page fault exception handler want to service a page fault[6], they both need to

[6] Note: kernel can check the VM_POE flag from vma to decide if it needs to get the lock or not.

obtain a lock at the PTE first.

We borrow the accessed bit (bit 5) of PTE as the lock bit. So the flow of section 4.4.5 4 becomes as follows. If POE can reach the PTE and the present bit is not set, but the POEable bit is set, then POE must `test_and_set` to acquire the lock[7]. If the returned value from `test_and_set` is zero (unlock), then POE continues to proceed as section 4.4.5 4. After that, POE will clear the lock bit to unlock it.

However, if the returned value from `test_and_set` is one (lock), meaning that another POE from another core or kernel page fault exception handler is handling a page fault for the same page, then POE would busy wait here until the lock bit is clear and present bit is set (this means that page fault has been solved), and updates TLB as well as re-executes the faulting instruction.

This architecture is more scalable compared to the existing kernel exception handler approach. POE only busy waits at the page level, and this busy wait happens only when multiple POEs try to access the same page simultaneously. However, the existing kernel page fault exception handler uses a global page table lock to synchronize all page faults of the threads belonging to the same process. Therefore, all faulting threads of the same process must busy wait no matter their faulting pages are the same or not.

4.4.11 Comparison with Existing and POE Page Fault Handling

Fig. 4.5 shows the flows of existing kernel and POE page fault exceptions handling. We can find out that POE re-orders the operations and moves most of them out of the page fault critical path. Only a few operations are left for POE hardware to process.

4.5 Evaluation

Since our POE mechanism is mainly to coordinate of the enhanced hardware page walker and the kernel, we have to modify the existing CPU hardware and try to run modified Linux kernel on top of it. To achieve it, we employ Gem5 [90] emulator to modify and emulate our new POE hardware. The Gem5 platform is configured as four-core 3.4GHz X86 TimingSimple CPU with

[7] Note: Because PTE is just read and cached, so this `test_and_set` action should only access the cache and does not need to access memory.

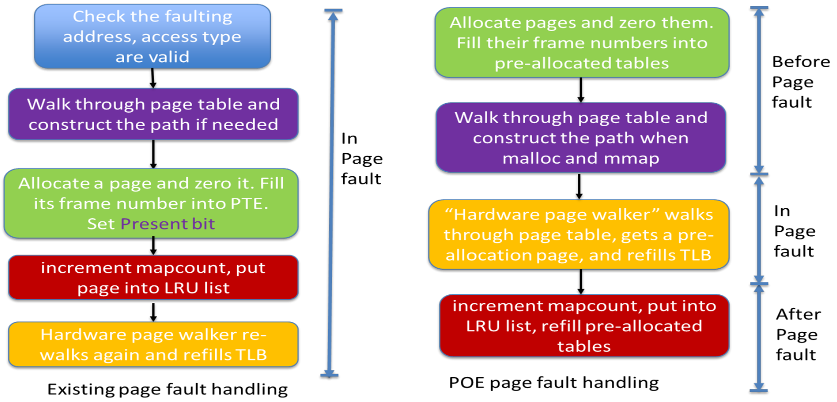

Check the faulting address, access type are valid
Walk through page table and construct the path if needed
Allocate a page and zero it. Fill its frame number into PTE. Set Present bit
increment mapcount, put page into LRU list
Hardware page walker re-walks again and refills TLB
In Page fault
Existing page fault handling
Allocate pages and zero them. Fill their frame numbers into pre-allocated tables
Walk through page table and construct the path when malloc and mmap
"Hardware page walker" walks through page table, gets a pre-allocation page, and refills TLB
increment mapcount, put into LRU list, refill pre-allocated tables
Before Page fault
In Page fault
After Page fault
POE page fault handling

Figure 4.5: Comparisons of page fault flow

8GB DDR4 main memory using Ruby[8] memory model. Gem5 full system (FS) simulation is employed and Linux kernel version 4.9.182 is modified and used in our evaluations.

4.5.1 Results of Page Fault Critical Latency

First, we want to evaluate the improvement of page fault critical path latency brought from POE. We conduct experiments inside Gem5 FS environment. We run a simple program which `mmaps` an anonymous private memory region and accesses one byte per page within this region; total 100 pages are accessed and their results are averaged and reported. We compare the read and write access latency between POE and the existing kernel page fault handler. Fig. 4.6 shows their results, from which some conclusions can be drawn.

First, we have shown that our POE mechanism works in Linux and Gem5 FS environment. The hardware and software cooperate to solve the page faults and therefore result in a very short critical path latency of page fault.

Second, the critical path latency of read and write page fault of POE are very close, less than a hundred processor cycles apart. This is because POE hardware always allocates a new page for a page fault, regardless of read or write. So the POE operations for read and write page fault handling are the same, and their latency should be the same, too.

[8] Note: As far as we know, only Ruby memory model can let Linux kernel, with multi-core X86 CPU, boot and run successfully.

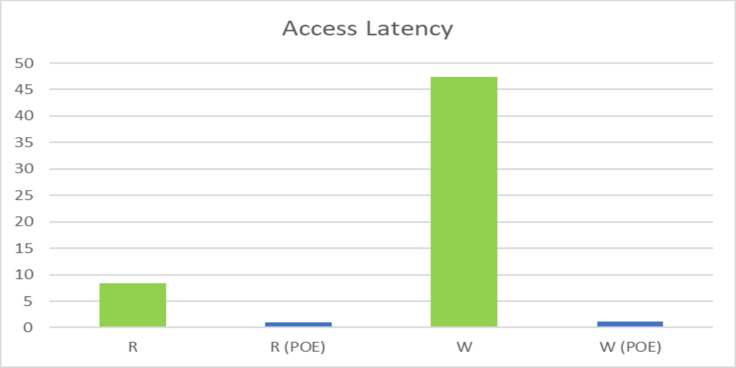


Figure 4.6: Normalized critical path cycle of a page fault of POE and existing kernel page fault handler. W stands for write page fault, and R stands for read page fault.

However, the existing kernel page fault exception handler handles page faults caused by read and write differently if page faults happen within the regions created by `malloc` or anonymous private `mmap`. Linux kernel always maps the faulting address to a special “zeroed” page for read page fault, and allocates a new page for write page fault later. Since mapping to a special “zeroed” page does not require to zero the page again, the critical path latency of read is much shorter than that of write. We can see from the Fig. 4.6 that the W (write page fault) has the worst critical path latency since it incurs both the overheads of the context switch and zeroing the page.

Third, the critical path latency of POE is much better than the existing Linux page fault exception handler in Gem5 FS environment. The read latency is improved by a factor of 8.3 and the write latency is improved by a factor of 47.3 times when compared to the traditional Linux page fault handler.

4.5.2 Results of Micro Benchmarks

In this section, we try to estimate the improvements of POE when we execute micro benchmarks. However, we do not/cannot directly run those benchmarks inside Gem5 FS because (1) Gem5 FS environment does not support all X86 instructions so many benchmarks cannot run directly in Gem5 FS mode. (2) The Ruby memory model of Gem5 is too slow and is not good enough. We conduct the same experiment described at section 4.5.1 in Gem5 and in a real machine and compare their results. For critical path latency of write page fault, the in Gem5 average

Table 4.1: Average critical path cycle counts of a page fault. POE latency is measured in Gem5 FS environment

	POE	write on real machine	read on real machine
cycle counts	431	3678	1158

39203 cycles is reported but in real machine we get an average of 15024 cycles.

Therefore, we count the number of total page faults of the benchmarks and estimate the percentage of expected improvements provided by POE on a real machine.

We use a machine with 32GB DRAM, and Intel i7-4770 four-core 3.4 GHz processor with hyperthreading enabled. Linux kernel 4.9.182 version is employed on this machine, too.

We measure the page fault critical path latency as explained in section 4.5.1 on a real machine, but we average results of 4096 pages instead of 100 pages. We execute the same (4096 pages) measurements in Gem5 to get the page fault latency when POE is enabled. Tab. 4.1 summarizes these measurements.

Next, we configure all the benchmarks as single-threaded, and measure their total execution time and the number of write and read page faults resulting from `malloc` and anonymous private `mmap`.

The numbers of page faults are then multiplied by the difference of latency of read (1158 - 431) and write (3678 - 431) cycle counts respectively and divided by 3.4G to get the time expected (in seconds), which gives us the time saved by POE. Finally, the expected time is divided by total execution time of the benchmarks to obtain the percentage of improvement from POE.

We leverage the PARSEC 3.0 [79] and SPLASH-2X [80] benchmark suites with native input sets. Fig. 4.7 shows our results, and we observe on an average 5.6% improvement.

4.6 Conclusion

We presented POE, Page fault handling Offload Engine, in this chapter. POE employs codesign of an enhanced hardware page walker and a modified Linux kernel to reduce the critical path latency of the page fault handling. We employ a kernel background thread to execute some of the

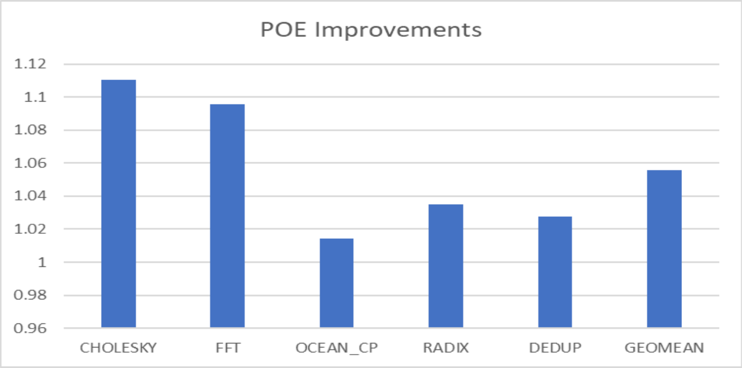


Figure 4.7: Improvement (%) provides by POE.

operations of page fault handling, asynchronous to the actual page fault and let the hardware carry out portions of the work that need to be handled at the time of the page fault.

We implemented the kernel modifications in Linux and simulated the enhanced hardware in Gem5 emulator. We have shown that our POE could significantly reduce the page fault critical path latency when virtual address regions created from `malloc` and anonymous private `mmap` are accessed.

The evaluation shows that the page fault critical latency can reduce to 2.1% for write and 12% for read of existing software exception handling times. In addition, POE can improve the execution time of some benchmarks by an average of 5.6%.

5. CONCLUSIONS

The NVM has become a promising storage device and will gradually replace the traditional storage devices, such as HDD and SSD, in the near future.

In this work, we considered the employment of NVM to optimize the emerging memory systems to boost the system performance. To do so, we proposed three approaches in three different directions.

In Chapter 2, we built a user space library, vNVML, to virtualize and share NVM in user space. Applications, by employing vNVML, could concurrently access NVM and also can "see" the capacity of NVM larger than physically available NVM. Through less than 10% overhead, vNVML can achieve much better performance compared with file system approaches.

Next, in Chapter 3 we improved application migration performance by employing FAM. We proposed checkpoint-based, post-copy live migration to achieve the shortest busy time and down time, while also offering shorter total migration time and less application degradation. We further pinpointed that, in terms of system performance, the busy time should be the most important metric, instead of the total migration time, among all the migration metrics.

Finally, in Chapter 4 we proposed POE, Page fault handling Offload Engine, to remove the overheads of context switching of page faults. Through page pre-allocation, enhanced hardware page walker, and background kernel post-page fault handling, we can eliminate the context switching from page faults in user space. We also showed that our page fault critical path latency of accessing DRAM is around 2.1% for write and 12% for read of that of the existing Linux kernel page fault handling. Also, POE is estimated to be able to improve the execution time of some benchmarks by about 5.6%.

REFERENCES

[1] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, "Phase-change technology and the future of main memory," *IEEE Micro*, vol. 30, pp. 131–141, Mar. 2010.

[2] D. Narayanan and O. Hodson, "Whole-system persistence," in *ASPLOS XVII Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, (London, England, UK), ACM, 2012.

[3] M. T. Krounbi, S. Watts, D. Apalkov, X. Tang, K. Moon, V. Nikitin, V. N. A. Ong, and E. Chen, "Status and challenges for non-volatile spin-transfer torque ram (stt-ram)," in *International Symposium on Advanced Gate Stack Technology '10*, (Albany, NY), Sept. 2010.

[4] "Intel optane technology." https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

[5] L. Liang, R. Chen, H. Chen, Y. Xia, K. Park, B. Zang, and H. Guan, "A case for virtualizing persistent memory," in *SoCC '16 Proceedings of the Seventh ACM Symposium on Cloud Computing*, (Santa Clara, CA, USA), ACM, 2016.

[6] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson, "Mojim: A reliable and highly-available non-volatile memory system," in *ASPLOS '15 Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, (Istanbul, Turkey), ACM, 2015.

[7] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson, "Strata: A cross media file system," in *SOSP '17 Proceedings of the 26th Symposium on Operating Systems Principles*, (Shanghai, China), pp. 460 – 477, ACM, 2017.

[8] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *EuroSys '14 Proceedings of the Ninth European*

Conference on Computer Systems, (Amsterdam, The Netherlands), ACM, 2014.

[9] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "NV-Tree: Reducing consistency cost for NVM-based single level systems," in *FAST '15 Proceedings of the 13th USENIX Conference on File and Storage Technologies*, (Santa Clara, CA, USA), USENIX, 2015.

[10] J. Xu and S. Swanson, "Nova: A log-structured file system for hybrid volatile/non-volatile main memories," in *FAST '16 Proceedings of the 14th USENIX Conference on File and Storage Technologies*, (Santa Clara, CA, USA), pp. 323 – 338, USENIX, 2016.

[11] "Gen-Z specitications." https://genzconsortium.org/specifications/.

[12] K. Keeton, "Memory-driven computing," in *FAST '17*, (Santa Clara, CA), USENIX Association, 2017.

[13] X. Wu and A. L. N. Reddy, "Scmfs : A file system for storage class memory," in *SC '11 Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, (Seattle, Washington, USA), ACM, 2011.

[14] S. Qiu and A. L. N. Reddy, "Nvmfs: A hybrid file system for improving random write in NAND-flash SSD," in *MSST '13 IEEE 29th Symposium on Mass Storage Systems and Technologies*, (Long Beach, CA, USA), IEEE, 2013.

[15] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *SOSP '09 Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, (Big Sky, Montana, USA), ACM, 2009.

[16] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and durable data structures for non-volatile byte-addressable memory," in *FAST '11 Proceedings of the 9th USENIX Conference on File and Storage Technologies*, USENIX, 2011.

[17] SNIA, *NVM Programming Model*. Storage Networking Industry Association, 2017. Rev. 1.2.

[18] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile

memories," in *ASPLOS XVI Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, (Newport Beach, California, USA), pp. 105 – 118, ACM, 2011.

[19] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *ASPLOS XVI Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, (Newport Beach, California, USA), pp. 91–104, ACM, 2011.

[20] E. R. Giles, K. Doshi, and P. Varman, "Softwrap: A lightweight framework for transactional support of storage class memory," in *MSST '15 Proceedings of the 31st Symposium on Mass Storage Systems and Technologies*, (Santa Clara, CA, USA), pp. 1–14, IEEE, 2015.

[21] A. Memaripour, A. Badam, A. Phanishayee, Y. Zhou, R. Alagappan, K. Strauss, and S. Swanson, "Atomic in-place updates for non-volatile main memories with kamino-tx," in *EuroSys '17 Proceedings of the Twelfth European Conference on Computer Systems*, (Belgrade, Serbia), pp. 499–512, ACM, 2017.

[22] K. Doshi, E. R. Giles, and P. Varman, "Atomic persistence for scm with a non-intrusive backend controller," in *HPCA '16 Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, (Barcelona, Spain), IEEE, 2016.

[23] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *ISCA '09 Proceedings of the 36th annual international symposium on Computer architecture*, (Austin, TX, USA), ACM, 2009.

[24] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *ISCA '09 Proceedings of the 36th annual international symposium on Computer architecture*, (Austin, TX, USA), ACM, 2009.

[25] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: Closing the performance gap between systems with and without persistence support," in *MICRO '13 Proceedings of the*

46th Annual IEEE/ACM International Symposium on Microarchitecture, (Davis, CA, USA), IEEE, 2013.

[26] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, “Atom: Atomic durability in non-volatile memory through hardware logging,” in *HPCA ’17 Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, (Austin, TX, USA), IEEE, 2017.

[27] J. Jung, Y. Won, E. Kim, H. Shin, and B. Jeon, “Frash: Exploiting storage class memory in hybrid file system for hierarchical storage,” *ACM Transactions on Storage (TOS)*, vol. 6, no. 1, 2010.

[28] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift, “Aerie: Flexible file-system interfaces to storage-class memory,” in *EuroSys ’14 Proceedings of the Ninth European Conference on Computer Systems*, (Amsterdam, The Netherlands), ACM, 2014.

[29] V. Fedorov, J. Kim, M. Qin, P. V. Gratz, and A. L. N. Reddy, “Speculative paging for future NVM storage,” in *MEMSYS ’17 Proceedings of the International Symposium on Memory Systems*, (Alexandria, Virginia), 2017.

[30] D. Watts, “Intel optane dc persistent memory product guide,” 2019. https://lenovopress.com/lp1066-intel-optane-dc-persistent-memory.

[31] “Intel persistent memory development kit.” https://pmem.io/pmdk/.

[32] Intel, “Storage performance development kit,” 2017. https://spdk.io/.

[33] Samsung, “Ultra-low latency with samsung z-nand ssd,” 2017. https://www.samsung.com/semiconductor/global.semi.static/Ultra-Low_Latency_with_Samsung_Z-NAND_SSD-0.pdf.

[34] L. A. Eisner, T. Mollov, and S. Swanson, “Quill: Exploiting fast non-volatile memory by transparently bypassing the file system,” UCSD CSE Tech. Rep. CS2013-0991, University of California, San Diego, San Diego, CA, 2013.

[35] M. Swift, "Persistent memory ordering," 2015. http://materials.dagstuhl.de/files/15/15021/15021.MichaelSwift1.Slides.pdf.

[36] H. Wan, Y. Lu, Y. Xu, and J. Shu, "Empirical study of redo and undo logging in persistent memory," in *NVMSA '16 Proceeding of the 5th Non-Volatile Memory Systems and Applications Symposium*, (Daegu, South Korea), pp. 1–6, IEEE, 2016.

[37] A. Chen, "A review of emerging non-volatile memory (NVM) technologies and applications," *Solid-State Electronics*, vol. 125, pp. 25–38, 2016.

[38] S. Yu and P.-Y. Chen, "Emerging memory technologies: Recent trends and prospects," *IEEE Solid-State Circuits Magazine*, vol. 8, no. 2, pp. 43–56, 2016.

[39] Y. Zhang and S. Swanson, "A study of application performance with non-volatile main memory," in *MSST '15 Proceedings of the 31st Symposium on Mass Storage Systems and Technologies*, (Santa Clara, CA), IEEE, 2015.

[40] B. F. Cooper, A. Silberstein, ErwinTam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *SoCC '10 Proceedings of the 1st ACM symposium on Cloud computing*, (Indianapolis, Indiana, USA), pp. 143–154, ACM, 2010.

[41] "Transaction isolation levels." https://docs.microsoft.com/en-us/sql/odbc/reference/develop-app/transaction-isolation-levels?view=sql-server-2017.

[42] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *ISCA '14 Proceeding of the 41st annual international symposium on Computer architecuture*, (Minneapolis, Minnesota, USA), pp. 265–276, ACM, 2014.

[43] C. Wang, Q. Wei, J. Yang, C. Chen, and M. Xue, "How to be consistent with persistent memory? an evaluation approach," in *NAS '15*, (Boston, MA), pp. 186–194, IEEE, Aug. 2015.

[44] S. Swanson, "A vision of persistence." https://www.sigarch.org/a-vision-of-persistence/.

[45] "Mongodb." https://github.com/mongodb.

[46] “Mmapv1 storage engine.” https://docs.mongodb.com/manual/core/mmapv1/.

[47] “Docker container.” https://www.docker.com/.

[48] “Docker container bind mounts.” https://docs.docker.com/storage/bind-mounts//.

[49] S. Borkar, “Designing reliable systems from unreliable components: The challenges of transistor variability and degradation,” *IEEE Micro*, vol. 25, pp. 10–16, 2005.

[50] B. Schroeder, E. Pinheiro, and W.-D. Weber, “Dram errors in the wild: A large-scale field study,” in *SIGMETRICS ’09*, (Seattle, WA), pp. 193–204, ACM, June 2009.

[51] K. Ye, D. Huang, X. Jiang, H. Chen, and S. Wu, “Virtual machine based energy-efficient data center architecture for cloud computing: A performance perspective,” in *GREENCOM-CPSCOM ’10*, pp. 171–178, IEEE, Dec. 2010.

[52] K. Chanchio and X.-H. Sun, “Communication state transfer for the mobility of concurrent heterogeneous computing,” *IEEE Transactions on Computers*, vol. 53, pp. 1260–1273, 2004.

[53] S. Gao, B. He, and J. Xu, “Real-time in-memory checkpointing for future hybrid memory systems,” in *ICS ’15*, (Newport Beach, CA), pp. 263–272, ACM, June 2015.

[54] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic, “Optimizing checkpoints using nvm as virtual memory,” in *IPDPS ’13*, (Boston, MA), IEEE, May 2013.

[55] X. Dong, N. Muralimanohar, N. Jouppi, R. Kaufmann, and Y. Xie, “Leveraging 3d pcram technologies to reduce checkpoint overhead for future exascale systems,” in *SC ’09*, (Portland, Oregon), ACM, Nov. 2009.

[56] J. Xie, X. Dong, and Y. Xie, “3d memory stacking for fast checkpointing/restore applications,” in *3DIC ’10*, (Munich, Germany), IEEE, Nov. 2010.

[57] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi, “Hybrid checkpointing using emerging nonvolatile memories for future exascale systems,” *ACM TACO*, vol. 8, July 2011.

[58] Intel, “Intel store intrinsics.” https://software.intel.com/en-us/node/524244.

[59] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *NSDI '05*, (Berkeley, CA), pp. 273–286, USENIX, May 2005.

[60] E. R. Zayas, "Attacking the process migration bottleneck," in *SOSP '87*, (Austin, TX), pp. 13–24, ACM, Nov. 1987.

[61] M. R. Hines, U. Deshpande, and K. Gopalan, "Post-copy live migration of virtual machines," *ACM SIGOPS Operating Systems Review*, vol. 43, pp. 14–26, July 2009.

[62] S. Sahni and V. Varma, "A hybrid approach to live migration of virtual machines," in *CCEM '12*, (Bangalore, India), IEEE, Oct. 2012.

[63] K. Ye, X. Jiang, D. Huang, J. Chen, and B. Wang, "Live migration of multiple virtual machines with resource reservation in cloud computing environments," in *CLOUD '11*, (Washington, DC), IEEE, July 2011.

[64] M. F. Bari, M. F. Zhani, Q. Zhang, R. Ahmed, and R. Boutaba, "CQNCR: Optimal VM migration planning in cloud data centers," in *IFIP '14*, (Trondheim, Norway), IEEE, June 2014.

[65] J. Li, C. Pu, Y. Chen, V. Talwar, and D. Milojicic, "Improving preemptive scheduling with application-transparent checkpointing in shared clusters," in *Middleware '15*, (Vancouver, BC, Canada), pp. 222–234, ACM, Dec. 2015.

[66] B. Cho, M. Rahman, T. Chajed, I. Gupta, C. Abad, N. Roberts, and P. Lin, "Natjam: Design and evaluation of eviction policies for supporting priorities and deadlines in mapreduce clusters," in *SoCC '13*, (Santa Clara, CA), ACM, Oct. 2013.

[67] Y. Shao, W. Bao, X. Zhu, W. Xiao, and J. Wang, "Chord: Checkpoint-based scheduling using hybrid waiting list in shared clusters," *Journal of Systems and Software*, vol. 131, pp. 22–34, Sept. 2017.

[68] S. Kashyap, C. Min, B. Lee, T. Kim, and P. Emelyanov, "Instant os updates via userspace checkpoint-and-restart," in *ATC '16*, (Denver, CO), pp. 605–619, USENIX, June 2016.

[69] P. Fernando, S. Kannan, A. Gavrilovska, and K. Schwan, “Phoenix: Memory speed hpc i/o with nvm,” in *HiPC ’16*, (Hyderabad, India), IEEE, Dec. 2016.

[70] “Checkpoint/Restore In Userspace (CRIU).” https://www.criu.org/.

[71] “GDB: the GNU Project debugger.” https://www.gnu.org/s/gdb/.

[72] “Linux pipe.” http://man7.org/linux/man-pages/man2/pipe.2.html.

[73] “Linux userfaultfd.” http://man7.org/linux/man-pages/man2/userfaultfd.2.html.

[74] C. Jo, E. Gustafsson, J. Son, and B. Egger, “Efficient live migration of virtual machines using shared storage,” in *VEE ’13*, (Houston, TX), pp. 41–50, ACM, Mar. 2013.

[75] “Direct access for files.” https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[76] “Linux write.” http://man7.org/linux/man-pages/man2/write.2.html.

[77] “Emulated nvdimm in linux.” https://nvdimm.wiki.kernel.org/.

[78] D. Baliley, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, “The NAS parallel benchmarks-summary and preliminary results,” in *SC ’91*, pp. 158–165, ACM, 1991.

[79] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The PARSEC benchmark suite: characterization and architectural implications,” in *PACT ’08*, (Toronto, Ontario, Canada), pp. 72–81, ACM, Oct. 2008.

[80] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, “The SPLASH-2 programs: Characterization and methodological considerations,” in *ISCA ’95*, (S. Margherita Ligure, Italy), pp. 24–36, ACM, June 1995.

[81] “Redis: An in-memory data structure store.” http://redis.io/.

[82] “Gen-z overview.” https://genzconsortium.org/wp-content/uploads/2018/05/Gen-Z-Overview.pdf.

[83] “Phase change memory.” http://www.pdl.cmu.edu/SDI/2009/slides/Numonyx.pdf.

[84] “iperf - the ultimate speed test tool for tcp, udp and sctp.” https://iperf.fr/iperf-download.php.

[85] R. Peglar, “The future of storage systems a dangerous opportunity,” in *MSST ’19 Proceedings of the 35st Symposium on Mass Storage Systems and Technologies*, (Santa Clara, CA), IEEE, 2019.

[86] C. C. Chou, J. Jung, A. L. N. Reddy, P. V. Gratz, and D. Voigt, “vNVML: An efficient user space library for virtualizing and sharing non-volatile memories,” in *MSST ’19 Proceedings of the 35st Symposium on Mass Storage Systems and Technologies*, (Santa Clara, CA), IEEE, 2019.

[87] H. Alam, T. Zhang, M. Erez, and Y. Etsion, “Do-it-yourself virtual memory translation,” in *ISCA ’17*, (Toronto, ON, Canada), ACM, June 2017.

[88] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, “Accelerating two-dimensional page walks for virtualized systems,” in *International conference on Architectural support for programming languages and operating systems ’13*, 2008.

[89] “Lockless ring buffer design.” https://www.kernel.org/doc/Documentation/trace/ring-buffer-design.txt.

[90] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, “The gem5 simulator,” *ACM SIGARCH Computer Architecture News*, vol. 39, pp. 1–7, May 2011.