2024

# The Crash Consistency, Performance, and Security of Persistent Memory Objects

Derrick Alex Greenspan
*University of Central Florida*

# THE CRASH CONSISTENCY, PERFORMANCE, AND SECURITY OF PERSISTENT MEMORY OBJECTS

by

DERRICK GREENSPAN
M.S. University of Central Florida, 2019
B.S. University of Central Florida, 2016

A dissertation submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Department of Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida

Spring Term
2024

Major Professors: Mark Heinrich (Advisor) & Yan Solihin (Co Advisor)

# ABSTRACT

Persistent memory (PM) is expected to augment or replace DRAM as main memory. PM combines byte-addressability with non-volatility, providing an opportunity to host byte-addressable data persistently. There are two main approaches for utilizing PM: either as memory mapped files or as persistent memory objects (PMOs). Memory mapped files require that programmers reconcile two different semantics (file system and virtual memory) for the same underlying data, and require the programmer use complicated transaction semantics to keep data crash consistent.

To solve this problem, the first part of this dissertation designs, implements, and evaluates a new PMO abstraction that addresses these problems by hosting data in pointer-rich data structures without the backing of a filesystem, and introduces a new primitive, psync, that when invoked renders data crash consistent while concealing the implementation details from the programmer via shadowing. This new approach outperforms a state-of-the-art memory mapped design by $3.2\times$ depending on the workload. It also addresses the security of at-rest PMOs, by providing for encryption and integrity verification of PMOs. To do this, it performs encryption and integrity verification on the entire PMO, which adds an overhead of between $3 - 46\%$ depending on the level of protection.

The second part of this dissertation demonstrates how crash consistency, security, and integrity verification can be conserved while the overall overhead is reduced by decrypting individual memory pages instead of the entire PMO, yielding performance improvements compared to the original whole PMO design of $2.62\times$ depending on the workload.

The final part of this dissertation improves the performance of PMOs even further by mapping userspace pages to volatile memory and copying them into PM, rather than directly writing to PM. Bundling this design with a stream buffer predictor to decrypt pages into DRAM ahead of time improves performance by $1.9\times$.

This dissertation is dedicated to the memory of my cousin,

Jeffrey Allen Boedeker (06/14/1982 - 10/03/2023).

*“I carry your heart with me. (I carry it in my heart)” – E. E. Cummings*

# ACKNOWLEDGMENTS

I have been using computers since before I could read. As a child, I had always wanted to be a doctor of some kind; and by the time I reached high school at Crooms Academy, I admired the famous computer scientists I learned about, and how they had left their mark on a field that was in their infancy. I wanted more than anything to emulate them, and I knew that I wanted to dedicate my life to the advancement of computer science as a discipline. So when I was admitted to UCF for the Fall 2012 term, there was no question what my end goal was going to be, and I embarked on the long journey towards my PhD.

A full list of people who helped on this journey would result in this section being as long as the rest of this dissertation. But to my mom and dad, Theresa Ann and Daniel Greenspan, *thank you*, for giving me a place to stay while working towards my degree, always being willing to help me, for being my strongest ally and confidant, for raising me, for giving me my faith, sense of humor, intelligence, and for your marriage being a model of true love. Words cannot express how thankful I am for all that you have done for me throughout my whole life. You two are truly the greatest parents ever, and I love you so much!

Thank you, to my brother, Chad and his girlfriend, Anna, who has constantly been a source of support. And thank you to my Grandmother, Elsie Greenspan, for all the years that you helped take care of me, helping me with my spelling quizzes, picking my brother and me up from school, and for making Thursday, not Friday, my favorite day of the week! I love you so much!

To the rest of my family: Aunt Lisa and Uncle Jeff, Aunt Linda and Uncle Wayne, Uncle Gordy and Aunt Delores, Grandma Linda, and to my cousins Matt, Sarah and Marsha, Christopher, Jessica, my late cousin Jeffrey, and to their partners: thank you for all you have done for me, and I love you all! Similarly, thank you to my late Great Grandmother Jean, my late Grandmother May Jean

Houser, and Jean Boedeker, all whom I miss dearly; and to my late Grandfathers Merrill Boedeker and Samuel Greenspan: although they passed before I could remember them, their legacy shines through my parents.

To all of the teachers and administrators during my time in K-12 education who helped me (of which there are too many to completely list), thank you! I know that I was a challenging student, but I am thankful for every teacher's support. If not for Dr. Collins, Mrs. Autorino, Ms. Roberts, Mr. Barsalou, Ms. Spears, Ms. Gooch, Mrs. Broome, Ms. Solomon, Mr. McCrory, and all the other faculty, admin, and staff, I don't think I would have even started on this journey.

To all of the people I know through St. Stephen's Catholic Community, particularly Caryl and her late husband, Jim, Father John and Father George, Zynep, Judy, Rosalie, Ali, Dr. Gonzalez, Mary, and so many others, thank you!

Of course any acknowledgment is incomplete without mentioning the committee. To Dr. Mark Heinrich, Dr. Yan Solihin, my chair and co-chair respectively, and to Dr. Paul Wiegand and Dr. Paul Gazillo: thank you! To the members of the ARPERS lab, particularly Dr. Mustafa, Ardhi, Shaikhul, Mansour, Rian, and Ali, thank you!

To all the professors I met throughout my many years here at UCF (a full enumeration of which would take far too long): Dr. Daly, Dr. Gerber, and all of the people that I've met throughout the years that I worked at IST, including (but not limited to) Drs. Hughes, Dieker, Xynidis, Ingraham, Kider (who directed me to Dr. Heinrich!), Walters (who encouraged me to follow my interest), Michlowitz, Hynes, and Martin: thank you!

To my friends; thank you for putting up with me for all this time. Michael, Justin, Magda, David, Devon, Maria, Derino, Ryan, Kiera, Nicoli, Kenneth, Andrew, the list goes on and on: thank you so much for being a part of my life. Words cannot express my gratitude.

To the senior design team: Andres, Chris, Connor and Samu; thank you! I could not have done this dissertation without your help.

Finally, I just want to recognize what a long and interesting PhD journey this has been. I started more than seven years ago, back in January 2017, but even if I had known at that time how long this journey would have taken, I still would have done it. I have loved every moment of my time here at UCF and would not trade it for anything. I am quite aware how well and truly blessed I am. It is true the cliché that "if one does what they love, they will not work a day in their life". Even through all the times that I would spend super late nights in the lab, leaving at the crack of dawn, I have loved every moment! To all who have encouraged me or who have contributed to my success on this journey, including anyone who I might have inadvertently omitted in this acknowledgment, thank you!

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# CHAPTER 1: INTRODUCTION

Among the earliest memory technologies in computing was magnetic-core memory, which unlike modern main memory today, is a form of persistent memory (PM). Main-memory lost its persistence in the 1970s, when DRAM overtook magnetic-core memory to become the near-exclusive technology for primary memory in computers. However, as scaling DRAM chips to smaller processor nodes has become more difficult, PM has found its place in the computing hierarchy once again. Modern PM technologies, such as Spin Transfer Torque (SST-RAM) [40], Phase Change Memory (PCM) [80], and Intel Optane PMem [13][1] are poised to augment or perhaps even replace DRAM outright, as PM is non-volatile and dense, and has performance closer to DRAM than block-based storage such as Solid State Drives (SSDs) and Hard Disk Drives (HDDs). Since it is non-volatile, it has the capability to host persistent data directly in main-memory, avoiding expensive serialization/deserialization calls.

Figure 1.1 (from [25]) illustrates a hypothetical memory system, with PM (NVRAM) occupying a space between traditional block based storage and volatile main-memory. Since PM has performance comparable to that of DRAM, it follows that abstractions for using PM should take advantage of both its persistence and its speed. However, existing PM abstractions are forced to duplicate effort by keeping two separate systems, the filesystem and virtual memory, consistent for the same underlying data [38], or alternatively abandon any type of abstraction by treating PM as a large volatile memory pool [1]. Clearly, another abstraction is needed.

To fill the gap more effectively, this dissertation proposes the Persistent Memory Object (PMO) [63] abstraction as an alternative. With PMOs, the PM is organized as a collection of objects that hold

[1] Intel has never confirmed which type of persistent memory technology is used with Optane PMem, but researchers have observed that it has characteristics consistent with PCM [30].

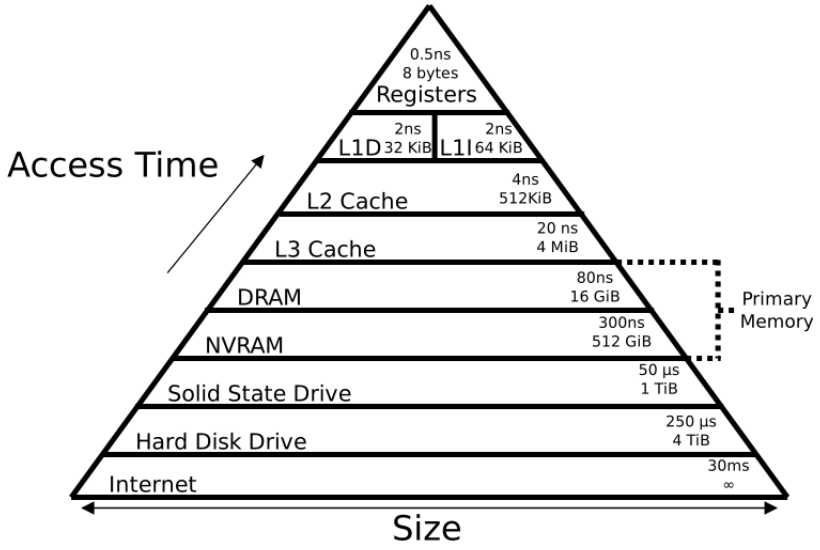



Figure 1.1: Memory hierarchy within a PM augmented system

data directly in pointer-rich data structures, without the backing of a file-system. Programmers can access a PMO and save data to it as if it were a file, while also maintaining pointers and buffers to other data. This approach is intuitive to the programmer, but it suffers from three major challenges.

- No commodity operating system supports PMOs. Before the Greenspan PMO (GPMO) system introduced by [27], only the design of PMOs had been discussed, and no implementation was provided.
- Before being introduced by [27], crash consistency, the property of allowing data to always be recovered to a consistent state even in the face of a crash, was not part of the inherent design of any PMO system. In works that do address it, crash-consistency within PM combines durability with transactional memory, which makes reasoning about crash-consistency much more difficult.
- While prior work addressed data in-use security for PMOs, before [27], no work had focused on at-rest attacks.

These challenges are all addressed in Chapter 2.

Although a PMO secured by the GPMO system described in Chapter 2 is protected against attacks at-rest and can detect at-rest integrity violations, attach and detach are now very slow. In addition, although data are always crash consistent under the design, is is not ”crash secure” - i.e., if a processes crashes with an attached PMO, the PMO will remain in a decrypted state. Chapter 3 addresses these dual problems with LOaPP and demonstrates its effectiveness. Finally, Chapter 4 extends the work of Chapter 3 by proposing the use of DRAM as a cache for PMOs, and then exploring mechanisms to speed up page faults by predicting which pages might be needed ahead of time and decrypting them into DRAM.

## Applications of PMOs

The primary advantage of PMOs is that they simplify the job of the programmer by making crash consistency, security, and integrity verification less cumbersome. Compared to files, crash consistency simply becomes part of the PMO abstraction, and the programmer does not have to know how it works or rely on compiler support for optimizations. For example, consider the microbenchmarks used to evaluate PMOs throughout this dissertation. These types of workloads are an example of the types of workloads that can benefit from PMOs. This class of applications originally used consistent checkpointing to a disk [56], which assumed the use of the normal I/O API and required serialization/deserialization. Converting this for use with PM is difficult and error prone. For example [17] relied on an LLVM pass to do this automatically, but this is cumbersome, requiring the use of `#pragma` calls and logging. This is difficult for the programmer since it does not follow the traditional I/O system calls such as fsync and msync. [2]

[2] A secondary problem with their approach is that their compiler pass only works with single threads.

Similarly, this dissertation uses Filebench to evaluate the impact that PMOs have on real-world applications that currently use I/O such as web servers, file servers, proxies, and mail servers. Another class of application that may take advantage of PMOs are in-memory databases such as Redis [61], which might handle sensitive data and also benefit from crash consistency. Both classes of applications can avoid the latency of serializing data to disk by using PMOs and invoking psync instead. Both classes of applications can also host sensitive data that can be the target of attacks, ence reducing the in-use attack surface by invoking attach/detach after each iteration of the main loop is another benefit of PMOs. Similarly, the data within these applications ought to be secured against corruption (either transient corruption such as a bit-flip, or intentional corruption such as an attacker attempting to deny access to data); hence integrity verification is important here too. While filesystems on PM can provide this, they cannot provide this and integrity verification without a cost (see Chapter 2).

## Contributions

To summarize, this dissertation makes the following **contributions**:

1. This dissertation identifies a new threat model for PMOs and presents an example security attack for exploiting PMOs at-rest, as well as a defense mechanism against attacks on PMOs at-rest through encryption and integrity verification.

2. This dissertation introduces *psync*, a novel primitive that relieves the programmer from the burden of relying on transactions or using low-level flushing and fencing to manage persistence.

3. For the first time, this dissertation proposes and implements the PMO abstraction in the Linux kernel on real Persistent Memory hardware (Intel Optane memory).

4. Furthermore, this dissertation extends the evaluated PMO system to provide *low-overhead*, *per-page* Encryption, Decryption and Integrity Verification to protect at-rest PMOs, aiming at encrypting, decrypting, and verifying only what is needed when it is needed. This dissertation then explores the design space for per-page encryption and systematically explores and analyzes them.

5. This dissertation evaluates the PMO system using a set of microbenchmarks and workloads from FileBench [66], and also evaluates a secure at-rest design against an insecure baseline.

6. This dissertation demonstrates that the performance of PMOs can be further improved through the use of DRAM as a cache, and pre-decrypting pages likely to be used at page faults.

## Organization

The rest of this dissertation is organized as follows. Chapter 2 provides the design, implementation, and analysis of persistent memory objects. Chapter 3 expands on earlier work by introducing LOaPP. Chapter 4 expands on LOaPP by introducing prediction and DRAM as cache for PMOs (PD-LOaPP); while Chapter 5 concludes.

# CHAPTER 2: THE CRASH CONSISTENCY AND SECURITY OF PERSISTENT MEMORY OBJECTS

This chapter explores and addresses three major problems with Persistent Memory (PM). First, this chapter introduces and designs a *Persistent Memory Object* (PMO) abstraction that allows data to be retained in memory across process lifetimes and system power cycles. Second, this chapter addresses the programmability of PM crash consistency management with a primitive *psync*, that decouples crash consistency and concurrency management; psync allows the programmer to specify *when* data are crash consistent but conceals *how* it happens. Third, this chapter addresses the security of PMOs while at rest against corruption and disclosure attacks. Finally, this chapter performs an evaluation and analysis of the PMO abstraction compared to state-of-the-art designs. [1]

## Introduction

DIMM-compatible Persistent Memory (PM), such as Intel Optane PMem, is expected to augment or replace DRAM as main memory due to its higher density and lower cost per byte. Due to its non-volatility, PM makes it possible for programs to host persistent data directly in memory.

Researchers have proposed at least two approaches for utilizing PM. In the *memory-mapped file* approach, persistent data are stored in files but memory-mapped to PM to allow direct access (DAX) through loads/stores [10, 15, 37, 71]. Such an approach avoids the use of system calls most of the time, but keeps data as an array of bytes and requires keeping two systems (filesystem and virtual memory) and their distinct metadata and semantics consistent for the same underlying

[1] The contents of this chapter are based on the SEED 2022 paper "Improving the Security and Programmability of Persistent Memory Objects" [27].

data. Alternatively, the *persistent memory object* (PMO) approach [4, 42, 73] organizes PM as a collection of persistent memory objects (PMOs) holding data directly in pointer-rich data structures without the backing of a filesystem.

The PMO approach is intuitive to the programmer but has several major challenges. First, it requires a new system abstraction that allows data to be long-lived across process runs and system boots; this abstraction does not exist in current Operating Systems (OS). Second, since data structures often contain pointers, they present an enticing target for security attacks. A pointer corrupted by an attacker in one run becomes persistent, enabling it to affect future runs of the same, or even different, applications [67]. Finally, there is a programmability challenge to achieve *crash consistency*. Crash consistency is the property that allows data to be recovered to a consistent state after a crash such as a power failure or system/application crash. Existing approaches to crash consistency add durability to transactional memory, relying on the tight coupling of concurrency management and crash consistency.

This chapter addresses the programmability challenge of achieving crash consistency. Prior work with PMOs, such as MERR [73] and Twizzler [4] do not address crash consistency, and they assume that the programmer or a library will manage it. Other works add durability to transactional memory, relying on the tight coupling of concurrency management and crash consistency, e.g. through Intel's Persistent Memory Development Kit (PMDK) [62], or through transactions in Mosiqs [42]. While such a coupling is feasible, it has several significant drawbacks. First, it forces crash consistency management to use the same granularity preferred by a transaction. To reduce the number of conflicts among threads (which trigger rollbacks and threaten forward progress), transactions prefer small code granularity. This may conflict with the preferred crash consistency granularity, which may be large because crashes are much rarer events than thread conflicts. Second, transaction-based consistency can only be achieved for transactional applications, restricting the use of PM to such applications [57].

Hence, this chapter proposes a different approach that does not rely on transactions. Instead, it introduces *psync*, a system call that provides a simple primitive to the programmer to achieve crash consistency. Psync decouples when and how data in a PMO reaches a crash consistent state; the programmer specifies points in the code where data are in a crash consistent state and the system then ensures that all stores prior to the psync are rendered durable prior to any store subsequent to the psync. Key to this, the system hides the mechanism of achieving crash consistency from the programmer; this chapter discusses how crash consistency is achieved in the underlying system, and the differences between psync and transactional approaches.

In addition, this chapter addresses some of the security and programmability challenges of a PMO, beginning with a discussion on the lifespan of a PMO, which can be conceptualized as a combination of two distinct periods: *In-use*, which is when a PMO is mapped (or attached) into the address space of a user process and accessible to it, and *At-rest*, which is when a PMO is not mapped to the address space of any process (i.e. detached, when a PMO is detached). While in-use, a PMO is exposed to *memory safety*-based security attacks. One defense to memory-safety attacks is to reduce exposure by attaching a PMO only when a process needs to access it and detaching soon after [73, 75, 77]. However, prior to this chapter, there have been no defenses proposed for at-rest PMOs. As with files, it is reasonable to suspect that a typical PMO spends of its time at-rest. While at-rest, it is vulnerable to disclosure or corruption. To protect against this, this chapter proposes encrypting an at-rest PMO and protecting its integrity with hashing. This way, even a disclosure reveals only its encrypted form, and corruption will be detected when the PMO's integrity is verified prior to use by a process.

This chapter presents the design and implementation of an actual PMO abstraction on a real Linux kernel with Intel Optane PMem; the design is guided by a desire to be **Secure**, **Crash Consistent**, and **Simple and Efficient**.

The major contributions of this chapter can be summarized as follows:

1. This chapter identifies a threat model for PMOs and presents an example security attack for exploiting PMOs at-rest.

2. This chapter presents a defense mechanism against attacks on PMOs at-rest through encryption and integrity verification.

3. This chapter introduces the *psync* primitive for crash consistency, which relieves the programmer from the burden of relying on transactions or using low-level flushing and fencing to manage persistency.

4. This chapter introduces the design and implementation of the PMO abstraction within the Linux Kernel, on a real system.

5. Finally, this chapter evaluates the PMO system using a set of microbenchmarks and workloads from Filebench [66]. The crash consistent design incurs only $27.8\%$ and $18.3\%$ overhead over a non crash consistent design for microbenchmarks and Filebench, respectively. When compared with a state-of-the-art crash consistent filesystem, NOVA-Fortis, the PMO system approach is $1.6\times$ and $3.2\times$ faster for microbenchmarks and Filebench, respectively. When evaluated against an insecure baseline, the secure at-rest design incurs a tolerable $3 - 46\%$ performance overhead, depending on the level of security enabled, for the executed workloads.

## Background

This section discusses the background and related work required to appreciate the rest of this chapter.

### *Approaches for Using PM*

PM may be used as *memory mapped files*, where persistent data are stored in files but memory-mapped to PM to allow direct access (DAX) through loads/stores. Examples include ext4-dax [15], Libnvmmio [10], splitFS [37], and NOVA-Fortis [71]. Alternatively, PM may be used as a repository of *persistent memory objects* (PMOs). Examples include MERR [73], Twizzler [4], and Mosiqs [42]. This work assumes the latter approach, where a PMO holds long-lasting data directly in pointer-rich data structures, without the backing of a filesystem.

Along the PMO approach, previous works include MERR [73], Twizzler [4], Mosiqs [42], and TERP [75]. Though Twizzler and Mosiqs provide a complete working design for an object-base abstraction of PM, they did not address security attacks, both *in-use* and *at-rest*. MERR includes a general defense strategy against in-use security attacks by attaching a PMO only when needed and keeping it detached otherwise; this lowers the temporal attack surface of a PMO by reducing the time a PMO is exposed in user-space. Xu et al. [77] extends the MERR approach to Intel MPK protection domains with the goal of restricting the access of PMOs only to the threads that require access to them. TERP extends MERR by providing a compiler pass to automatically insert attaches and detaches. These works did not address the security of PMO data at-rest, which is the focus of this chapter.

### *Crash Consistency*

Crash consistency is an important requirement for storing persistent data structures in PMOs. Otherwise, in the event of a system or application failure, partial or unordered writes might leave a PMO-resident data structure in an inconsistent state from which it cannot be recovered. For example, consider a persistent linked-list where node $A$ points to node $B$ and a new node $X$ is to be in-

serted between $A$ and $B$. The insertion operation can be performed in two steps: $X \rightarrow next = B$; $A \rightarrow next = X$. If a system failure happens when only the first update is persisted, $X$ is not reachable from $A$ after the system is restored. On the other hand, if a crash happens when only the second update is persisted, $B$ is not reachable on reboot. To protect against such partial updates, resulting in memory leaks and dangling pointers, PMO-resident data structures must be updated in an *atomic*, *crash consistent* way.

All prior work in PMOs [4, 42, 73, 75] do not provide crash consistency as an intrinsic feature of the abstraction design. Instead, they outsource the responsibility to the programmer, either as a library, or requiring the programmer to use the low-level primitives of flushing and fencing. This approach places the correctness burden on the programmer, who always needs to ensure that flushing and fencing are complete and done in the proper order. Any mistake is hard to debug and potentially leaves the data structure in an inconsistent state on a system failure.

### *Persistent Pointers*

A PMO can be designed with either absolute or relative persistent pointers. An absolute pointer contains a virtual address, e.g. in Mnemsoyne [68], and is fast to dereference because it relies on traditional address translation mechanisms. However, it makes PMO mapping rigid and PMO Space Layout Randomization (PSLR) [73] costly; any time the PMO is mapped to a different virtual address region, pointers in the PMO must be rewritten accordingly. Finally, if multiple processes are allowed to simultaneously share a PMO, absolute pointers require that the PMO be mapped to the same virtual address range in all processes.

A relative pointer is a combination of a PMO ID and offset in format of `object:offset`. It can use a regular 64-bit format or use a fat pointer format where a pointer is represented by multiple fields. To dereference a pointer, a translation table is used to translate the system-wide unique

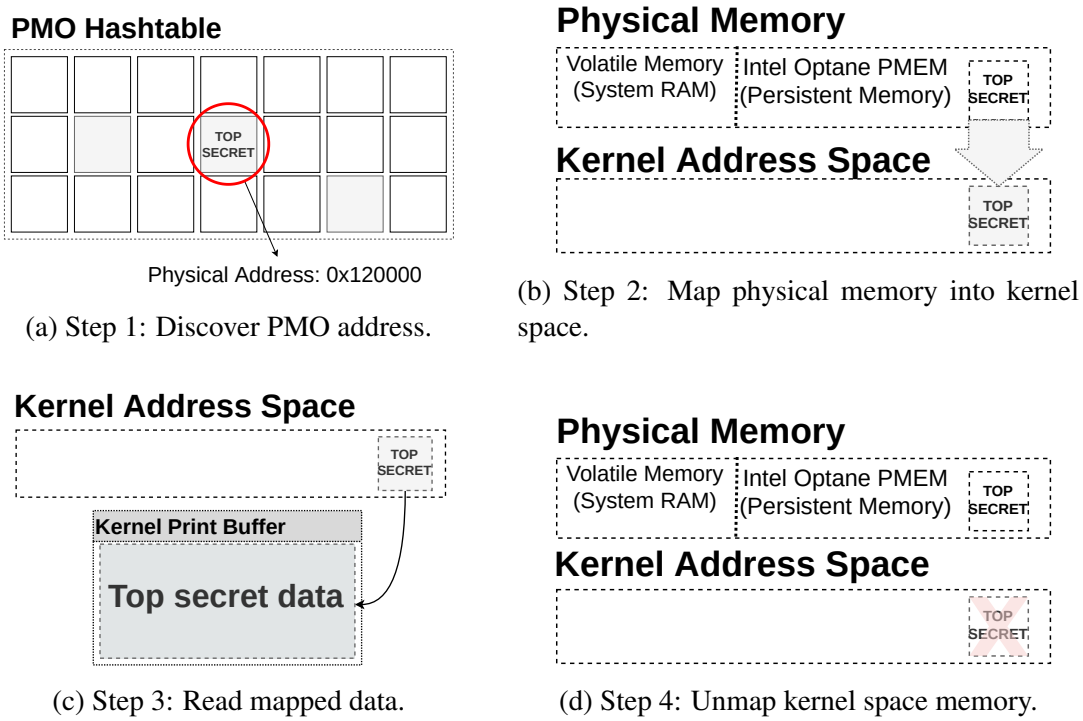
PMO ID to its base virtual address [69], and then the offset is applied. Unlike absolute pointers, relocating such PMOs is straightforward to perform, but dereferencing is expensive. MERR/TERP and PMDK uses the relative pointer approach, as does Mosiqs, which uses PMDK internally. Twizzler also uses relative pointers, and provides a lightweight design that repurposes the virtual memory capabilities of the x86 Memory Management Unit (MMU) for persistent pointers.

While relative pointers facilitate relocation of PMOs on every attach and thus support PSLR, they are inherently expensive for dereferencing as they require an extra translation step to be mapped to a virtual address. Therefore, there is a clear trade off between performance and flexibility that is difficult to reconcile, and no option is clearly superior.

## Threat Model

Consider a threat model where a PMO is not attached to any user process and so is not accessible in the user space context (i.e., the PMO is at-rest). This dissertation's threat model assumes that PMO-resident data-structures may contain buffers and pointers, and does not trust system software except for a small subset; specifically, the Linux Kernel Crypto API [49], critical memory functions, such as memcpy and memset, and the PMO kernel subsystem. This assumption is reasonable, as the code-size of these components are small enough to be formally verified. For example, the the Linux kernel Crypto API for version 5.14.18 contains about 82,500 source lines of code, the PMO kernel subsystem contains about 1,200 source lines of code, and the kernel memory functions contain about 100 lines of architecture-specific inline assembly, compared to the rest of the kernel, which contains about 2.2 million lines. That means that the PMO subsystem, the critical memory functions (which are architecture specific, and written in assembly), and the Linux Kernel Crypto API are only $0.4\%$ of the entire kernel.

The goal of the attacker is to disclose private data of a user-process held in an at-rest PMO or to overwrite it with malicious data. Furthermore, the model assumes that the attacker knows the location and layout of a PMO in persistent memory; hence, this threat model is different from the threat models of both [67, 73], since their threat model requires that the PMO be attached by a user process before it can be exploited.




(a) Step 1: Discover PMO address.

(b) Step 2: Map physical memory into kernel space.

(c) Step 3: Read mapped data.

(d) Step 4: Unmap kernel space memory.

Figure 2.1: Steps of PMO example attack.

Figure 2.1 illustrates an example PMO data-disclosure attack, assuming that the attacker has already exploited an existing vulnerability in the kernel code to alter its control flow. In Step 1, the attacker discovers the physical address of the desired PMO by stepping through the PMO metadata hashtable (described in the layout section). In Step 2, the attacker maps the PMO into the kernel virtual address space (With Linux, the data are mapped into the vmalloc/ioremap kernel

space [15]). In Step 3, the attacker copies the contents of the PMO into the kernel print buffer; disclosing the secret information within. Finally, in Step 4, the attacker clears the print buffer and unmaps the PMO from the kernel address space, leaving no trace of the attack. Alternatively, in a PMO data-injection attack, an attacker would instead write and then persist invalid or malicious data into the PMO during Step 3. If a user process attaches the PMO in the future, and relies on its data, this could potentially alter its control flow.

## Persistent Memory Object Design

To achieve atomic crash consistent updates, there are two broad approaches: *logging* and *shadowing*. Logging (undo/redo) requires the creation and management of an undo/redo log, which is expensive to achieve within the kernel due to the need to intercept every first store (undo log) or every store (redo log). The kernel can only intercept a store if it has previously marked the page as read-only; in that case, the store incurs an interrupt (page fault) that is caught by the kernel.

Therefore, this dissertation proposes the use of shadowing, where each *writable* PMO is backed by a shadow copy with the same allocation size as the primary copy (i.e., the PMO itself). All updates are performed on the shadow until a crash consistent point is reached. However, before performing updates, a shadow must be initialized by copying over pages from the primary PMO. Instead of creating a shadow for the entire PMO, the system creates a shadow page only for a page that is actually written. Thus, regardless of the PMO size, the shadow only consists of the subset of pages that are actually modified. To achieve this, a page is copied over only on a page-fault caused by a write over a read-only page (i.e. on-demand page copying). Subsequent invocations of psync persist updates in the shadow before synchronizing it with the primary. Here, synchronization refers to the process of copying updated shadow pages back to the primary PMO and persisting them. Note that for a PMO that is attached with read-only access, no shadow copy is created at all.

*PMO System Layout*

As shown in Figure 2.2, the PMO design divides the persistent memory fabric into three regions: the *Header Region*, which contains information important to the entire PMO system, the *PMO Metadata Region*, which contains a hashtable designed to make PMO operations fast, and the *PMO Region*, which contains the PMOs themselves.

The Header Region is situated at the start of the persistent memory, making it easier for the kernel to access the header information. To keep space overhead low, the header is only 4KiB (1 page) large. It contains a magic key indicating that the device has been formatted as a PMO system, the name of the PMO system, the starting and ending addresses of the PMO system, the starting address of the PMO Metadata Region, the starting address of the next available space for a PMO, and padding for future expansion (such as versioning information). The header must be mapped as uncacheable[2] in kernel memory to ensure all header updates are durable.

The Metadata Region stores the metadata information of all PMOs. It consists of a header containing the allocated node count, which represents the number of PMOs in the system. The count is used to keep track of the number of created PMOs, which is capped by the size of the hashtable. The count is followed by the *Metadata Entry Hashtable*. Each entry contains the minimum necessary information to ensure the correct operation of PMOs, consisting of the current state of the PMO (states and their transitions are described in a later subsection), the name and size of the PMO, a pointer to a linked list in volatile memory tracking the PIDs of processes for which the PMO is currently mapped to (when mapped as read), the current boot ID[3], and the address of the PMO and its shadow copy (if any). The PID and Boot ID are used in combination to ensure that

[2]In Linux, using the x86 architecture, this is done via `ioremap_uc()`. This tells ioremap to map the specified address as being ”strongly uncacheable”, which bypasses the CPU cache.

[3]In Linux, this is found at `/proc/sys/kernel/random/boot_id`.

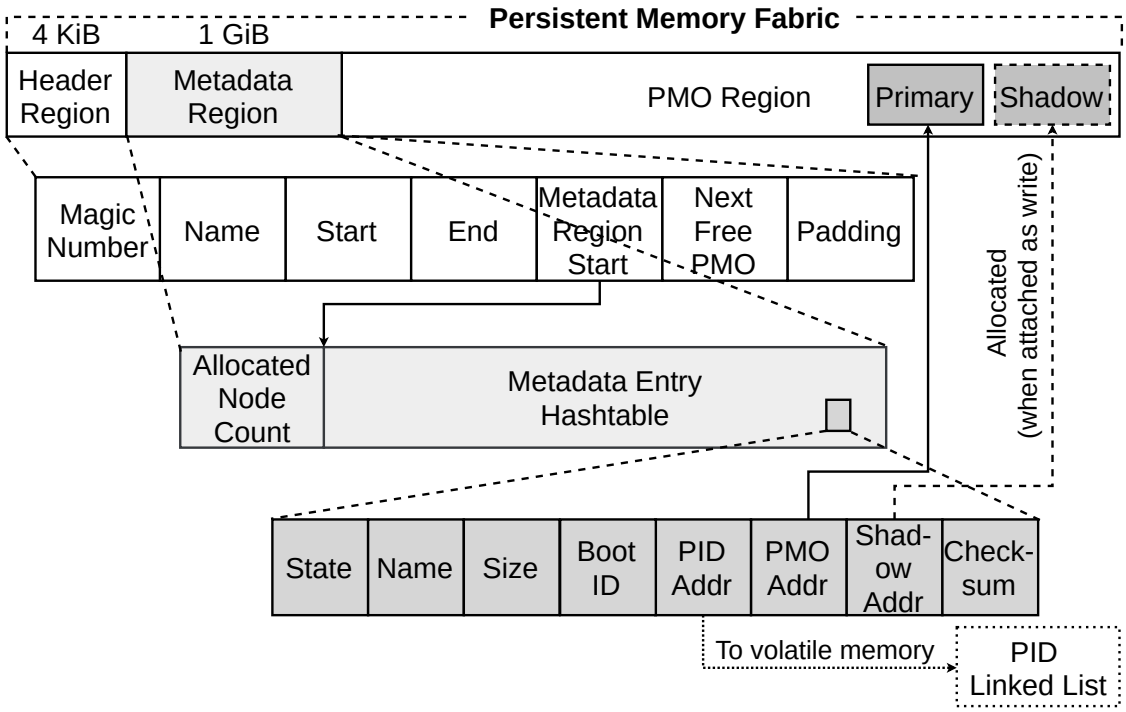



Figure 2.2: PMO System Layout. See also Figure 3.4 in Chapter 3.

only one process has attached a PMO with write permissions at a time. The checksum field is used for PMO integrity verification as described later. To avoid false sharing, each entry should be a multiple of the size of the CPU cache line. The rest of the PMO system consists of the PMO Data Region, where the PMOs themselves reside.

### *The PMO API*

The PMO API consists of multiple primitives: attach, detach, psync, pcreate, and pdestroy.

Table 2.1: Summary of PMO programming interface

| **Primitive** | **Description** |
|---|---|
| attach(name,perm,key) | Render accessible the PMO `name`, given a valid `key` with permissions `perm`. |
| detach(addr) | Render inaccessible the PMO `addr` points to. |
| psync(addr) | Force modifications to the PMO associated with `addr` to be durable. |
| pcreate(name,size,key) | Create a PMO `name` of `size` and `key`. |
| pdestroy(name,key) | Given a valid `key`, delete PMO `name`, reclaiming the space for a new PMO. |

*The pcreate/pdestroy primitives*

The **pcreate** primitive creates a PMO of a specified name, size, and key. Once created, a PMO resides within the PM unless it is destroyed by **pdestroy**. Upon invocation, the kernel searches for unoccupied space within the PMO Region by using the "Next Free PMO" field of the header region (Figure 2.2). If the requested size is larger than the remaining available space for the PMO system, then the call sets (`errno`) to `ENOSPC` and returns a null pointer. Otherwise, a persistent region of a requested size is reserved for the PMO, the "Next Free PMO" field is updated, and an entry is created in the PMO metadata hashtable.

*The attach primitive*

Successful invocation of the **attach** primitive maps a PMO of a specified name into the virtual address (virtual address.space of the calling process, rendering it accessible to the process. The system only allows one **process** to attach a PMO with intent to write, but allow multiple processes to attach a PMO with intent to read. Read and write permissions are *mutually exclusive* of each other, e.g., a PMO cannot be attached as a read by one process, and as a write by another. This avoids data consistency problems that arise from multiple writers.

**PMOs: A(R), B(RW), C(RW)**

| Process P1 | Process P2 | Process P3 | Outcome |
|---|---|---|---|
| attach(A, rw,) | | | invalid (permissions) |
| | attach(B,r,) | | valid |
| | | attach(B,r,) | valid |
| | | attach(C,rw,) | valid |
| | attach(C,rw,) | | invalid (>1 writer) |
| attach(C,r,) | | | invalid (existing writer) |

Figure 2.3: PMO inter-process sharing semantics.

Figure 2.3 illustrates how attach works in the context of multiple processes. One PMO has read only access permission (PMO A), and two others have read and write permission (PMOs B and C). If P1 attempts to attach A with a read/write access request (top line), the call returns with an error due to insufficient permissions, as A is restricted to read only access. Later, if P2 attempts to attach B with read only access, the attach succeeds. P3's attempts to attach B with read only access is also granted as multiple readers are permitted. P3's attempts to attach PMO C with read/write access is valid, but P2's attempts to attach it returns an error. Finally, an attach request for PMO C by P1 also returns an error because there is already an existing process that has attached the writer.

To ensure mutually exclusive read and write access to a PMO, the boot ID is used in combination with a linked-list of PIDs. A PID entry is added to the linked-list on successful attach and removed on detach. On an attach call, a boot ID field (in the corresponding hashtable entry) that is different from the current PMO system's boot ID indicates that a system crash or forcible reboot has occurred. On the other hand, if the boot ID matches but none of processes listed in the PID linked list are alive, it indicates that they have abnormally terminated. In both cases, the kernel starts the recovery procedure for the requested PMO. Otherwise, for a read-only attach or a write attach when the PID linked-list is empty, the kernel adds an entry to the linked list, sets the boot ID field to the system's boot ID and returns success. If the linked-list is not empty for a write attach,

the kernel rejects the attach, sets `errno` to `EAGAIN`, and returns a null pointer.

*The detach primitive*

The **detach** primitive renders a PMO inaccessible to the calling process. If a process attempts to detach a PMO that is already detached or has never been attached, this results in undefined behavior, as the virtual address has an invalid mapping to the physical address of the PMO. Successfully detaching a PMO attached as write sets the boot ID to sentinel value and removes the PID from the linked-list, while successfully detaching a PMO attached as read only removes the PID linked-list entry associated with the calling process, unless the calling process is the only process that has attached the PMO; in that case, the entire linked-list is destroyed. Detach does not persist modifications after the last psync, hence the programmer is expected to call psync prior to detach to persist all modifications.

*The psync primitive*

Since all modifications to the PMO are performed on its shadow copy, the **psync** primitive forces all modifications made on the shadow copy to reach the persistency domain by initiating a sequence of flushes followed by a memory barrier, and then synchronizes it with the primary copy. Psync is designed to have similar semantics to the POSIX **msync** and **fsync** [28], but with only one argument: a pointer to the PMO. As described earlier, psync has *atomic semantics* for stores to the PMO (i.e. primary copy), but *non-atomic semantics* for all other stores.

Figure 2.4 illustrates the atomic semantics with an example, showing two psync calls for PMO A, with stores to A (st1, st2, st4, and st5) or to PMO B (st3 and st6). If the first call completes, then the PMO A (i.e., its primary copy) in memory reflects the durable state of st1. Prior to the completion

of the second psync, PMO A is unchanged. It is only afterwards that PMO A reflects the durable state of st2 and st4 but not st5. Therefore, psync is atomic for PMO A as the stores between the two psync calls are either entirely reflected (in the case of a successful completion) in PMO A or not at all (in the case of a failure between two psync calls). PMO B (i.e., its primary copy) remains unchanged as there is no psync involving PMO B. Note however that st3 and st6 may or may not be reflected in the shadow copy of PMO B, depending on whether the corresponding cache blocks have been evicted; this is because stores to the shadow copy are non-atomic.

These data-centric semantics distinguish psync from a transaction, which is thread/code-centric. Not invoking psync prevents prior modifications from becoming persistent, which is in contrast to fsync's semantics, where changes to the copy in memory may still be reflected on disk, even if fsync is not invoked. As a result, psync gives the programmer explicit control over crash consistency points for PMO data in their code. Upon crash-recovery, the PMO (i.e., primary copy) reflects the updates persisted by the most recently completed psync. From the point of view of the system, psync provides a PMO-specific persistency barrier, and is idempotent.

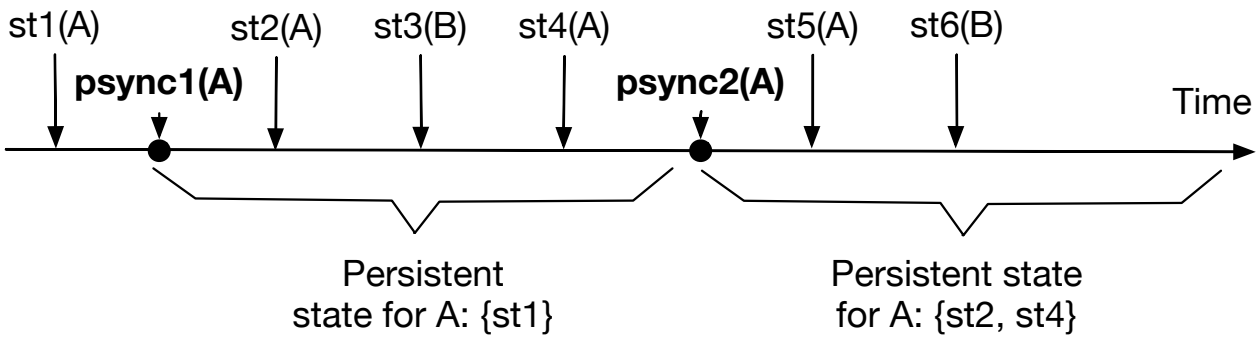



Figure 2.4: Transactional semantics of psync

In a multi-threaded application, a situation can arise where a psync that was invoked by one thread on a given PMO is in progress, while another thread (from the same process) wants to write to the same PMO. There are several options to address this challenge. The naive approach is to block

all reads/writes on a PMO for which psync is in progress. Though a workable solution, this can significantly slow an application's progress. A more aggressive approach is to mark shadow pages that have been synchronized with the primary PMO and allow the writer thread to update only those pages while blocking a writer thread requesting unmarked pages. This approach, similar to one adopted by NOVA-Fortis [71], can reduce the blocking interval as it blocks a thread only when it requests access to unmarked pages. For the current design, and for performance reasons, the programmer is expected to avoid this situation through synchronization.

### *Design for Fast Access*

The latency seen by an application storing its persistent data in a PMO is affected by two main factors: primitive latencies and pointer dereferencing latency. *Primitive latencies* refer to the latencies for performing PMO primitives, such as creating PMOs, mapping and unmapping them in a process address space, and rendering them durable in a crash consistent manner. These latencies in turn depend upon the amount of metadata that must be managed by the kernel while controlling accesses to a PMO. *Pointer dereferencing latency* is the time-overhead involved in translation between virtual PMO pointers to their physical counterparts while accessing the PMO-resident data structures. A low-latency design is provided by the following design choices:

#### *PMO Layout*

Most filesystems use pointer chasing to locate the next block of a file or track free blocks, such as with filesystem inodes. Though this approach supports the dynamic growth of a file, it is not conducive to fast access. Instead, an approach where a PMO is a contiguous region of memory with a static size set at the PMO's creation is preferred. Any PMO-resident data can be accessed by adding a given offset to the base-address of the PMO; this approach is faster as it does not need

to chase pointers. If the size of the data structure grows beyond what was allocated initially for the PMO, a resize operation can be performed by creating a new PMO with a larger size, copying its content, and deleting the original PMO.

*Low-latency Attach/Detach*

In a naive approach, on invocation of an *attach* system call, the kernel could map the entire PMO into the process address space and unmap the entire PMO at *detach*. This solution is expensive for a large PMO with multiple page table entries (PTEs), as PTEs need to be initialized by the kernel, invoking expensive TLB shootdowns and subsequent TLB misses. MERR [73] proposed embedding the page table subtree into the PMO itself, so that when a PMO is attached only one PTE needs to be initialized. As a result, this solution means that regardless of PMO size, only a single TLB shootdown is needed when a PMO is mapped into virtual memory. However, MERR's solution needs a custom hardware permission-matrix to provide access-control to PMO. Since the PMO system must work with commodity hardware, a different solution, *demand paging* [24] is needed.

When a PMO is attached, the kernel sets a flag to indicate that future page faults for pages within a PMO should map the faulting page into a virtual address space. When a PMO is detached, the kernel renders the specified address associated with a PMO inaccessible, by setting the metadata entry to the detached state, and then disabling the read/write permissions of all *faulted* pages, ensuring that all page faults on the address range generate a segmentation fault. This solution is not as efficient as MERR's solution, which requires specialized hardware. However, in most cases, this solution is faster than simply mapping the entire PMO at attach time and also works on existing systems, because only accessed pages have been faulted in, instead of all of them.

*Low-Latency Pointer Dereferencing*

A key challenge of PMO-resident (i.e. persistent) pointers is that the virtual address that a pointer refers to must be associated with the physical address of a PMO beyond the process lifetime [2]. This is required to ensure that pointers within and across PMOs are always valid. As described in the background, one solution to this challenge is to use *relative pointers* in `object:offset` format [6], and use a per-PMO Persistent Object Table (POT) for efficient pointer translation from persistent to virtual form. However, this approach puts the pointer-to-virtual address translation on the critical path to PMO access, incurring substantial latency, and increases the amount of PMO metadata. For TPC-C, persistent pointer dereferencing was reported to cause a 15% execution time overhead [69]. While hardware supported pointer translation [69] could significantly reduce its latency, it is not clear if such an expensive hardware solution is necessary. The high latency software translation is in conflict with the design goal of fast access to PMOs, while hardware-based translation conflicts with the goal of PMO systems being available on existing systems.

As an alternative to relative pointers, *static pointers* can be used. Static pointers are already in virtual address format, so they can be dereferenced without additional overhead and without any need for hardware support, just like non-persistent pointers. However, the drawbacks are that all pointers in the PMO need to be updated when the PMO mapping address changes, and that two different objects must not map to the same virtual address. Therefore, the virtual address range is assigned to PMOs at creation time such that no two addresses overlap. To achieve this, several techniques are used. First, to avoid an overlap between PMOs and non-persistent data, the effective virtual user space address space is split into two halves based on the most significant bit of an address: persistent and volatile, with the persistent-half reserved for PMOs and starting at (for example) the virtual address $x$. The kernel maps a PMO into the persistent-half of the virtual address space by assigning to it the address range from $x + y$ to $x + y + s$ where $y$ is the offset of

the PMO in PM from its start, and $s$ is its size. To prevent two PMOs from mapping to the same virtual address range, the system assigns PMOs globally unique virtual addresses.

Such an approach runs the risk of running out of virtual address space if there are too many large PMOs. For example, in a 48-bit address space, the persistent half can only hold a maximum of 128 TiB (i.e., 64 million 2MiB-sized PMOs, but only 128 thousand 1GiB-sized PMOs). One possible mitigation strategy is to use static pointers for small to medium PMOs (KiBs to MiBs), and use relative pointers for larger PMOs (GiBs and above), although this is not implemented in this dissertation.

Since the virtual address of a PMO is determined by its physical location in the PM, this approach makes it costlier to perform the PMO address randomization used in MERR [73]; moving a PMO to a different virtual address requires updating all pointers in the PMO. Nonetheless, the common case of quickly dereferencing persistent pointers *without* the need of software or hardware translation, or additional per-PMO metadata, means that the approach is attractive overall.

### *Design for Crash-Consistency*

#### *Psync*

The psync system call should persist updates in a PMO without requiring explicit logging by the programmer. Also, updates should be persisted in an atomic fashion; all or none should become durable. To achieve these goals, the PMO system manages two copies of the PMO data: the primary copy and the shadow copy. Writes to the PMO are performed in the shadow copy until psync is invoked, at which point the writes are copied over to the primary copy in a durable atomic manner by the system call.

As a naive approach, at the time of PMO creation, the system could allocate twice the requested PMO size in PM and split it in two halves. The first half of the allocation can be used for the primary copy and the second half for the shadow copy. Since the hashtable entry for a PMO keeps track of its start address and size, calculating the starting and ending address of each copy of the allocation is simple. However, this approach is wasteful, especially when a PMO is attached only with read permissions and so a shadow copy exists, but is never used. Instead, a different approach is used where at the time of creation, a memory region of requested size is allocated and serves as the primary copy. When a PMO is attached by an application with write permission, the design allocates the size of the PMO again and designates it as the shadow copy. This approach can potentially result in primary and shadow copies non-contiguous to each other. Therefore, the system tracks the shadow copy through the "PMO Shadow Addr" entry pointing to the location of the shadow entry (see Figure 2.2). A null entry indicates that the PMO is either detached or attached as read only.

It is worth noting that this design prohibits a potential alternative mechanism for psync, whereby rather than copying pages over, the shadow page is persisted and the pointers to the primary and shadow are swapped. This solution also suffers from the fact that it would cost an additional pointer dereference.

*PMO state transitions*

To achieve crash consistent updates, a PMO is always in one of the five states shown in Figure 2.5. The state is kept in an uncacheable portion of the metadata hashtable. A state transition is performed using an atomic instruction, and since the state pages are not cacheable, the changes made by the atomic instruction are also durable.

A PMO is initially in the Ⓓ (detached) state upon creation. If it is attached by a process with read-

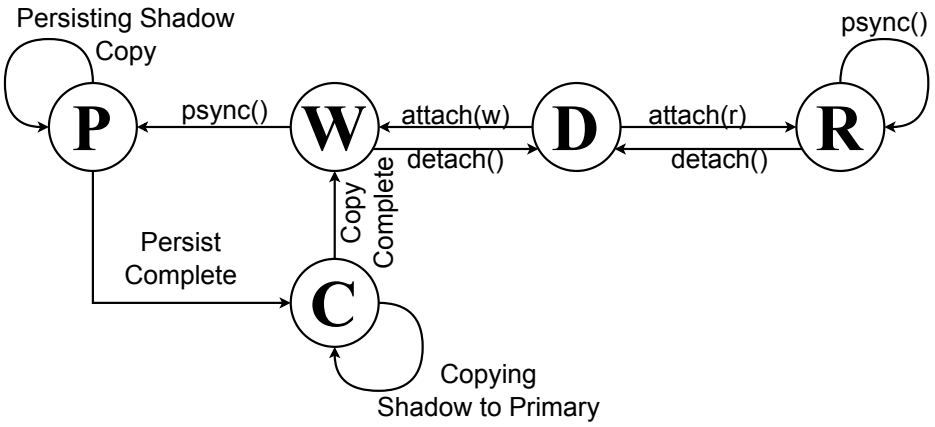



Figure 2.5: PMO state transitions without encryption and integrity verification

only permission, the PMO state transitions Ⓓ → Ⓡ (Read), where updates to the PMO are not allowed, and psync is ignored. When attached with write permissions, the PMO state transitions Ⓓ → Ⓦ (Write) where updates are permitted. If a programmer invokes detach on a PMO in the Ⓡ or Ⓦ state, it transitions (Ⓡ, Ⓦ) → Ⓓ. If a programmer invokes psync on a PMO in the Ⓦ state, it transitions Ⓦ → Ⓟ (Psync), to indicate the start of psync. The kernel then performs a page table walk to identify all dirty shadow pages [21] associated with the PMO, and the cache lines belonging to the dirty pages are flushed. After this point, the shadow copy is durable and consistent, and the PMO state transitions Ⓟ → Ⓒ (Copy). The kernel copies all modified pages from the shadow to primary, flushes the cache lines, and emits a memory barrier. The PMO returns Ⓒ → Ⓦ after the memory barrier completes.

*Recovery*

If psync is interrupted by a crash or power failure, the kernel must ensure that the PMO is recoverable. It is helpful to start with an invariant: at least one of either the primary or shadow copy contains a consistent version of the data (the "valid copy"). The recovery process depends on the

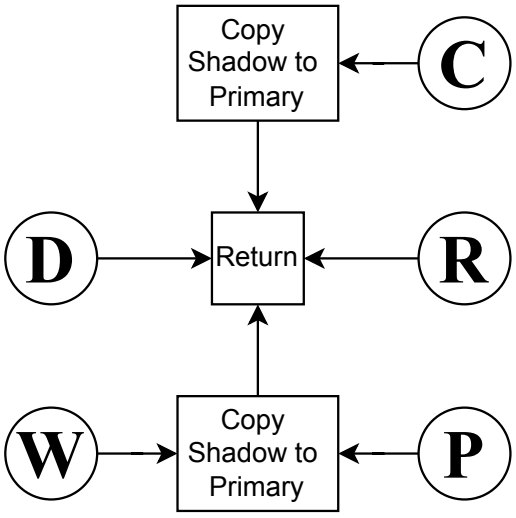



Figure 2.6: State transitions for PMO recovery

state of each PMO to determine which copy to rely on as consistent, illustrated in Figure 2.6. On post-crash attach, the kernel checks the state of the PMO. If Ⓓ or Ⓡ, then the primary and shadow PMOs are both valid, so there is nothing to do; if Ⓦ, psync has not started, and the primary copy is the consistent one, so it is copied over to the shadow, this in effect removes transient updates in the shadow copy since the last psync. If Ⓟ, psync has started but there is no guarantee that the shadow copy is consistent, so this case is treated the same as Ⓦ; finally if Ⓒ, the shadow copy is known to be consistent and reflects all the updates until the current psync, but the primary copy might not (it could be partially copied over). In this case, the shadow copy is copied to the primary.

### Security Protection of at-rest PMOs

This section presents a design that protects against the threat model and example attack described previously. The design provides a defense of at-rest PMOs against corruption and disclosure through integrity verification and encryption, respectively.

*Protection From Corruption*

To protect a PMO at-rest from corruption, the PMO design relies on a checksum computed over the PMO at detach time, and durably stored in the checksum field of the corresponding hashtable entry (Figure 2.2). A future attach on the same PMO triggers checksum recomputation and compares it against the stored one; a match indicates that integrity is verified, i.e. the PMO has not been modified at-rest, and the attach returns a pointer. Otherwise, it returns an error code to the calling process.

The computation of a checksum on every detach/attach increases the latency of the attach/detach system calls, especially for larger PMOs. This subsection identifies two optimizations to address this problem. First, for detach, the checksum can be computed out of the critical path, in the background. This is achieved by returning the detach system call immediately after the data within the PMO have been rendered inaccessible, while launching a background kernel thread to compute the checksum. The latency of the checksum computation is thus hidden, except if an attach request is made for the same PMO, which is then blocked until the computation is completed.

However, integrity verification is inherently a part of the critical path of an attach, hence hiding it is challenging. One possible optimization is to allow computation to continue speculatively before integrity verification is completed. If verification fails, computation is rolled back and speculative state is discarded. Most modern processor architectures already have a mechanism for speculative execution to support out-of-order execution. This would prevent integrity verification from blocking forward progress on a PMO, while some of them include an additional transactional memory support to execute a transaction speculatively. Unfortunately, they are only capable of speculation up to several tens to hundreds of instructions, allowing only tens to hundreds of nanoseconds of latency hiding capability, while requiring hardware/kernel coordination. Furthermore, as demonstrated in Spectre/Meltdown-style attacks, speculative execution may result in data leakage even as

they are eventually rolled back. An alternative solution is to split the PMO into chunks, maintain a separate hash for each chunk, and perform integrity verification not at attach time but on first load/store to that chunk. This on-demand and chunk-level integrity verification can potentially lower the latency incurred on the critical path. This is discussed in further detail in Chapter 3.

A second aspect to consider is the checksum algorithm selection and checksum hash length, which may range from slower/more secure to faster/less secure. For example, MD5 is faster and less secure than SHA256.

### *Protection From Disclosure*

Recall from the threat model that the Kernel Crypto API, certain kernel memory routines such as memcpy and memset, and the PMO subsystem are trusted. To protect a PMO at-rest from disclosure, the PMO subsystem invokes the Kernel Crypto API to decrypt the PMO only when it is in use, and immediately decrypts it when it becomes at-rest, i.e. detached. One challenge in providing encryption to protect PMOs from at-rest disclosure is to retain the crash consistency offered by the base design. The difficulty in retaining crash consistency arises from the fact that encryption/decryption is not atomic; therefore, like normal writes to a PMO, it may also be interrupted by crashes and power failures. A crash consistent PMO that supports encryption should be either entirely encrypted or entirely decrypted, hence crash consistency orchestration is needed. Furthermore, encryption/decryption modifies the PMO, so even a read-only PMO incurs modification.

To that end, crash consistency with encryption continues to utilize a shadowing approach, including when a PMO is attached for read-only access. Here, the the shadow copy is repurposed to manage the crash consistency of encryption/decryption. With shadowing, a PMO is never encrypted or decrypted in place. In this way, it is guaranteed that the system always has either a primary or shadow copy free of partial encryption/decryption in the case of a system or application crash.

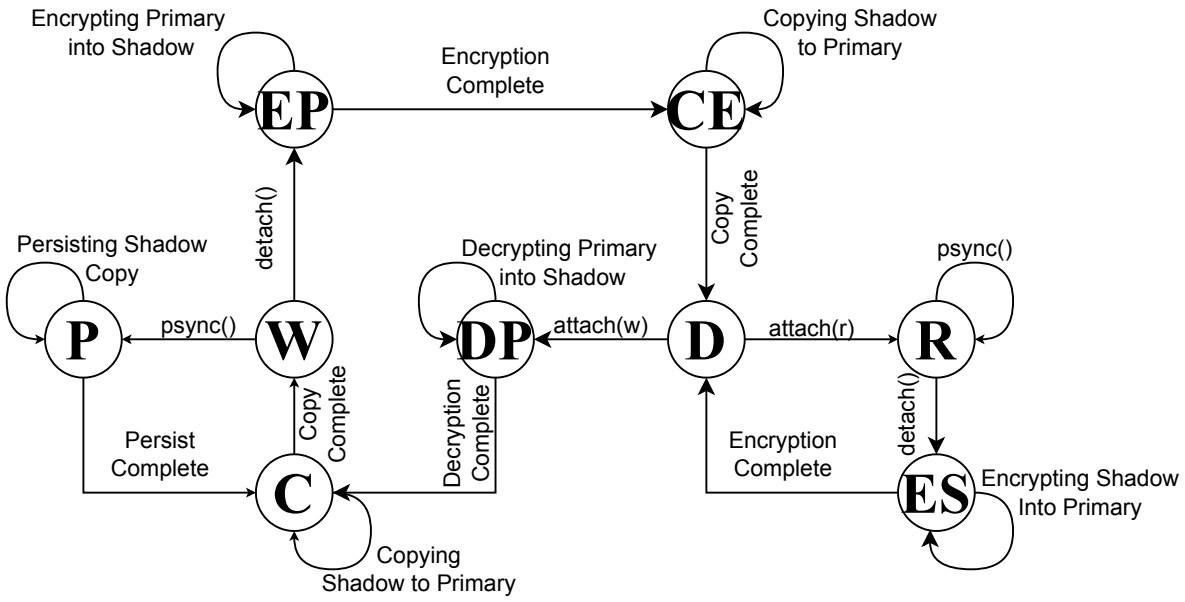



Figure 2.7: PMO state transitions with encryption

Figure 2.7 shows the state-transition diagram for the PMO design with encryption. A PMO is initially in the Ⓓ state. On an attach call, irrespective of the permissions, it transitions Ⓓ → (DP) (Decrypt Primary) where the kernel decrypts the primary copy into the shadow. This ensures that if the system crashes while decryption is in progress, the primary copy is still consistent and can be used for recovery. After completing decryption and persisting the shadow, the PMO transitions (DP) → Ⓒ, where the shadow is copied back and persisted to primary. At the completion of copying, both the shadow and primary copies are decrypted and durable, and state changes to either Ⓡ or Ⓦ, depending on attach permissions.

In Ⓡ, psync is ignored, while the invocation of detach transitions Ⓡ → (ES) (Encrypt Shadow), and the kernel encrypts the shadow copy into the primary and persists it. Once encryption has completed, the shadow is zeroed, persisted, and set free for future use. Finally, the PMO transitions Ⓡ → Ⓓ.

In (W), the invocation of psync triggers state transitions in the same way as in the basic crash consistent design. In case of invoking detach in (W), the PMO transitions (W) $\rightarrow$ (EP) (Encrypt Primary) where the kernel encrypts the primary copy to the shadow and persists it. Note that detach does not automatically persist modifications; the expectation is that the programmer will call psync prior to detach, making both the primary and the shadow copy updated and persisted before detach is invoked. Therefore, encrypting the primary into the shadow still preserves the updates in the primary copy. Upon the completion of encryption, the PMO transitions (EP) $\rightarrow$ (CE) (Copy Encrypted), where the kernel copies the encrypted shadow to the primary and persists it. Encryption is not performed in place due to the risk of partial writes in case of a system crash. Finally, when copying is completed, the PMO transitions (CE) $\rightarrow$ (D). Note that in the (D) state, both the primary and shadow copy are encrypted, so it is safe to destroy the shadow copy.

It is important to note that encryption keys are *not* stored alongside the PMO, in persistent memory, or in the kernel when the PMO is at-rest. Rather, the kernel receives a copy of the encryption key from the programmer at attach time, which means that an attacker must know the encryption key to modify an at-rest PMO.

The above scheme exposes the full decryption latency in the attach system call. A possible optimization is to decrypt the primary into the shadow and then immediately return with success, avoiding copying of shadow into primary and cutting its latency from the attach critical path. This should reduce attach latency substantially. However, since this design would have the primary encrypted but shadow decrypted, psync would require that the shadow to be encrypted first before it is copied to the primary. This optimization is therefore ineffective for a situation where psync is frequent, but effective when psync is infrequent. This optimization is applied for PMOs that are read-only, since psync does not occur.

## Implementation

This section discusses the implementation of the PMO system on the Linux kernel.

### *Persistent Memory Provisioning*

Intel Optane PMem supports namespaces which expose the memory as a logical device [55] with different modes. The namespace is provisioned in `devdax` mode, providing direct access (DAX) to the underlying PM [31]; this mode emits a *character device* (as opposed to a block device). Analogous to block devices being formatted with `mkfs`, a similar utility for formatting PMO systems, `mkpmo` is provided. This utility zeroes a given namespace and writes the PMO header and metadata structures.

### *Kernel Modifications*

Multiple modifications to the kernel are made to support PMOs. Three system calls, attach, detach, and psync are added. In addition, the Linux per-process memory descriptor (`mm_struct`) and the virtual address page fault handler are both modified to use demand paging for PMOs, adopting the `VM_SOFTDIRTY` flag [21] to track modified pages, so that psync only copies dirty pages to the primary copy.

To represent PMOs mapped in the address space of a process, the existing `vm_area_struct` in the Linux kernel is extended to include new fields needed only when mapping PMOs in the address space of a process. This extension is a separate struct that is a pointer from the VMA: called a `vpm_area_struct`. To allow for quick access, a new red-black tree is introduced to `mm_struct`, with `pmo_rb` as its root.

Most modern x86-64 CPUs can access $2^{48}$ virtual addresses[4], divided into kernel and user space. The user-space virtual address range is split in half and the upper-half (i.e. the second most-significant-bit is 1) is reserved for PMOs, resulting in $2^{46}$ addresses for PMOs and $2^{46}$ addresses for normal user-space processes. The page fault handler is modified so that a page fault to a PMO invokes the PMO page fault handler code to check the state of the PMO and handle demand paging.

### *Attach/Detach and Psync*

The metadata about the PMO is organized in a new kernel wide radix tree (pmo_radix_tree), which provides fast lookup when an attach call is made to determine if the PMO exists. If a PMO is newly attached, a VPMA is created, initialized, and data structures (hashtable and red black tree) updated. On detach, the kernel searches the VMA associated with the specified address and process and traverses the linked list to change each page permission to `PROT_NONE`. For the integrity verification checksum and encryption, SHA256 and XTS-AES-256 are used, respectively

The psync call *walks* the page table for shadow pages associated with the attached PMO to identify dirty pages through the soft-dirty bit. Modified pages' indices are added to a linked list, and all associated cache lines are flushed to persist the page using memcpy_flushcache, and their dirty bits are cleared. A memory barrier is emitted before the kernel traverses the linked list to copy each page from the shadow copy to the primary copy.

## Evaluation Methodology

This section discusses how the PMO system was evaluated, as well as the testing platform used.

[4]With 5-level page tables, Ice Lake-SP and newer can access up to $2^{56}$ virtual addresses.

### *Correctness and Crash Consistency*

To verify that the PMO implementation is crash consistent, a `panic()` is inserted into psync immediately after the persist stage, but before the copy stage, which generates a kernel panic, and forces the system to crash. When the system is restarted, the PMO is reattached and its contents examined, revealing that the data from the previous psync are in the PMO, as expected. Inserting a panic immediately before the persist stage results in the data from before the psync remaining at attach.

### *Performance Assessment*

In order to test PMO system performance, two schemes are compared. The baseline scheme has *no crash consistency* (NCC) and represents an ideal performance case that is functionally incorrect. NCC uses the ext4-dax filesystem which uses Intel libpmem's `pmem_persist()` to persist updates without any crash consistency. A crash with NCC may cause data corruption with dangling or invalid pointers, from which the original data structure may be unrecoverable. The second design to be compared is the state-of-the-art crash consistent filesystem, NOVA-Fortis [71]. NOVA-Fortis employs snapshots to support crash consistency. It is important to note that NOVA-Fortis only guarantees crash consistency of the file system, but does not guarantee crash consistency of application data. The PMO system is compared against NCC and NOVA-Fortis in terms of execution time and I/O bandwidth of specified workloads; the thread scalability and synchronization-rate sensitivities are evaluated. Finally, the overhead caused by adding integrity checking and/or encryption to the PMO system is determined.

The system described in Table 2.2 and depicted in Figure 2.8 was used in the evaluation. The PMO system was implemented on a modified version of Linux Kernel v5.14.18; a stock version of the

Table 2.2: Configuration of the PM system used for evaluation. Note that the OS differs from Chapter 2's Table 3.3.

| Component | Specifications |
|---|---|
| Motherboard | Dual socket Supermicro X11DPi-NT (w/ADR) |
| CPU and Clock | $2\times$Intel Xeon Gold 6230, 20 cores, 40 threads; $2.1$GHz ($3.9$GHz Boost) |
| CPU Cache | L1: 32KiB; L2: 1MiB; L3: 27.5MiB |
| DRAM and PM | $4 \times 32$GiB DDR4 @ $2666$MHz; $4 \times 128$GiB Intel Optane DC (PMem) |
| OS and Kernel | $64$-Bit Fedora $33$; Linux 5.14.18 (PMO, NCC), 5.1.0 (NOVA) |

Linux kernel v5.14.18 was used to evaluate NCC ext4 with dax. The Linux Kernel v5.1.0 was used for NOVA-Fortis, as it is the latest version NOVA-Fortis supports. The Linux distribution was Fedora 33[5].

*Microbenchmarks*

To measure PMO performance from the perspective of *execution time*, an OpenMP version of LU decomposition provided by [43], (and originally from the SPLASH benchmark suite [60]), a 2D-Convolution (2dConv) benchmark and a Tiled Matrix Multiplication (TMM) benchmark, taken from a recent PM study [18] are used. LU, 2dConv and TMM are run with matrix sizes of $3584 \times 3584$ doubles, $4096 \times 128$ integers, and $3072 \times 3072$ integers, respectively.

These benchmarks were ported by replacing their dynamic memory allocation calls (e.g., malloc and calloc) with a pair of pcreate and attach (to create and map into the process' address space a PMO of required size) and with pmem_map_file for NOVA-Fortis. Each benchmark ported to use PMOs uses multiple PMOs determined by the number of memory allocation calls in the original version. At the end of each iteration of the performance critical loop in the benchmarks, if a specified time duration $\Delta$ has elapsed from the previous invocation of the synchronization, a syn-

[5]Note that Chapters 3 and 4 use AlmaLinux instead


Figure 2.8: A picture of the PM system used for evaluation

chronization point (i.e., psync in case of PMO, generating a snapshot in case of NOVA-Fortis, and invoking pmem_persist in case of NCC) is inserted. Varying $\Delta$ varies the synchronization rate.

### *Filebench*

Filebench benchmarks [66], which represent I/O intensive real-world applications, are used for measuring I/O bandwidth performance. These benchmarks are ported to use PMOs by replacing files with PMOs of respective sizes. For ext4-dax (i.e., NCC) and NOVA-Fortis files are mapped via DAX. Furthermore, synchronization points are inserted in the benchmarks after **every** update (i.e., *append* or *wholefilewrite* flowop, in Filebench's terminology). Also, to avoid races between threads, a pthread barrier is emitted before and after each psync. Each workload was run twice for ten minutes, and the result is the averages between the runs. Each workload has a different percentage of write operations: FileServer (FS) is $67\%$ writes, VarMail (VM) is $50\%$, WebProxy (WP) is $16\%$, and WebServer (WS) is $9\%$.

### *Encryption and Memory Integrity*

In addition to testing the PMO system performance with crash consistency, the overhead compared to a baseline crash consistent design when adding encryption and data integrity is determined. As with the crash consistency evaluation above, encryption and integrity are evaluated with Filebench; and done so by adding new attach/detach calls between each file operation, rather than simply at the beginning and end of the benchmark. Note that this lowers the performance of Filebench: on average, Filebench with attach/detach calls between each operation is about $\frac{1}{3}$ of the performance without. Adding attach/detach calls between each operation makes the microbenchmarks too slow to obtain useful results, so this evaluation is not performed for them.

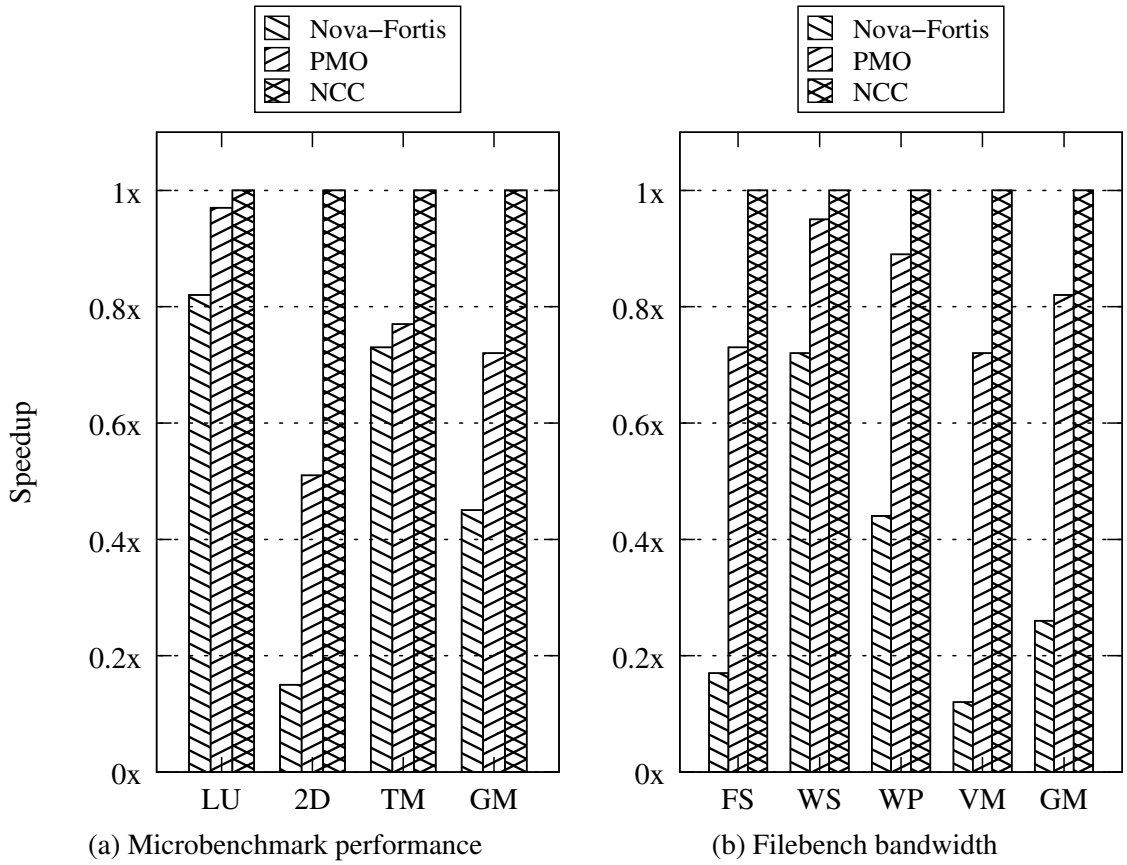
## Evaluation Results

This section seeks to answer several questions. How much performance overhead does the PMO system incur compared to a non-crash consistent (NCC) and a crash-consistent system not using PMOs? How scalable is the system, as the number of threads increase, and as the frequency of psync increases? Is encryption and integrity verification effective against at-rest attacks, and what penalty do they incur upon performance? Since there are no existing object-based abstractions providing intrinsic support for crash-consistency, the PMO abstraction is compared with NOVA-Fortis, a state-of-the-art filesystem.

### *Performance Evaluation of Crash Consistency*

Figure 2.9a compares the performance of crash consistent systems i.e., PMO and NOVA-Fortis with the Non-Crash-Consistent (NCC) system i.e., ext4-dax. Results are normalized to NCC and reported for benchmarks executed with 16 threads while synchronization is performed at rate of $4\times$ per second. When compared with NCC, the PMO system slows down the execution time by only $\approx 27.8\%$ (geometric mean) vs. $\approx 55.1\%$ with NOVA-Fortis. This indicates that the PMO system is not much slower than NCC and is $\approx 1.61\times$ faster than NOVA-Fortis.

This can be attributed to the fact that unlike NCC, crash consistent systems employ additional mechanisms to support crash consistency (i.e, shadowing with PMOs, and snapshots in NOVA-Fortis). Since it synchronizes at each synchronization point in a benchmark, only those PMOs that are actively used by the benchmark, the PMO system performs better. On the other hand, NOVA-Fortis takes a crash consistent image of the whole filesystem at each synchronization point and not only the files that are in active use. This illustrates the strength of an application-centric approach for crash consistency.




(a) Microbenchmark performance

(b) Filebench bandwidth

Figure 2.9: PMO performance compared to NOVA-Fortis and Ext4-DAX

*I/O Bandwidth and Crash Consistency*

Figure 2.9b compares the I/O bandwidth of different Filebench workloads achieved by the PMO system, to the I/O bandwidth of NCC and NOVA-Fortis. Results are normalized to NCC and reported for 16 threads with synchronization performed on every update operation. On average, shown by the geometric mean (GM) bar, PMOs and Nova-fortis provide crash consistency at the expense of losing $18.3\%$ and $74.4\%$ bandwidth, respectively. This result means that the PMO system achieves bandwidth $\approx 3.2\times$ higher than NOVA-Fortis. Performance of both PMOs and NOVA-Fortis vary across benchmarks as each benchmark has a different number of synchroniza-

tion points in accordance with their write percentage. More frequent synchronization incurs more overhead and thus lower bandwidth performance.

*Thread-scalability*

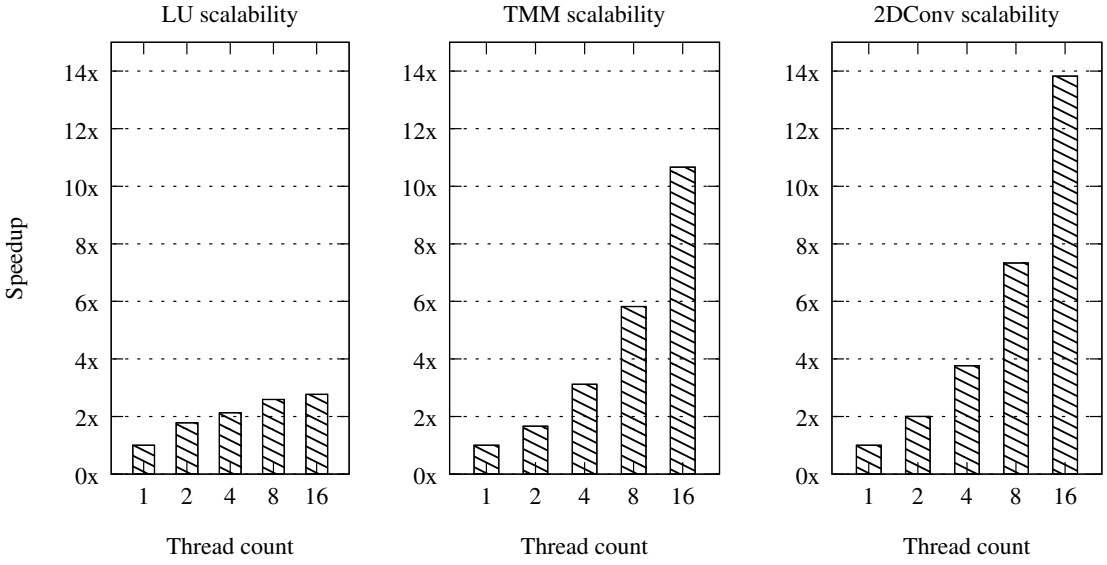



Figure 2.10: Thread-scalability of the PMO system

Figure 2.10 shows the thread-scalabilty of the PMO system's performance. Results, normalized to a single thread, are shown for 2DConv, TMM and LU workloads when executed with $N(= 1, 4, 8, 16)$ threads and synchronized four times per second. Results show that performance scales with increasing number of threads. However, the rate of scaling decreases from 2DConv to TMM, and LU. This is explained by the number of pages updated per synchronization operation (work assigned to each thread) in each workload. These are $184$, $9216$, and $24451$ pages for 2Dconv, TMM and LU, respectively.

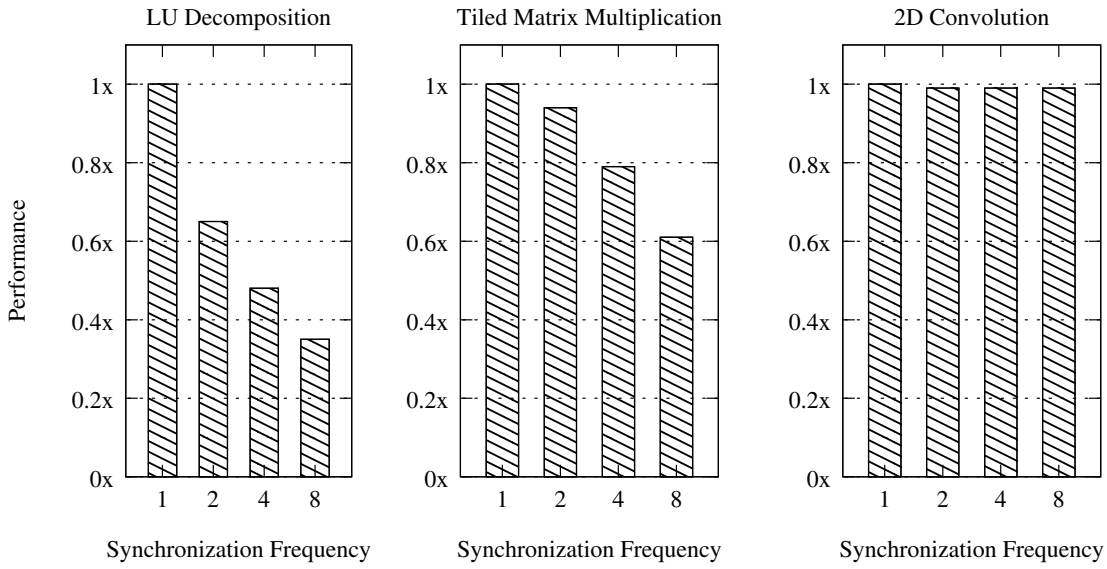



Figure 2.11: Synchronization-sensitivity of the PMO system

*Synchronization Frequency Sensitivity*

Figure 2.11 shows the sensitivity of the PMO system's performance to the frequency of psync. Results, normalized to 1 psync/sec, are shown for 2DConv, TMM and LU workloads executed with 16 threads and $N(= 1, 2, 4, 8)$ psync/sec. The results show that the performance of LU and TMM degrades rapidly as the number of times psync is invoked increases. This is for the same reason that the rate of performance scaling decreases in Figure 2.10: more pages are updated per synchronization operation with LU and TMM.

*I/O Bandwidth of Encryption and Memory Integrity*

Figure 2.12 shows the impact of the security scheme on the bandwidth achieved with Filebench. The figure shows bandwidth for four systems: an insecure baseline (BASE) to which all others are normalized, PMO with integrity verification only (INT), PMO with encryption only (ENC), and

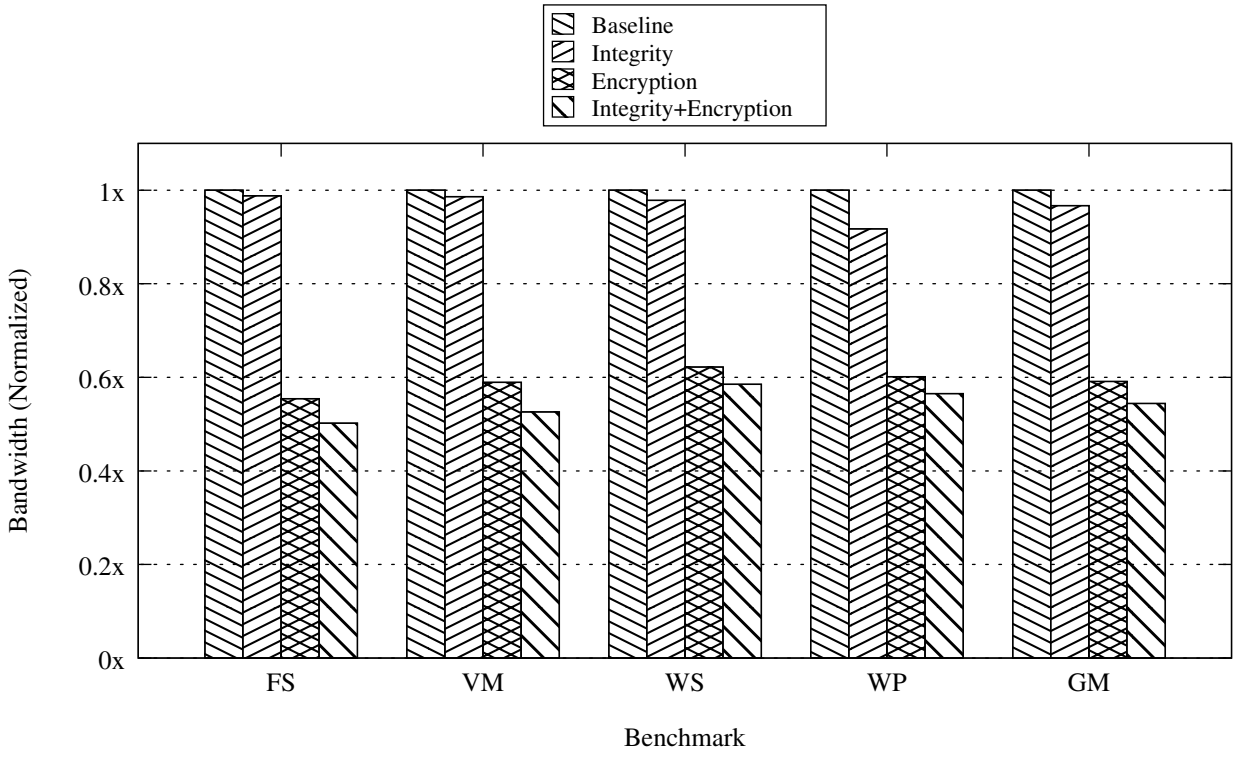



Figure 2.12: Bandwidth comparison of attach/detach PMO

both integrity verification and encryption (INT+ENC). Results are obtained with 16 threads. The figure shows that integrity checking incurs a small overhead (geometric mean of $3\%$), but encryption incurs a substantial overhead (geometric mean of $41\%$). Together, integrity and encryption lower the bandwidth by $46\%$. ENC is more expensive due to the fact that encryption/decryption affects both primary and shadow copies of a PMO, and is performed on the whole PMO. This result indicates that only what is needed should be applied; for example, the programmer should choose only INT if data secrecy is not important. This result also points to the idea that performing decryption at a smaller granularity is a better solution; a solution investigated thoroughly in Chapter 3.

*Security Evaluation of Encryption and Memory Integrity*

To evaluate the strength of the PMO approach to protect against at-rest PMO attacks, an experiment to detect whether the PMO approach can prevent unauthorized disclosure and detect unauthorized modifications of at-rest PMO data is undertaken. Initially, $10^3$ PMOs are created, and a secret is written into a random selection of them. An attacker (in this case, a malicious Linux kernel module) selects multiple PMOs at random and discovers its physical address by reviewing the metadata hashtable. The kernel module then maps the selected PMO into kernel address space via `memremap()`, and either compares the data within the PMO with the expected secret (i.e., the module attempts an unauthorized disclosure) or attempts to modify the data within the PMO (i.e., the module performs an unauthorized modification). The module keeps track of each time the disclosure is attempted, each time the disclosure reveals a secret, and each time an unauthorized modification occurs.

A user process, using the same seed as the kernel module, invokes attach on the same random selection of PMOs. The process tracks when the PMO subsystem detects that an unauthorized modification has occurred. The effectiveness of the PMO approach is determined by calculating the ratio between the number of detected unauthorized modifications or unauthorized disclosures, divided by the total number of attacks.

When not using encryption, the kernel module discovers $100\%$ of the secrets (i.e., all of the PMO's secrets were leaked). When using encryption, the kernel module discovers $0\%$ of the secrets (i.e., no secrets are leaked). When using integrity verification, $100\%$ of the attaches fail (i.e., the kernel detects data corruption in all of the affected PMOs). These results demonstrate that the PMO design is effective against both at-rest disclosures and at-rest modifications of data.

## Conclusion

Security and programmability are two important requirements for the design of a widely acceptable crash consistent object-based abstraction of persistent memory. This chapter discussed the design and implementation of a secure persistent memory objects (PMO) system with intrinsic support for crash consistency. Results show that the crash consistent PMO system performs $1.67\times$ and $3\times$ faster, for two sets of evaluated benchmarks, compared to the state-of-the-art file-based competitor NOVA-fortis. Security adds an overhead of $3\%$ when protecting PMOs at-rest only from corruption, $41\%$ when protecting from disclosure only, and $46\%$ for both.

## Acknowledgments

The material from this chapter was supported in part by the Office of Naval Research (ONR) under grant N00014-20-1-2750, and by the National Science Foundation (NSF) under grant 1900724.

# CHAPTER 3: LOAPP: LOW-OVERHEAD AT-REST PMO PROTECTION

This chapter expands on the work of Chapter 2 by improving the performance of PMOs without affecting security. This chapter observes that the performance of secure PMOs (with encryption and integrity verification) is lacking, especially with larger PMOs, because the system uses whole PMO encryption/decryption with integrity verification (WEDI). This chapter addresses this problem with WEDI for the first time. First, it observes that a PMO can be broken into pages and that demand paging can be adopted for PMO encryption (per-page encryption). Second, this chapter explores the design space of per-page PMO encryption and integrity verification, which this chapter refers to as Low Overhead at-rest PMO Protection (LOaPP), and discusses the trade-offs of each design. Third, this chapter introduces a crash handler to ensure that PMOs are always secure, even in the face of crashes. The new design, with per-page encryption alone, outperforms whole-PMO encryption without integrity verification (WED) by $1.4\times$ and $2.6\times$ for two sets of evaluated workloads. Adding per-page integrity verification on top of per-page encryption outperforms the original WEDI design by $2.19\times$ and $2.62\times$. [1]

## Introduction

Persistent Memory (PM) offers byte-addressability, high data densities, and low latency access [78]. These features make PM a preferred choice to hold persistent data compared to a storage device. While the first commercial PM (Intel Optane) was recently discontinued, other products are entering the space, including CXL Persistent Memory [14, 7] and a battery-backed SoftPM approach [74]. To manage PM-resident data, prior work has proposed two main approaches: either host a file system or memory mapped files within PM [37, 72], or consider PM as a repository of

[1] The contents of this chapter are based on [26].

Persistent Memory Objects (PMOs) [73, 76, 5].

A PMO holds persistent data in pointer-rich data structures without the backing of a file, while the operating system kernel manages the namespace, permissions, and sharing semantics of PMOs. A PMO is mapped/unmapped to/from the address space of a user process by invoking `attach()`/`detach()` system calls [27]. Once attached, PMO data is accessed directly as if it were traditional volatile memory. PMO systems provide the `psync()` system call as a primitive to persist data and to manage crash consistency: any modifications to a PMO are not made durable until `psync()`, and a crash will result in the PMO being restored to the last durable state of the most recent `psync()`. PMOs support key-based protection where a successful attach requires that a user-supplied key must match the system-stored key. Once attached, the granted access permissions are enforced by the kernel throughout the attach session. Like a file, a PMO can outlive the process that created it and survive system boots; like traditional pointer-rich data, a PMO can store buffers, pointers, and other assorted data structures.

A PMO can be shared among multiple reader processes simultaneously, but a writer process must have exclusive PMO access. Once created, a PMO is either *in-use*, i.e., attached to a user process or *at-rest*, i.e., not attached to any user process. Like files, PMOs are likely to spend most of their lifetime at-rest, holding the persistent data of user processes. This makes PMOs susceptible to data remanence attacks, where unless deleted, PMO data remains in plaintext in PM for a long time. Another vulnerability is when the adversary compromises the OS kernel to steal or corrupt a PMO's data. This results in the adversary stealing or corrupting a PMO's data without being noticed by the user processes accessing those PMOs. Therefore, protecting at-rest PMOs is as important as protecting in-use PMOs.

A key problem with WEDI is that the latency of `attach()`/`detach()`/`psync()` system calls scales proportionally to the size of a PMO [27], and the latency is difficult to hide. An `attach()`

call is on the critical path of the requesting process as a user process cannot access the PMO until it is mapped to its address space. The latency of `psync()` is also in the critical path because a consistent state must be achieved prior to continuing computation past a `psync()`. While the latency of `detach()` may be off the critical path, a subsequent `attach()` must be delayed until `detach()` completes, hence potentially exposing it. These mostly exposed and non-scalable latencies present an impediment to achieving better security even for in-use PMOs, since a prior work proposed frequent attaches and detaches to improve security [73]. They result in a higher total execution time of a user application with WEDI as compared to No Encryption, Decryption and Integrity verification (NEDI).

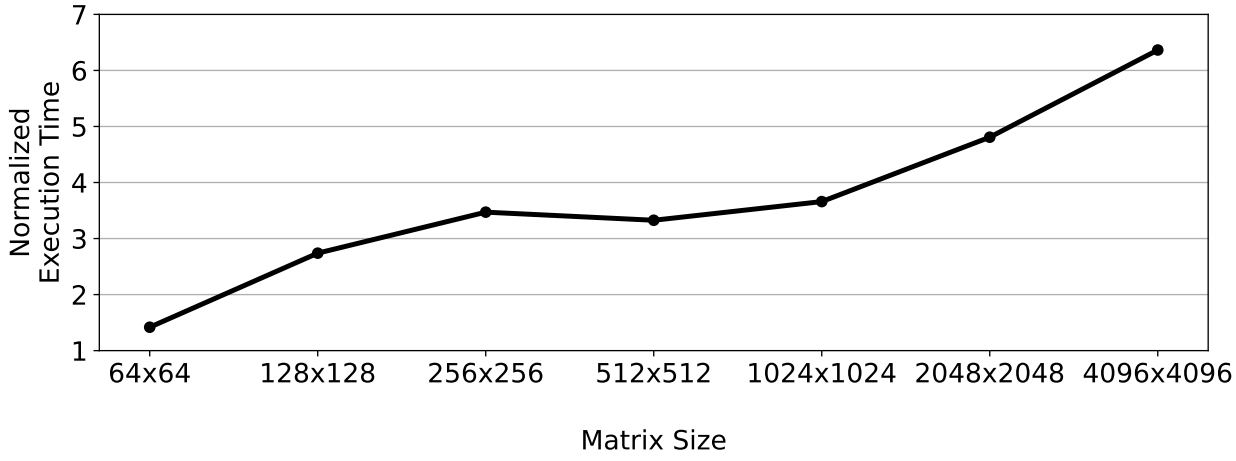



Figure 3.1: Execution time of the Tiled Matrix Multiplication benchmark with WEDI.

To illustrate the extent of the problem, Figure 3.1 shows the execution time of a PMO-ported Tiled Matrix Multiplication (TMM) benchmark [19] with WEDI, normalized to those of NEDI for each corresponding size. The x-axis shows the effective size of the PMO holding the input matrix for the workload. The matrix size within the PMO increases from $64 \times 64$ items to $4096 \times 4096$ items (i.e., the PMO increases from $64 \times 64 = 4096$ bytes to $4096 \times 4096 = 16$MB). Each iteration of the performance critical loop attaches the PMO, updates an array element, psyncs the PMO, and

finally detaches it. The figure shows that the overhead of WEDI rapidly increases along the PMO size; from $1.5\times$ at $4096$ bytes to $6.4\times$ at 16MB.

The source of WEDI's non-scalable performance is that it enforces protections (encryption, decryption, and integrity mechanisms) at the granularity of a whole *PMO* size, irrespective of the actual working set size of a single attach session.

Thus, this chapter proposes a new approach, referred to as *Low-Overhead at-rest PMO Protection (LOaPP)* ("Low-App") scheme without lowering the security level. The key idea is to reduce the PMO overheads by protecting PMO data at a finer granularity (i.e. pages) and paying the protection costs only when data are actually accessed.

While conceptually simple, this chapter has to deal with several challenges. Note that PMOs use a shadowing approach, where each primary page is backed by a shadow page [27], to guarantee crash-consistent atomic updates. An important question is *what* to decrypt/verify/encrypt for low-overhead? I.e., shadow pages, or both shadow and primary pages? Another challenge is *when* to decrypt/verify/encrypt a PMO during an attach session. Finally, there is the question of how to ensure that LOaPP maintains the *checksum integrity* of PMOs. This chapter conducts an extensive design-space exploration to address these challenges.

This chapter makes the following **contributions**:

1. This chapter proposes a novel, *Low-Overhead at-rest PMO Protection (LOaPP)* scheme.
2. This chapter presents an exploration of LOaPP's design space with several performance optimizations to reduce the protection overhead.
3. This chapter implements LOaPP on a Linux kernel, with a real system equipped with Intel Optane PMem. Extensively evaluating LOaPP on several workloads reveals that the most

performant design is $2.19\times$ faster than WEDI. For Filebench, the most performant design is $2.62\times$ faster than WEDI.

## Background

This section discusses the necessary concepts and related work required to more fully understand per-page encryption with PMOs, including an overview of PM and the general PMO subsystem design. See Chapter 2 for more details on the PMO system used here.

### *Persistent Memory*

Persistent Memory (PM) is a class of memory technologies that retains data after power loss, is byte addressable, has access latency closer to volatile memory than storage devices, and is more dense than volatile memory. Given its characteristics, PM becomes attractive for hosting small and medium sized persistent data, while large data is likely to still rely on block-based storage. To manage persistent data residing on PM, one approach is to use a filesystem. Designs based on this approach store persistent data in files and then map them to the address space of a user process [10, 37, 72]. However, this approach must keep both memory metadata and file metadata consistent and reconcile their semantics. Furthermore, the approach creates large ($\approx 13\times$) overheads compared to raw persistent memory device write bandwidth [37].

### *Persistent Memory Objects*

An alternative approach of managing PM is to view it as a collection of Persistent Memory Objects (PMOs) [5, 41, 73, 76]. A PMO holds persistent data stored in potentially pointer-rich data

structures, while the kernel manages the namespace, permissions, and sharing semantics of PMOs. Though LOaPP is applicable to any PMO system, LOaPP is implemented on the Greenspan PMO (GPMO) system[27] (See Chapter 2 for more details). The GPMO system is the only available PMO system that can work with a real operating system on real hardware. PMO system calls for the GPMO system are shown in Table 3.1. A user process creates a PMO of a given name, size and key by invoking the `pcreate()` system call, while `pdestroy()` deletes a PMO and reclaims its space. Once created, a PMO is mapped into the address space of a user process by invoking `attach()`, making it accessible to the process. Conversely, invoking `detach()` unmaps a PMO from the address space of a process, making it inaccessible. Any PMO updates, performed after attaching it, are persisted in a crash-consistent way by invoking the `psync()` system call. When `psync()` is invoked, the kernel uses shadowing to achieve crash-consistent persistency of PMO updates.

Table 3.1: PMO system calls.

| **Primitive** | **Description** |
|---|---|
| pcreate(name,size,key) | Create a PMO `name` of `size` and `key`. |
| pdestroy(name,key) | Given a `key`, delete PMO `name`, and reclaim its space. |
| attach(name,perm,key) | Render accessible the PMO `name`, given a valid `key` with permissions `perm`. |
| detach(addr) | Render inaccessible the PMO `addr` points to. |
| psync(addr) | Persist updates to the PMO `addr` points to. |

### *Crash-Consistency*

Crash-consistency is an important requirement for managing persistent data on PM regardless of whether data is stored in memory-mapped files or persistent memory objects. An application, system, or power failure may cause partial or unordered writes to persistent data; in the absence of crash consistency, this can lead to the data being in some inconsistent state from which it cannot

be restored. Logging and shadowing are two popular approaches for achieving crash-consistency. The GPMO system uses the latter whereby it maintains a primary and shadow copy of each PMO. The system provides a fundamental guarantee that, even in the case of a crash, either the primary or the shadow PMO remains consistent (i.e, free of partial and unordered writes). Based on the PMO's state at the time of a crash, one of the two copies are used to restore the PMO to a consistent state afterwards.

## Related Work

### *Hardware support for memory encryption*

There is hardware support for memory encryption; some examples include AMD's Secure Memory Encryption (SME) [39] and Intel's Total Memory Encryption - Multi Key (TME-MK) [32]. Additional related work include Secure Enclaves such as Intel's Software Guard Extensions (SGX) [12] and AMD's Secure Encrypted Virtualization (SEV). While theoretically these solutions could be used to accelerate PMO performance and improve security, this chapter does not use them for several reasons.

First, AMD's SME is lacking because it uses a single securely generated key and encrypts the entire virtual address space. Under this scheme, the whole PM may be encrypted, but isolating one PMO from another through whole-memory encryption keys is impossible.

Second, although Intel's TME-MK design seems more promising, since different parts of memory can be encrypted with different keys, it suffers from a limited number of supported domains. For example, according to Intel's own specifications [32], a system supporting TME-MK has an *absolute theoretical limit* of $31,999$ domains that may be supported; but since it is anticipated that PMOs will be small, this limit is likely to be easily met and exceeded (for example, if each PMO is

4096 bytes large, the limit is exhausted after 128 GB). Furthermore, this limit is described by Intel as the *maximum* possible value supported by the CPU model specific register (MSR) for TME-MK; currently available systems support several orders of magnitude fewer domains [77]. Finally, TME-MK is only available on 3rd Generation Scalable Xeons or newer, and it is exclusive to Intel, whereas this chapter is aiming for a design that is platform agnostic. For all of these reasons, this chapter does not use TME-MK.

Finally, while secure enclaves provide the ability to completely isolate processes from one another, AMD's SEV and Intel's SGX solutions are lacking. First, they are vulnerable to Spectre and Meltdown-style attacks [9, 52]; second, Intel's current implementation is limited to only 93MB; third, Intel's support for it has been deprecated on modern processors [33], while AMD's SEV is designed per virtual machine rather than per process [39].

### *PMO Security*

Prior work with PMOs has largely focused on protection of PMOs *in-use* (i.e., while they are attached). For example, [77, 73, 75] all focus on protecting in-use PMOs from attacks. MERR [73] proposes attaching a PMO only when needed and otherwise keeping a PMO in a detached state, while Xu et al. [77] proposes extending the MERR approach to use Intel Memory Protection keys (MPK) to limit access of PMOs only to those threads that require access (i.e., isolating PMOs between threads). TERP [76] expands on MERR by providing a compiler pass to perform automatic attaches and detaches.

Mustafa et al. [51] demonstrated that the use of PMOs break inter-process isolation, at least so long as a PMO is shared between multiple processes; Mustafa and Solihin [50] expanded on this to show that inter-process isolation can be broken even without sharing a PMO over time, so long as the processes are linked in some way.

## Threat Model

The threat model's goal is to protect at-rest PMO data to provide confidentiality and integrity. Note that this threat model is different from [73, 51, 50], as they require a PMO to be in-use to be exploited for a security attack. To fully protect a PMO, protection for at-rest PMO data can be paired with other protection schemes for protecting in-use PMO data.

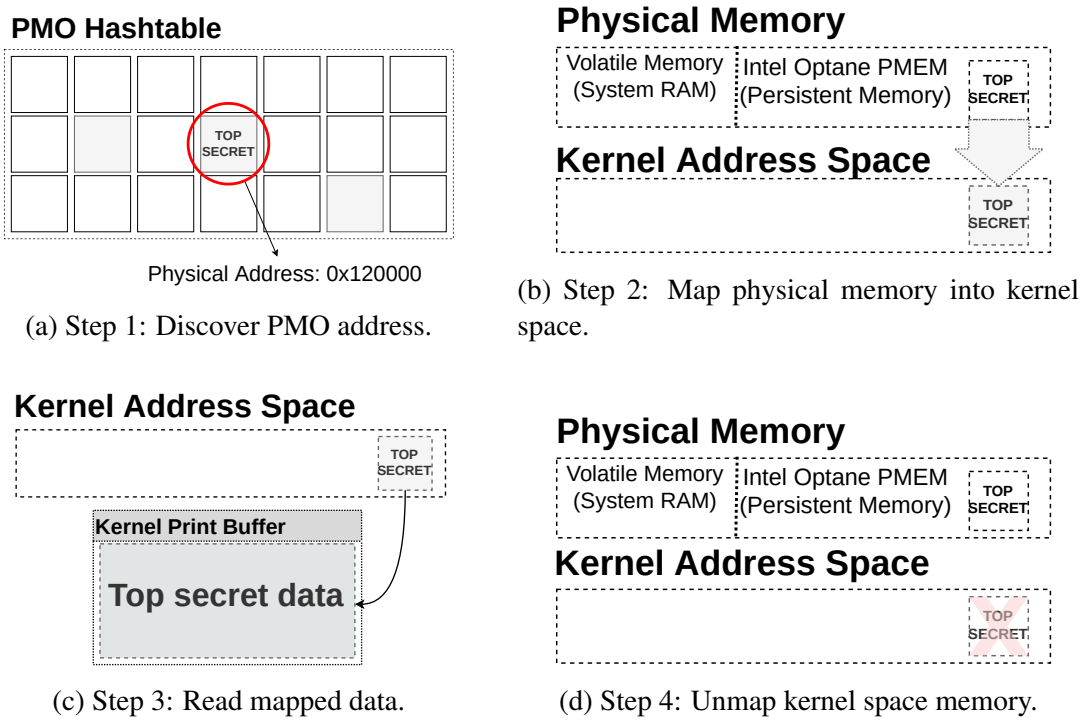
The attacker's goal is to either reveal or tamper with the confidential data belonging to a user-process stored within an at-rest PMO. It is assumed that the attacker knows or has the capability to find the location of the target PMO in memory. Like files, PMOs are likely to spend most of their lifetime at-rest, holding the persistent data of user processes. One attack a PMO is susceptible to is data remanence, where unless deleted, PMO data remains in plaintext in PM for a long time. A stolen or improperly disposed of PM may be analyzed by the attacker to obtain sensitive data. Such attacks have been documented in data stored in files in hard drives [58]. Since then, filesystem encryption has become widely used to protect at-rest data in files (but no similar mechanism exists to protect at-rest data in PM).

Another attack to consider is an attack where the adversary compromises the OS kernel to steal or corrupt a PMO's data. Assume that the data structures residing in the PMO contain buffers and pointers, which may be targeted for overwrites by the compromised OS; this results in the adversary stealing or corrupting a PMO's data without being noticed by the user processes accessing those PMOs.

Trust is limited to specific components of the system software, notably the Linux Kernel Crypto API [11], crucial kernel memory functions like memcpy and memset, and the PMO kernel subsystem. These components can be assumed to be devoid of any code vulnerabilities, a plausible belief since their code sizes are small enough to undergo formal verification, which has been done

on similarly sized programs [44]. To illustrate, the Linux Kernel Crypto API for version 5.14.18 encompasses approximately 82500 source lines of code (SLOC), while the LOaPP PMO system contains about 2700 source lines of code, and the kernel memory functions comprises roughly 100 lines of architecture-specific inline assembly. This contrasts with the entirety of the kernel, which contains about 2.2 million lines. Consequently, the kernel subsystem, the critical memory functions (specific to architecture and written in assembly), and the Crypto API collectively contribute to a mere 0.4% of the overall kernel. Furthermore, the threat model trusts the encryption hardware of the CPU.

*Example Attack*




(a) Step 1: Discover PMO address.

(b) Step 2: Map physical memory into kernel space.

(c) Step 3: Read mapped data.

(d) Step 4: Unmap kernel space memory.

Figure 3.2: Steps of PMO example attack.

Figure 3.2, reproduced from Chapter 2 [27], demonstrates a data-disclosure attack on an at-rest PMO. Assume the attacker has already exploited a kernel code vulnerability to manipulate its control flow. In step a), the attacker locates the physical address of the targeted PMO by navigating through the GPMO metadata hashtable. In step b), attacker maps the target PMO into the kernel virtual address space (e.g., through `vmalloc`/`ioremap` in Linux). In step c), the attacker copies the PMO's contents into the kernel print buffer, thereby exposing confidential information. Finally, in step d), the attacker clears the print buffer and unmaps the PMO from the kernel address space, effectively erasing any trace of the attack. Note that an attacker can also perform a data-injection attack by writing into the mapped PMO in step c) and persisting the updates (e.g., by flushing the updated pages).

### Low Overhead at-rest PMO Protection Design

Consider two scenarios from the perspective of the threat model: 1) when at-rest PMOs need to be protected only against unwarranted reads and 2) when protection against both unwarranted reads and writes is required. When a design D protects against unwarranted writes through Integrity (I) verification with a checksum updated at a system call c, it is denoted as $D/I_c$. Otherwise, the design is denoted only as D.

This design space exploration is guided by finding answers to the two design challenges for at-rest PMO protection. **C1:** *what* should be decrypted, encrypted, or verified? **C2:** *when* to decrypt (D) a data item, verify (V), and update (U) its checksum, and encrypt (E) it during a PMO's attach session? **C3:** How to ensure the checksums are in a consistent state in the case of an application or system crash? This design space is shown in Table 3.2.

The Whole PMO Encryption, Decryption, and Integrity verification (WEDI) design of [27] pro-

vides at-rest PMO protection at the granularity of an entire PMO (C1). Since the approach provides integrity verification and updates the checksum of an attached PMO on a `psync()` system call, it is referred to as WED/$I_p$ for the rest of the chapter and is summarized by the first row of Table 3.2. WED/$I_p$ decrypts the entire PMO on an `attach()` request (C2.D). Recall that a PMO has both a primary copy and a shadow copy. Therefore, to achieve crash-consistent atomic PMO updates, the decryption is performed in two steps: first, the primary copy is decrypted and persisted into the shadow copy. Then, the shadow copy is copied back and persisted to the primary copy. The two-step process ensures that if a crash happens during the decryption step, at least one valid copy is available to recover the PMO to a consistent state. After decrypting, the checksum is computed on the decrypted PMO and verified against the stored one (C2.V). In the case of a successful verification, the PMO is mapped into the requesting process' address space. On `psync()`, after persisting all updates in the shadow copy, the PMO's stored checksum is updated (C2.U). On `detach()`, the shadow copy is first encrypted and persisted into the primary copy, and then the shadow copy is memset to zero and persisted (C2.E). Since the design always encrypts/decrypts a PMO in two steps, it offers an innate guarantee that the stored checksum will always match the data within the PMO (C3). More specifically, since the checksum is updated whenever `psync()` is, the checksum and the data within the primary copy will always match.

The rest of this section discusses the design of LOaPP starting from the base design to incrementally optimized versions.

### *Encrypting/Decrypting Both PMO Pages (BP)*

WED/$I_p$ decrypts, verifies and encrypts all pages of a PMO, including those that are never accessed by the attaching process. This contributes to increasing the latency of all three system calls: `attach()` decrypts all the pages and then verifies the checksum at the level of the PMO,

Table 3.2: LOaPP's design-space exploration.

| Design | C1 | C2 | | | | C3 |
|---|---|---|---|---|---|---|
| | | D | V | U | E | |
| WED/$I_p$ | PMO | attach | -/attach | -/psync | detach | innate |
| BP/$I_p$ | Both Pages | PF | -/PF | -/psync | detach | innate |
| BP/$I_d$ | Both Pages | PF | -/PF | -/detach | detach | CrH |
| SP/$I_p$ | Shadow Page | PF | -/PF | -/psync | psync | innate |
| SP/$I_d$ | Shadow Page | PF | -/PF | -/detach | psync | CrH |

`psync()` updates this checksum, and `detach()` encrypts all the pages. To avoid paying the extra cost, the Low Overhead at-rest PMO Protection (LOaPP) design reduces the protection granularity to the *page-size* and decrypts PMO pages only on-demand. On an `attach()`, if the requested PMO is not already attached to another process and the user-supplied key matches the system-stored key, the call returns control immediately to the requesting process. However, the PMO is not mapped into the process' address space until a Page Fault (PF) happens. On a PF, the fault handler routine decrypts both the requested primary page and its shadow copy in a two-step crash consistent way (same as described for WED/$I_p$) i.e., C1 and C2.D. Since this design decrypts both pages, it is referred to as BP. On `detach()`, the dirty pages that were already persisted in the shadow copy of the PMO by the last `psync()` are then encrypted and persisted into the corresponding primary copy (C2.E). The shadow copy is then destroyed, by calling `memset()` on the shadow pages, setting their values to 0, and then persisting. Since BP, like WEDI/$_p$, always perform decryption/encryption of PMO pages in two steps, it always maintains a valid copy of each PMO page, a PMO page that is free of any partial updates. Therefore, the design provides innate guarantees of crash-consistency (C3). Note that BP alone *does not provide integrity protection* against unwarranted writes.

BP offers several performance benefits: 1) By adapting on-demand paging, it does not decrypt/encrypt a PMO page until the Page Fault (PF) happens. This not only significantly reduces the *attach()/detach()*

latency but also avoids unnecessarily delaying any non-PMO computation following an *attach()*. 2) By lowering protection granularity to the sizes of pages, unlike WED/$I_p$, BP decrypts/encrypts only those pages that are actually accessed in an attach session and avoids paying the extra protection cost for unaccessed pages.

*BP with Integrity verification, checksum updated on* `psync()` *(BP/$I_p$)*

To protect against unauthorized writes in addition to unauthorized reads, it is necessary to add integrity verification to the BP design. However, unlike the PMO-level checksum of WED/$I_p$, the new design maintains a per-page checksum. The design allows a decrypted page to be accessed by the requesting process only when the page fault handler verifies that the faulted (and decrypted) page's computed checksum matches with its stored checksum (C2.V). The checksum of *only* dirty PMO pages are updated on a `psync()` (C2.U), hence the design is referred as BP/$I_p$. Since per-page checksums are updated by means of a crash-consistent `psync()`, the design maintains the innate guarantee that the checksum and the data within the primary copy will always match (C3).

Note that maintaining per-page checksums increases the size of metadata per PMO (See the PMO system layout subsection). However, the cost of maintaining extra metadata is outweighed by the significantly lower latency of `psync()`, compared to WED/$I_p$, as it only updates the checksums of dirty pages instead of updating a single PMO-level checksum. The optimization is likely to result in significantly lowering the protection overhead for applications frequently invoking `psync()` in an attach session.

*BP with Integrity verification, checksum updated on detach (BP/$I_d$)*

While BP/$I_p$ significantly lowers the latency of `psync()` as compared to WED/$I_p$, it is still high compared to BP (which provides no integrity protection), especially when `psync()` is invoked more frequently in an attach session, as shown in Figure 3.7. Hence, the following question ought to be asked: Can a protection scheme be devised that still provides integrity verification, ensures crash-consistency but further lowers the latency of `psync()`? One option is to update the checksum of dirty pages on `detach()` (C2.U) while the dirty pages themselves are still persisted by `psync()`. This design is referred as BP/$I_d$. While BP/$I_d$ reduces the `psync()` latency, it creates a checksum-consistency problem: If a crash happens between a `psync()` and the `detach()`, on reboot, there is a mismatch between between a page's data (persisted at `psync`) and its checksum (updated at `detach()`). To address the issue, the BP/$I_d$ is equipped with a Crash Handler (CrH) routine that restores the checksum-consistency guarantee for BP/$I_d$ design (C3). The crash handler is discussed in a later section.

*Encrypting/Decrypting only Shadow PMO Page (SP)*

Unlike WED/$I_p$ that decrypts/encrypts the whole PMO on `attach()`/`detach()`, BP/$I_p$ and BP/$_d$ aim at reducing the `attach()`/`detach()` latency by decrypting/encrypting PMO pages only on-demand. BP/$I_p$ and BP/$_d$ have both the primary and shadow copies in decrypted form after a PMO Page Fault (PF)g. To reduce the latency of the two system calls even further,a design that decrypts the primary page on PF (C2.D) and persists it in the shadow page but *does not* copy back and persist the shadow page to the primary page is proposed. In other words, only the Shadow Page is decrypted/encrypted (C1), hence the design is referred to as SP. Like BP, SP also verifies the checksum of a page on PF (C2.V) and updates it on `psync()` (C2.U).

The SP design involves a trade-off: Since a shadow page is in decrypted form while its corresponding primary page is in encrypted form, a `psync()` operation *after* persisting a shadow page *must encrypt* and persist the shadow page into the primary page (C2.E). The additional operation of encryption *increases `psync()` latency*. Since updates are always guaranteed to be encrypted and persisted in the primary page by `psync()`, SP can simply zero the shadow pages on `detach()` and free them. As the updates are persisted by already crash-consistent `psync()`, SP maintains the innate guarantee of crash-recovery (C3). Note that the SP design is likely to reduce the protection overhead when an application attaches/detaches a PMO more often than psyncing a PMO (e.g., frequent read-only PMO attach-sessions).

*SP with Integrity verification, checksum updated on `psync` ($SP/I_p$)*

Just as $BP/I_p$ adds integrity protection to BP, $SP/I_p$ adds integrity protection to SP with a page's checksum updated on `psync()` (C2.U). The $SP/I_p$ design maintains an innate guarantee that the checksum will always match (C3).

*SP with Integrity verification, checksum updated on `detach()` ($SP/I_d$)*

$SP/I_d$ differs from $SPI_p$ in that it updates a page's checksum on `detach()` (C2.U), to further reduce `psync()` latency, but relies on the Crash Handler (CrH) routine to guarantee that the checksums on a detached PMO will always match the data stored within it (C3). $SP/I_d$ performs integrity verification in the same way as $BP/I_d$.

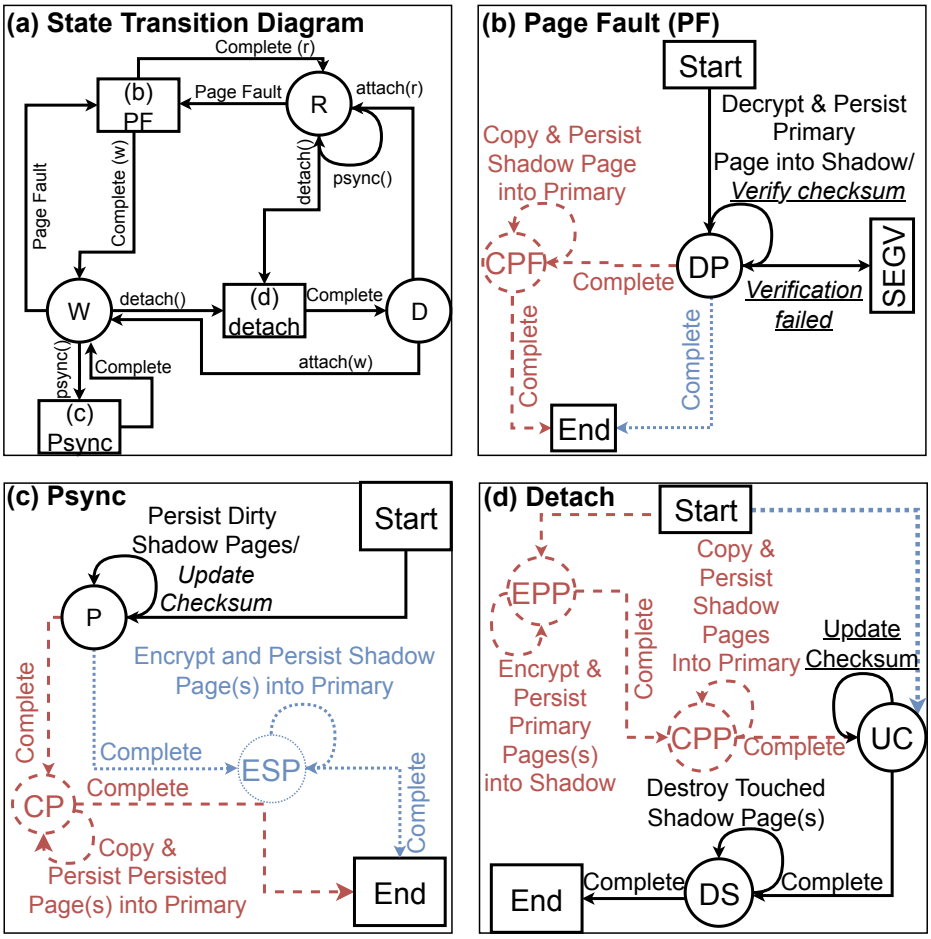



Figure 3.3: High-level state transitions for **BP** (red/dashed), and **SP** (blue/dotted)

*PMO State Transition Diagram*

The design of Low Overhead at-rest PMO Protection (LOaPP) keeps per-PMO state information. PMO states and transitions among them are shown in Figure 3.3 for all designs i.e., BP/$I_p$, BP/$I_d$, SP/$I_p$, and SP/$I_d$.

*State transition for BP, BP/$I_p$ and BP/$I_d$*

For the PMO state transitions in the BP, BP/$I_p$ and BP/$I_d$ designs, consider only the black/solid and the red/dashed parts of Figure 3.3. A PMO is initially in (D) (detached). On `attach(w)` or `attach(r)`, the PMO transitions (D) → (W) (write) or (D) → (R) (read). Note that BP maps PMO pages into the address space of the requesting process only on demand. That is to say, on a page fault, the PMO transitions (W) → (DP) or (R) → (DP) (decrypt and persist). In the (DP) state, the kernel decrypts and persists a primary PMO page into its corresponding shadow page, computes the checksum over the decrypted page and compares it to its stored checksum. In the case of a mismatch, a segmentation fault is reported (Note that checksum verification is not applicable to the BP design). Otherwise, the PMO transitions (DP) → (CPF) (copy and persist faulted) where the decrypted (and faulted) shadow page is copied back and persisted into the primary page. On completion, PMO transitions to the (R) or (W) state.

A `psync()` on a PMO in (R) triggers no state transition and nothing happens. A `psync()` on a PMO in (W) transitions the state (W) → (P) (persist), where all dirty shadow pages are persisted and their checksum is updated (only for BP/$I_p$). Once completed, the PMO transitions (P) → (CP) (copy and persist) where all persisted shadow pages are copied and persisted into their associated primary pages. Upon completion, the PMO transitions back to (W).

On `detach()`, the PMO states transitions in the same way for BP and BP/$I_p$ design, but differently for the BP/$I_d$ design. For BP and BP/$I_p$, `detach()` transitions a PMO (R) → (EPP) or (W) → (EPP) (encrypt and persist) where all touched primary PMO pages are encrypted and persisted into corresponding shadow pages. On completion, the PMO transitions (EP) → (CPP) (copy and persist to primary) where all persisted pages in the shadow PMO are also copied back and persisted into associsaated primary pages. On completion, PMO transitions to (Ds) (destroy) where now all uncessary shadow pages are zeroed. Finally, the PMO transitions (Ds) → (D).

For the BP/$I_d$ design, `detach()` transitions a PMO Ⓡ → (UC) (update checksum) where checksum of all PMO pages written in the attach session (i.e., zero for detach(r)) is updated. Once done, the PMO transitions to (Ds) (destroy) where now all unnecessary shadow pages are zeroed. Finally, PMO transitions (Ds) → Ⓓ.

*State transition for SP, SP/$I_p$ and SP/$I_d$*

For the PMO state transitions in the $SP$ design, SP/$I_p$ and SP/$I_d$ designs, consider only the black/solid and the blue/dotted parts of Figure 3.3. For the sake of brevity, only those transitions that are different from the corresponding BP designs will be explored.

On a page fault, after decrypting and persisting a faulted primary page to associated shadow page in (DP), the PMO transitions (DP) → Ⓦ or (DP) → Ⓡ. The decrypted shadow page is not copied and persisted back to the primary page (i.e., there is no (DP) → (CPF) transition). Recall that checksum verification is not applicable for the SP design. On `psync()` for PMO in Ⓦ, after persisting all dirty pages in the shadow (and updating their checksum for SP/$I_p$) in Ⓟ, the PMO transitions Ⓟ → (EPP) (encrypt and persist). In (EP), all the touched shadow pages are encrypted and persisted into associated primary pages. This is required to ensure that the primary PMO pages always remain encrypted. Upon completion, the PMO transitions (EPP) → Ⓦ. On `detach()`, for SP and $SPI_p$, the PMO transitions Ⓡ → (Ds) or Ⓦ → (Ds) to zero all touched PMO pages. Note that in all SP designs, it is guaranteed by the last `psync()` that all touched shadow pages have been encrypted and persisted into associated primary pages. Therefore, all shadow pages can be safely zeroed. However, on `detach()` for SP/I $_d$, the PMO transitions Ⓡ → (UC) or Ⓦ → (UC) (update checksum) where all checksums are updated for all touched PMO pages before the PMO transitions to (Ds) state to zero the pages.

*PMO System Layout with Integrity Verification*

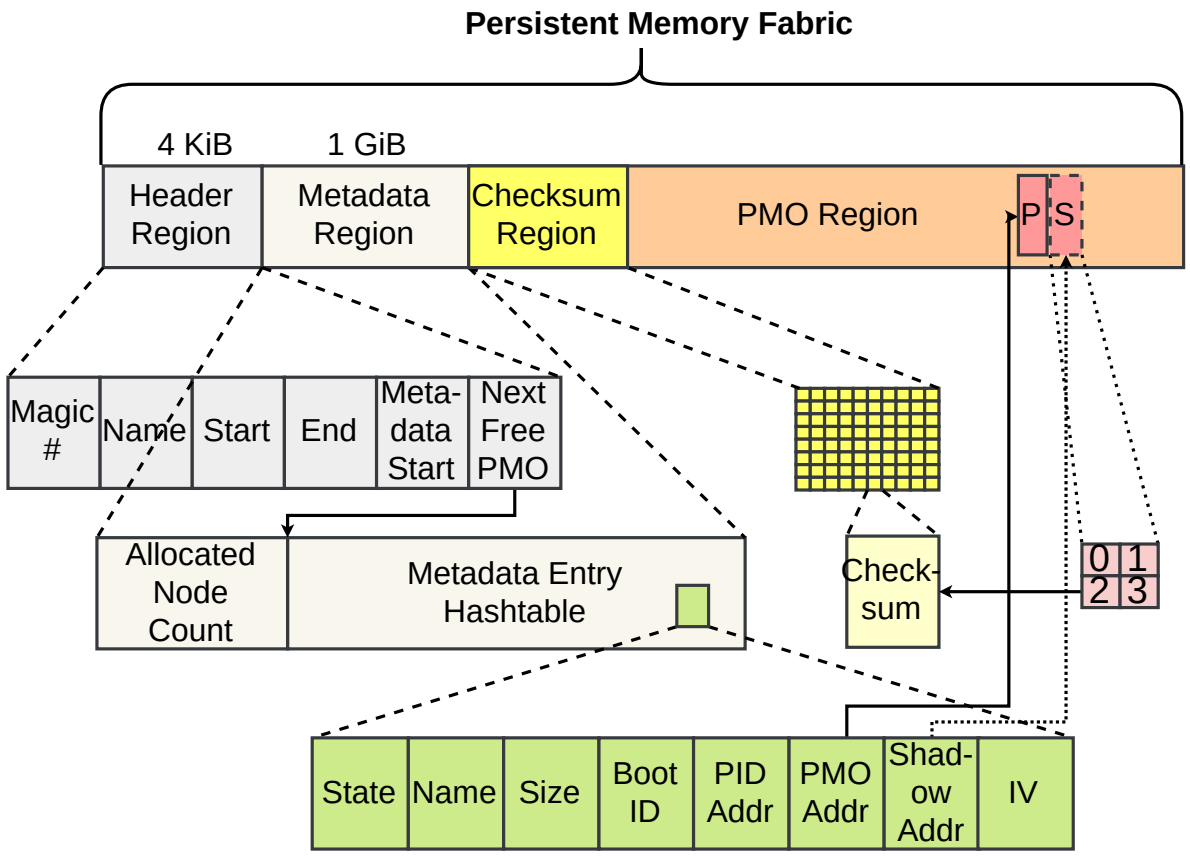



Figure 3.4: Enhanced PMO System Layout for per-page checksums. Modified from Figure 2.2 in Chapter 2.

Regardless of whether $I_p$ or $I_d$ is used, it is necessary to store the checksums of each page somewhere. As depicted in Figure 3.4, a new region, the *checksum region* is added to the PMO system. Each page has an associated checksum that points to an entry in this region. Since each checksum is $32$ bytes (the GPMO system uses SHA-$256$), this means that each $4096$ byte page has an additional $32$ byte overhead. Alternatively, the hash could have been added to the end (or beginning) of each page, but for simplicity and for performance reasons, this work stores them in a separate region of the PM. This overhead can be reduced with the use of compression.

### *Performance analysis*

Since detach and page faults are far simpler (simply drop the shadow, or decrypt the primary into the shadow), it is reasonable to expect both operations to be much faster with SP when compared to BP. On the other hand, encryption is not an entirely free operation, so encrypting the shadow into the primary at psync should take slightly longer. The evaluation section evaluates the performance impact of each design.

### *Crash handler*

In the prior PMO design [27], the PMO system only ensured that *crash consistency* was guaranteed in all circumstances. A PMO was only ensured to be encrypted if a PMO was detached successfully. This means that in prior work, a PMO at-rest was *not* guaranteed to be secure or to have its latest checksum (although it was always guaranteed to be crash consistent): a PMO, at-rest or in-use, was always crash consistent, but only secure when at-rest after a successful detach.

In contrast, it would be ideal to extend the PMO security domain to include some types of abnormal process termination, such as if power is lost or a process dies. In this case, a **crash handler** makes sense to be used. The crash handler performs encryption routines that would have been performed by `detach()` if the process had terminated normally. For brevity, only the crash handler with SP will be explored.

Figure 3.5 illustrates the crash handler's state transitions. Since the crash handler will run in the case of a process failure, it cannot know which pages are dirty and which are not. All the crash handler knows is the PMO state at the time of the crash (which is saved within the PMO metadata) and the data within each page. In the most common case, the PMO is in the read or write state (R/W) and is not being synchronized; in that case it is safe to simply drop all of the extant shadow

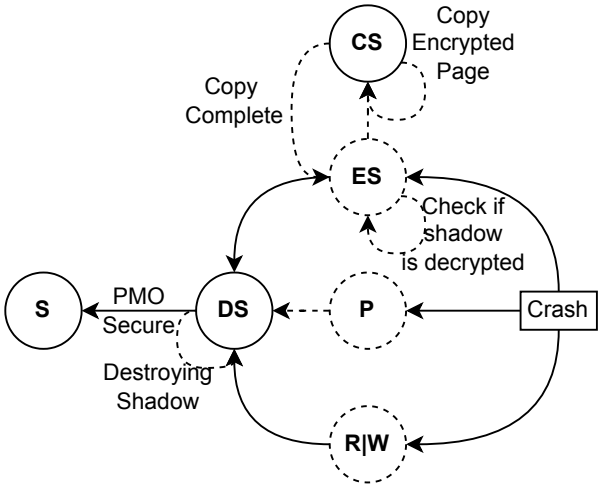



Figure 3.5: State transitions for the crash handler

pages (Ⓡ/W → Ⓓ). Similarly if the PMO is persisting the shadow copy, those pages can simply be dropped without concern (Ⓟ → Ⓓ)

On the other hand, if the shadow is being encrypted into the primary (as a result of a `psync()`), then the crash handler will have to check whether a particular shadow page has data (and thus was faulted in). If the shadow page is not null, the crash handler will encrypt and copy the page from the shadow into the primary for every page within the PMO (Ⓔ → Ⓒ → Ⓔ), which may mean overwriting primary pages that have already been copied over, and then drop the shadow (Ⓔ → Ⓓ). Since the threat model assumes that any crashes are not caused by hardware failures and does not consider attacks against a PMO in-use, it can be safely assumed that copying and encrypting the shadow pages will not preserve corruption. In either the case of $I_p$ or $I_d$, the crash handler will update the checksums on these pages (i.e., the crash handler effectively performs a `detach()`).

A simple test demonstrates that the crash handler is a feasible way to ensure that data within a PMO are always secure: a 1GB PMO where every shadow page has data and has been decrypted

takes approximately $2.2$ seconds to perform a `detach()` that encrypts each page. Assuming a PMO system that is $120$GB large, this would mean that it would take about $264$ seconds to encrypt each page of the PMO system (a worst case scenario), or about $4$ minutes. This is more than long enough to withstand a power fault when using an uninterruptible power supply (UPS). For example, a high-end UPS can last for more than 6 minutes at a full-load of $2700$ Watts. [65].

Note that, although it is not implemented, it is also possible to further extend the crash handler to handle circumstances such as a kernel panic. For example, one could use a crash kernel that runs in the case where the kernel panics [22]. With this, as long as the system's hardware is functioning correctly, a PMO can be rendered secure.

## Implementation

### *Linux Crypto Subsystem*

Performing encryption on individual pages, rather than on the entire PMO, allows the kernel to take advantage of multiple threads. The kernel encryption subsystem is multithreaded and asynchronous. Specifically, the Linux kernel crypto API's documentation [11] states that with regard to the Symmetric Key Cipher API, "Asynchronous cipher operations imply that the function invocation for a cipher request returns immediately before the completion of the operation. The cipher request is scheduled as a separate kernel thread and therefore load-balanced on the different CPUs via the process scheduler." When an encryption request is submitted to the subsystem, the subsystem assigns it to a kernel thread; that thread is then free to run on any available CPU core. This means that if multiple pages are waiting to be encrypted at `psync()` time and the system is using $SP$, all the pages waiting can be encrypted and copied into the primary copy in parallel. Similarly, `detach()`'s performance with BP can be accelerated.

### *Crash Handler*

To ensure that the stored checksum always matches a valid copy of the PMO data, the crash handler described in the previous section is implemented. The crash handler is invoked whenever a process dies (either normally or abnormally), through `do_exit()` (which the Linux kernel *always* invokes whenever a process dies, regardless of cause).

### *Nonblocking Detach*

In the original implementation described in Chapter 2 [27], detach is blocking; i.e., the kernel does not return control back to the process until `detach()` completes. This design is inefficient, as the user process is stuck waiting on a detach call even if it does not need it. Hence, the design here is nonblocking. When `detach()` occurs, a flag is set and control is immediately returned to the user process. However, any future `attach()` must wait until `detach()` completes.

## Evaluation Methodology

All of the designs described in the design section are evaluated. The baseline design is the Whole Encryption + Integrity Verification ($WEDI + IV_p$) scheme, the same scheme originally described in the Greenspan PMO system [27]. The system used to evaluate these different designs is found in Table 3.3. This enhanced PMO system is a modified version of the Greenspan PMO system described in Chapter 2 [27, 50], which is itself a modified version of the Linux Kernel Version $5.14.18$.[2]

To switch between different PMO designs, the kernel exposes a procfs file, `/proc/pmo` that can

[2]Note that this chapter uses AlmaLinux, while Chapter 2's Table 2.2 used Fedora.

Table 3.3: Configuration of the PM system used for evaluation.

| Component | Specifications |
|---|---|
| Motherboard | Dual socket Supermicro X11DPi-NT (w/ADR) |
| CPU and Clock | 2×Intel Xeon Gold 6230, 20 cores, 40 threads; 2.1GHz (3.9GHz Boost) |
| CPU Cache | L1: 32KiB; L2: 1MiB; L3: 27.5MiB |
| DRAM and PM | $4 \times 32$GiB DDR4 @ 2666MHz; $4 \times 128$GiB Intel Optane DC (PMem) |
| OS and Kernel | 64-bit AlmaLinux 9.0; Linux 5.14.18 |

be read or written to. Writing to the file changes the PMO design scheme, reading from it produces the currently selected scheme. This allows for running all the different PMO designs in a script, without having to reboot to switch between schemes.

*Evaluated Benchmarks*

The following microbenchmarks are used for this evaluation: 2d Convolution (2dConv), Gaussian Elimination (Gauss), LU decomposition (LU) and Tiled Matrix Multiplication (TMM), all from [27] and originally obtained from [20]. These microbenchmarks were ported to use PMOs by designing and implementing a user-space free-list allocator and API. This allocator implements substitutes for standard dynamic memory functions, such as `malloc()` and `free()`, with its own versions, `p_malloc()` and `p_free()`, respectively.

The Filebench benchmarks [66] (FileServer, VarMail, WebProxy, and WebServer) are also used, adopted from [27] with no modifications.

*Microbenchmarks*

Each of the microbenchmarks invoke `psync()` after a specified number of iterations of the performance-critical loop, set to occur approximately every second when using the original, no encryption, case. This tempo (once per second) was chosen because Append on File Redis (AOF Redis)'s suggested default policy is to invoke fsync once per second [61], which is also used in other works utilizing Redis such as NVMove [8]. Additional works such as [23, 54] persist once per second and hence it is a common standard. The benchmarks use iterations instead of timers for consistency: using timers means that the amount of work between synchronization calls varies based on the PMO design, which negatively impacts the quality of the results.

Each of the microbenchmarks detach after a specified number of `psync()`s, from $1 - 16$. For the overall evaluation of each PMO design, it makes sense to start with 2 `psync()`s per detach. This is done because it is a reasonable anticipation that, in the real-world, most attach/detach session will not fault every page within a PMO, but only a subset of them. Table 3 shows the configurations of each benchmark.

Table 3.4: Microbenchmark Configuration

| **Benchmark** | **Configuration** | **PMO Size** |
|---|---|---|
| **2dConv** | N=6144, M=104 | 32MiB |
| **Gauss** | N=8192 | 256 MiB |
| **LU** | N=7168 | 756 MiB |
| **TMM** | N=4096, Tile Size=16 | 196 MiB |

*Filebench benchmark configuration*

Filebench [66] represents I/O intensive real-world applications and is designed for measuring I/O bandwidth performance. These benchmarks were ported to use PMOs by [27], and are adopted

here. It is important to note that synchronization points (`psync()` invocations) are invoked at every update and a pthread barrier is emitted before and after each `psync()` with the goal of avoiding data races. Each workload was run $5$ times for $30$ seconds, and the result is the average between the runs. Each workload has a different percentage of read/write operations. FileServer (FS) uses $67\%$ writes, VarMail (VM) has $50\%$, WebProxy (WP) is $16\%$, and finally, WebServer (WS) is $9\%$.

## Evaluation

This section attempts to answer several questions: How much more performant are the per-page designs compared to the baseline Whole Encryption design? Is BP or SP more performant, and in which workloads? How scalable are BP and SP as the attach/detach size increases, and how is thread scalability impacted? How much performance is gained with the designs supporting integrity verification compared to the original WED$I_p$ design? What is the performance impact of $I_p$ compared to $I_d$?

### *Microbenchmark performance evaluations of per-page designs*

Figure 3.6 compares the execution times of different PMO designs. Results are normalized to 8 threads. The results are split into different categories: Psync Per+IV, Psync Encrypt/Copy, Detach, PF Overhead, Attach Other, Attach Stall, and Compute.

Figure 3.6 compares the execution times of different PMO designs. Results are normalized to 8 threads. The results are split into different categories: **Psync Per+IV**, **Psync Enc/Cpy**, **Detach** , **PF Overhead**, **Attach Other**, **Attach Stall**, and **Compute**.

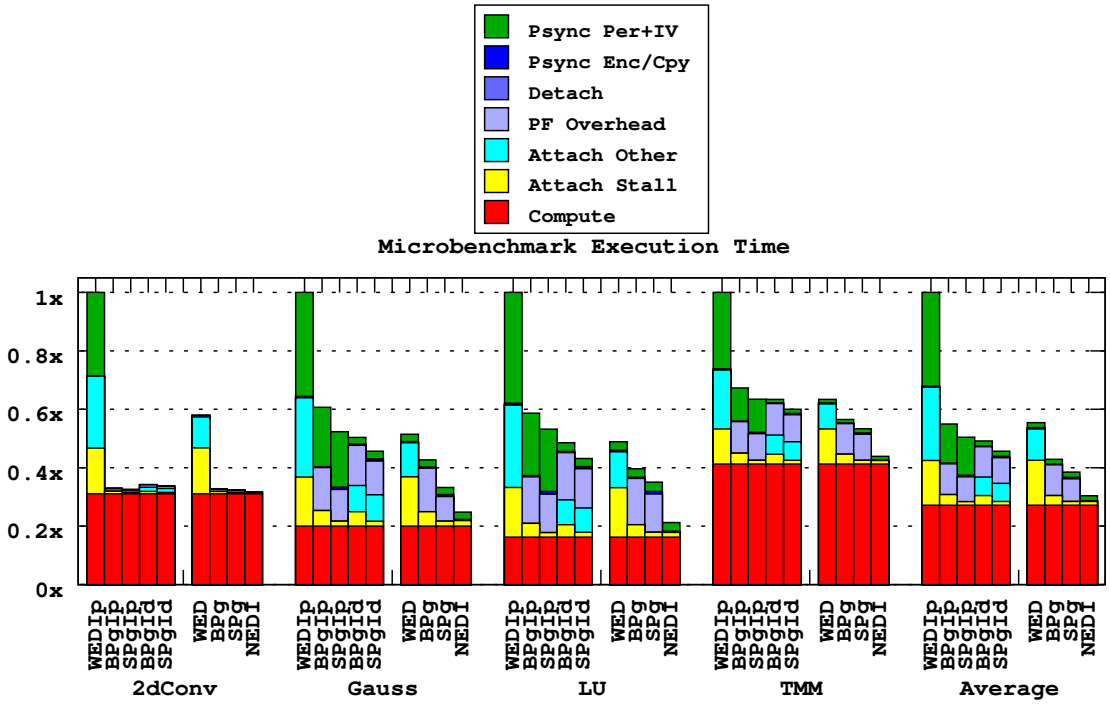



Figure 3.6: Execution time by design, with attach session size of 2, normalized to $WEDI_p$

**Psync Per+IV** consists of the components of `psync()` that are not rendering the primary PMO crash consistent i.e., they include invalidating the cache lines to render the shadow copy valid and updating checksums when using the IVp designs. **Psync Enc/Cpy** includes updating the primary copy via encryption (Enc) or memcpy (Cpy) calls. **Detach** is the time it takes between when the detach system call is invoked and when it returns control back to the user process; since it is non-blocking, the time spent doing this is very small. **PF Overhead** is the additional time over the no-encryption case spent on servicing page faults (i.e., the time spent decrypting the page and rendering it available to the calling process). **Attach Other** is the time spent performing all attach operations that are independent of detach (i.e., if using the Whole design, the time spent decrypting the PMO), while **Attach Stall** is the time spent waiting for the detach thread to complete (since a detach call is immediately followed by an attach call, this is a proxy for the time it takes for the detach thread to complete). Finally, **Compute** is all other time: the time spent not waiting

on the PMO system (this includes the base latency for servicing page faults, and the time spent performing computation).

The performance of the individual categories matches expected performance; e.g., it makes sense that `psync()` calls will be more expensive with SP rather than BP (since SP requires encrypting the page into the primary rather than simply copying). Similarly, attach stall times should be more expensive since detaching a PMO using the BP method takes additional time (the primary must be encrypted into the shadow and then copied back, to ensure no loss of data).

With these four designs, the best performing IV design ($SP/I_d$) is, on average, $2.1\times$ faster than the WED$I_p$ design. The second best design ($BP/I_d$) is $2.03\times$ faster; a difference of $3\%$. For the non-IV designs, the performance of $SP$ is $1.4\times$ and $BP$ is $1.3\times$ faster than WED.

*Sensitivity Study*

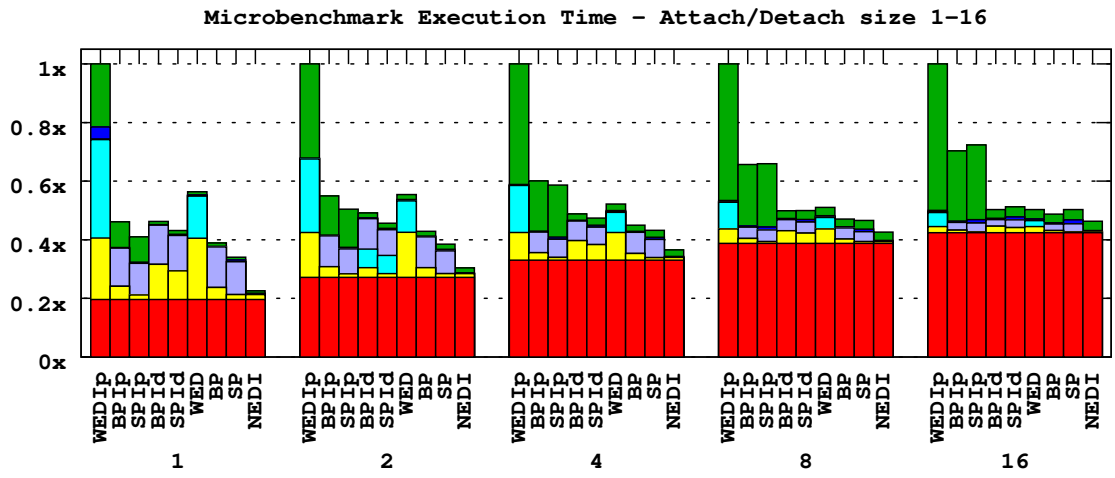



Figure 3.7: Average execution time for all the microbenchmarks with different attach sizes, normalized to WED$I_p$

Figure 3.7 shows `attach()` session sensitivity by growing the attach/detach size. The attach

session size is the number of `psync()`s before a `detach()`. For example, if the system is performing `psync()` once per second, and after each `psync()` it performs a `detach()`, then the attach session size is 1. On the other hand, if the system only performs a `detach()` after 16 `psync()`s, then the attach session size is 16.

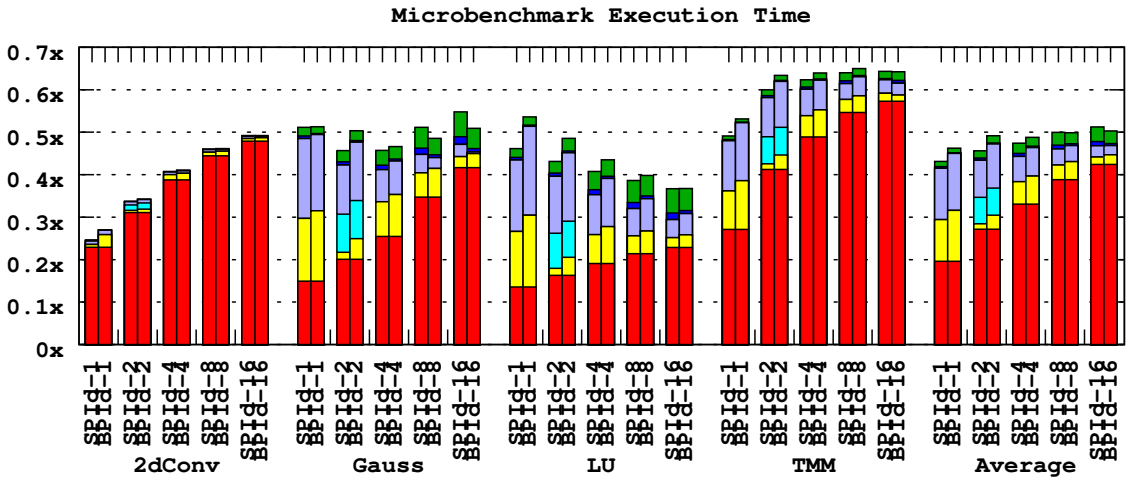



Figure 3.8: Comparison of $BP/I_d$ and $SP/I_d$ with different attach session sizes, normalized to WED$I_p$.

Figure 3.8 focuses on the two most performant designs ($BP/I_d$ and $SP/I_d$) as the attach session size increases. The results are normalized to $SP/I_d$ at 1. At a size of 16, $BP/I_d$ is slightly faster, $2.75\%$. What this demonstrates is that the $BP/I_d$ designs are superior to $SP/I_d$ when performing attach/detach infrequently, which is the expected behavior since `psync()` is more expensive with $SP/I_d$ (encrypting the shadow into the primary vs. simply copying the shadow into the primary). While page faults are more expensive with $BP/I_d$, a large attach/detach size means that there are fewer pages being faulted in for the first time out of a total number of pages accessed. This is illustrated by Figure 3.9. Figure 3.9 shows the number of pages touched between an attach/detach session for each benchmark by attach/detach session size (lightly colored bars, $a/d$) and shows that as the number of pages touched gets larger, the number of page faults declines relative to the number of `psync()`s (darkly colored bars, $p$). All of these facts combined means that $BP$ tends

to have better performance in these cases.

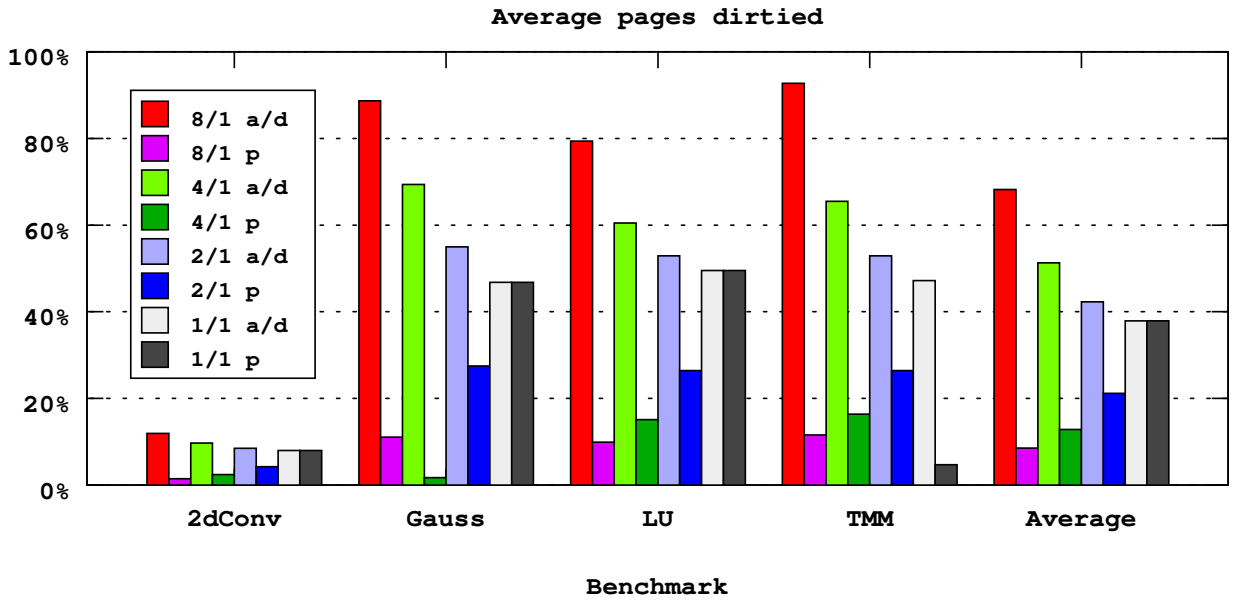



Figure 3.9: Pages accessed between attach/detach calls and psyncs.

It is important to note that in an actual real-world application, the cost of psync may well prove to be much more important than the cost of detach, since detach is done off the critical path. In an application that attaches a PMO, performs multiple operations on it, detaches it, and does not use it for a long-time, $BP$ is likely to be the more performant choice. Yet, as described in the background section, prior work by Xu, et. al. [75] proposes reducing the amount of time a PMO is attached to a bare minimum; if this suggestion is followed, then $SP$ is the better option.

### *Filebench*

Figure 3.10 compares the I/O bandwidth of different Filebench workloads achieved by different PMO designs. Results are normalized to $WEDI_p$ and reported for 8 threads with synchronization performed after every write or append operation. On average, $SP/I_d$ is $2.56\times$ faster than WED$I_p$,

while SP alone is $3.2\times$ faster.

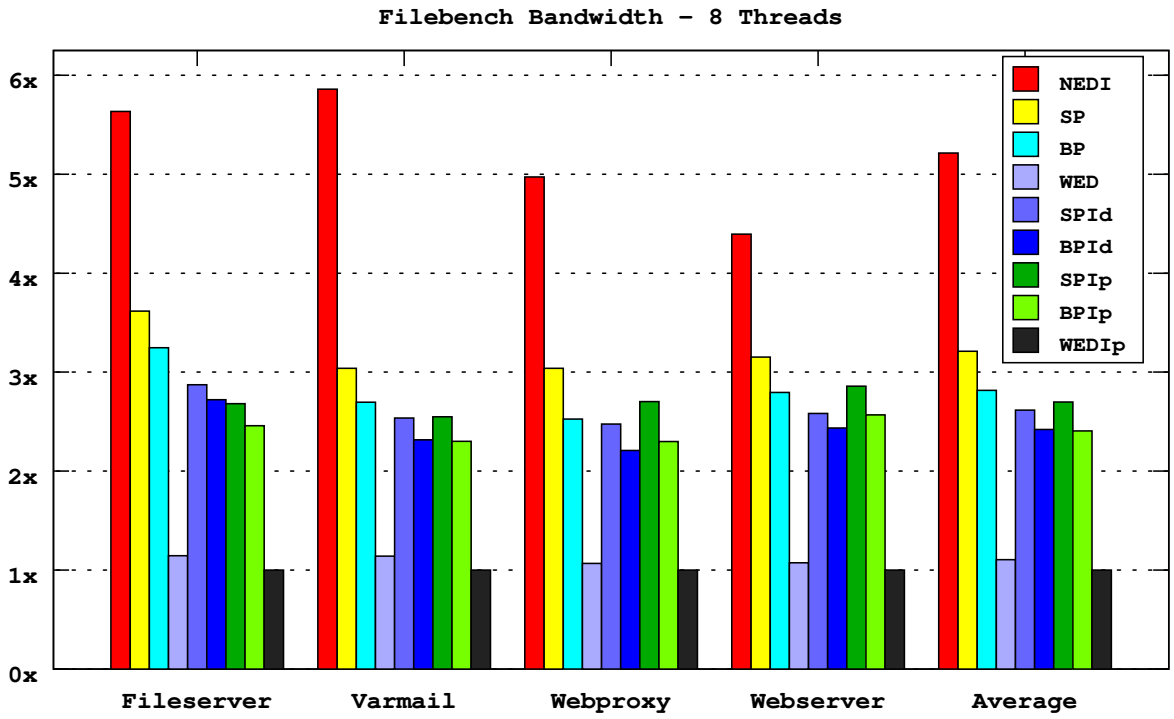



Figure 3.10: Filebench results, normalized to NED$I_p$.

Interestingly, with WebServer, $I_p$ is slightly more performant than $I_d$ because WebServer calls detach much more often than psync ($9\%$ writes, $91\%$ reads). This allows us to conclude that in scenarios where `psync()` is infrequent or does little, $I_d$ is worse than $I_p$, which is preferable in this scenario.

*Thread scalability of LOaPP*

Figure 3.11 shows the thread-scalability of the PMO system's performance when using the best-case design ($SP/I_d$). Results, normalized to a single thread, are shown for all of the microbenchmarks when executed with $N = (1, 2, 4, 8)$ threads and synchronized once per second, with an

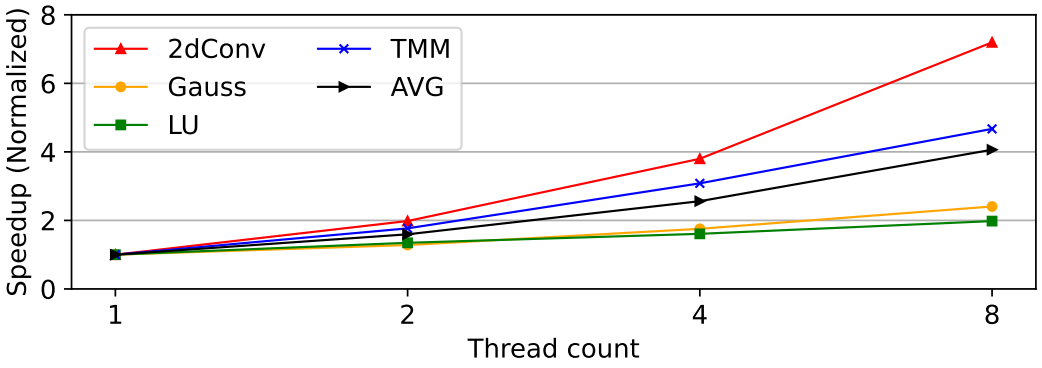



Figure 3.11: Microbenchmark scalability

attach/detach size of 2. Results show that performance scales for an increasing number of threads, but Gauss and LU at 8 threads are only about $2\times$ faster than at 1 thread. This behavior is expected and unrelated to PMOs; as it is an effect of the physical property of the PM fabric (Optane), as originally discovered by [34, 78]. Excessive numbers of writer threads slow the PM fabric down. To verify that this is not related to LOaPP, the microbenchmarks with a filesystem (ext4-dax) on top of the PM are also tested; with that, the thread scalability results remain the same. However, this behavior does *not* occur when testing the microbenchmarks with volatile memory.

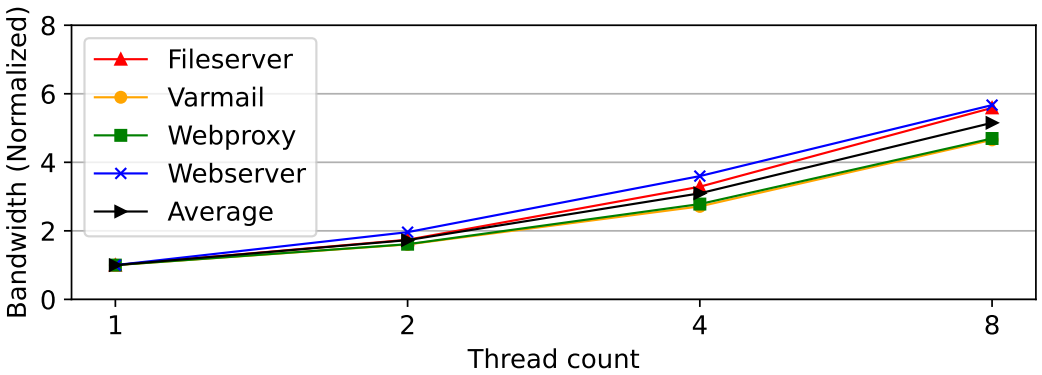



Figure 3.12: Filebench scalability

Figure 3.12 shows the thread-scalability of Filebench with $SP/I_d$; these results are as expected;

each thread is independent of each other, since Filebench is really testing sustained I/O bandwidth.

### *Correctness and Crash Consistency*

To verify that the per-page encryption design is crash consistent and secure even in the face of crashes, two different tests are performed. First, `do_exit()` is inserted after a random number of PMO page faults are handled. This causes the associated user space process to terminate abnormally, and invokes the PMO crash handler. Examining the contents of the PMO after the crash handler has completed reveals that none of the faulted pages are visible, and that the shadow copy of the PMO is empty. Performing this same test within `psync()` after the persist stage but before the copy stage similarly reveals that the data within the shadow are gone, and the data from before the `psync()` are still in the primary copy, as expected.

## Conclusion

Security and performance are both important design considerations in persistent memory abstractions. To improve performance, this chapter introduced per-page encryption to PMOs, and discussed the design and implementation of a PMO system with high performance, and high reliability (crash consistency) without reduction in security. Results show that, compared to the prior PMO abstraction, per-page encryption with integrity verification yields performance $2.19\times$ and $2.62\times$ faster without sacrificing security or reliability. The next chapter investigates predictive decryption on attach; where pages that have been used often in the past can be decrypted ahead of time off the critical path, before a page fault uses them.

Acknowledgments

The material in this chapter is supported in part by the Office of Naval Research (ONR) under grant N00014-20-1-2750, and by the National Science Foundation (NSF) under grant 1900724.

# CHAPTER 4: PD-LOAPP: IMPROVING THE PERFORMANCE OF PMOs VIA PRE-DECRYPTION AND DRAM-AS-CACHE

This chapter expands on the work of Chapters 3 and 2 by further improving the performance of PMOs through the use of DRAM as a cache for PM and the use of pre-decryption to predict when pages might be used and decrypt them ahead of time. This chapter observes that the performance of secure PMOs, even when using decryption with demand paging, is degraded by the fact that the underlying Persistent Memory device (in this case, Optane) has much slower reads and writes compared to DRAM, and that it is well known that PM has a limited durability compared to DRAM. To rectify this problem, the assumption of previous chapters that the PM system has no or negligible amounts of DRAM is removed, since current PM systems have a large amount DRAM; this chapter observes that removing this assumption does not change the PMO abstraction, while also improving performance dramatically. However, decrypting pages within a PMO on demand into DRAM is inefficient, so this chapter also proposes decrypting pages ahead of time so that when page faults occur, they are already decrypted. Using DRAM as a cache alone outperforms a pure PM design by up to $1.7\times$. When bundled with a simple stream predictor, the improvement reaches $1.9\times$, depending on the workload.

## Introduction

Persistent Memory (PM) allows systems to access persistent storage in a byte-addressable manner. Although the first commercial PM (Intel Optane) has been discontinued, CXL Persistent Memory [7] with memory-semantic SSDs is a promising alternative [14, 3]. One method of managing PM-resident data is to consider PM as a repository of Persistent Memory Objects (PMOs) [27]. A PMO holds persistent data in pointer-rich data structures without any file backing and the kernel manages

the underlying semantics of the PMO. When a PMO is mapped to userspace via `attach()`, it is accessed in a byte-addressable manner. When a PMO is unmapped via `detach()`, the PMO then becomes available for other processes to access. PMO systems also provide `psync()` to manage crash consistency; when data are written into a PMO, the data are not rendered durable until `pysnc()` is invoked.

The previous chapters of this dissertation have covered the design and implementation of PMOs (see Chapter 2) as well as solving the problem of crash consistency. The previous chapters also improved security and performance by splitting PMOs into individual pages and performing encryption, decryption, and integrity verification on them instead of the entire PMO (See Chapter 3) [26].

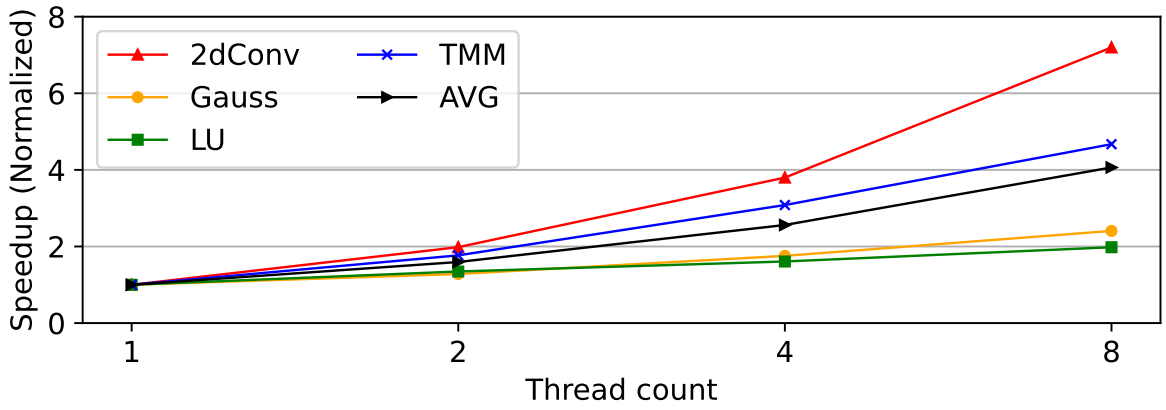



Figure 4.1: Microbenchmark scalability by thread count

Figure 4.1 reflects the performance of multiple benchmarks from 1 to 8 threads. As can be seen, when using PMOs, the performance gains of additional threads drops as the number of threads increases.

From other analysis of Optane memory's performance characteristics [79, 78], it is evident that this problem is not specific to the PMO system design, but instead is a problem with the underlying PM fabric. In addition, writes to PM are known to wear down the medium; hence PM has

both limited write bandwidth and limited write endurance [48]. To fix these issues, this chapter proposes extending the LOaPP design from Chapter 2 to map userspace pages to physical DRAM page frames rather than map those pages to the shadow copy; this new design is called *D-LOaPP* (DRAM Low Overhead at-rest PMO Protection).

Another problem with LOaPP is that it decrypts pages within a PMO *on-demand* rather than beforehand. The Threat Model of Chapter 3 only requires that a PMO at-rest be secure, it does not require that only those pages that are strictly needed to be decrypted are decrypted. Hence, it is desirable to decrypt pages ahead of time if possible, since from a security perspective, there is little consequence to doing so before they are needed. To that end, this chapter proposes using a *stream-buffer* like design to predict when pages might be needed ahead of time and to perform their prediction operations. Since modern operating systems move decryption off of the critical path, this has the potential to further improve performance. Furthermore, the write bandwidth of DRAM is much larger than the write bandwidth of PM, so writing decrypted pages into DRAM should not impact the performance of other writes to DRAM. This further enhanced design is referred to as *PD-LOaPP* (Predictive DRAM Low Overhead at-rest PMO Protection).

Finally, Chapter 3 demonstrated that while Integrity Verification does not have tremendous overheads when done at detach, there is a performance cost to it nonetheless. Ideally, this performance cost can be reduced or even completely eliminated with PD-LOaPP.

To design and implement PD-LOaPP, this chapter must contend with several challenges. With regards to using DRAM for userspace writes, the primary question is how to ensure that PMOs are still crash consistent. A secondary concern is *how often* to call `psync`, since each call to `psync()` will invariably be slower since an additional write (from DRAM to the shadow copy) must occur. With regards to page prediction, the primary concern is to ensure that the design has high accuracy, high coverage, and high timeliness. In addition, the PMO system can adopt a different design from

classic Stream Buffers since the buffer size can theoretically be infinite. All of these challenges are addressed in this chapter.

In summary, this chapter makes the following **contributions**:

1. This chapter proposes a novel scheme, *Predictive-DRAM Low-Overhead at-rest PMO Protection (PD-LOaPP)* to use DRAM as cache for PMOs and redirect writes to there, and to predict ahead of time which pages should be decrypted to reduce the page fault handling time.

2. This chapter presents an exploration of PD-LOaPP's design space, with multiple optimizations to further reduce protection overhead without compromising crash consistency.

3. This chapter implements PD-LOaPP on a Linux kernel, on a real system equipped with Intel Optane PMem. Extensively evaluating PD-LOaPP on several workloads reveals that using DRAM as a cache alone outperforms a pure PM design by up to $1.7\times$. When performing prediction with a design similar to a traditional stream buffer, the improvement is up to $1.9\times$.

## Background

This section discusses the concepts and prior works required to understand Predictive Decryption and PMO with DRAM as cache. This section provides a brief review of the PMO subsystem (see Chapter 2 for more details) and per-page encryption (see Chapter 3 for more details). In addition, this section discusses prior work in predictive decryption.

### *Persistent Memory*

Persistent Memory (PM) is byte-addressable memory that is persistent across power-cycles, and denser than volatile memory, with far lower latencies compared to other long-term storage devices such as solid state drives (SSDs) and hard disks (HDDs). Two major approaches exist for managing PM: either to treat PM as a host for a traditional filesystem [10, 37, 70, 72, 71], or to view it as a collection of Persistent Memory Objects (PMOs) [27, 63, 73, 76].

Hosting a filesystem on a PM require storing persistent data in files and then using `mmap()` with direct memory access (DAX) to map the data into the address space of another process. This approach is lacking because of two reasons. 1) The approach requires that memory and file metadata be consistent despite their different semantics. 2) The approach creates huge overheads compared to the raw persistent memory device write bandwidth [37], which is a serious problem since PM already has extremely limited write bandwidths compared to traditional DRAM.

### *Persistent Memory Object subsystem*

The PMO design, in contrast to a filesystem, looks at PM as a collection of byte-addressable *objects* rather than block-addressable files. Persistent data within a PMO is stored in data structures that have the potential to be pointer-rich, and the metadata required for a filesystem are mostly eliminated and handled by the kernel instead. PD-LOaPP uses the Greenspan PMO (GPMO) system [27], as it is the only available PMO system that works on real hardware and has been implemented in a real operating system.

A PMO can be created with `pcreate()`, attached into the address space of the calling process with `attach()`, rendered durable with `psync()`, severed from the calling process with `detach()`, and destroyed with `pdestroy()`.

*Crash Consistency*

A technical failure can cause data stored within a PM system to become corrupt due to partial or improperly ordered writes. To prevent this, PM systems utilize *crash consistency*. Logging and shadowing are two methods for achieving crash-consistency; logging tends to be used with filesystem approaches [71], while the GPMO system uses shadowing by maintaining a primary and shadow copy of each modified page within a PMO. The system ensures that at least one of the primary or the shadow copy of the PMO remains consistent, with the valid copy being used to restore the PMO in the event of a crash rendering the data otherwise inconsistent.

*Multithreading performance*

As discussed in the introduction, PM's physical properties strongly impact the performance of multithreading. First, Intel Optane's physical properties mean that only a limited number of writes can occur at any one time [34]. Second, the upcoming replacement to Optane, CXL, will have similar properties: it is known that access to CXL memory is roughly equivalent to the access latency for the far node on a NUMA node [7]. Since CXL memories will use similar material to Optane (or use Memory-Semantic SSDs [36]) this issue is not easily resolvable through new hardware alone.

*Per-page encryption*

Chapter 3 introduced Low-Overhead at-rest PMO Protection (LOaPP) through breaking PMOs into separate pages, with the goal of improving the performance of PMOs without affecting security. The chapter observes that the performance of PMOs using integrity verification and encryption is lacking, especially with larger PMOs, because the entire PMO is encrypted/decrypted and its

integrity is validated at every attach/detach call. To fix this, Chapter 3 proposed breaking a PMO into individual pages and adopting demand-decryption, where on page fault, each page is not only remapped into userspace, but also decrypted. The chapter refers to this as Low Overhead at-rest PMO Protection (LOaPP). It discusses the trade-offs of four different designs: 1) Both Primary and Shadow pages decrypted and checksums updated at psync ($BP/I_p$). 2) Only Shadow pages decrypted and checksums updated at psync ($SP/I_p$). 3) Both Primary and Shadow pages decrypted and checksums updated at detach ($BP/I_d$). 4) Only the Shadow pages decrypted and checksums updated at detach ($SP/I_d$). That chapter found that $SP/I_d$ had the best performance with integrity verification (on average, $2.10\times$ faster) compared to the original Whole Encryption/Integrity Verification design ($WED/I_p$), while $SP$ alone was $1.4\times$ faster than the original Whole Encryption only ($WED$) design. Hence, this chapter will only focus on the $SP/I_d$ and $SP$ designs.

## *Prior Work*

### *DRAM as last-level cache*

The idea of pairing volatile memory to act as a buffer for persistent memory is not new. For example, Intel Optane's Memory Mode treats DRAM as a cache, effectively allowing large memory pools of PM as volatile memory, with DRAM as an L4 Cache [53] and the PM as a huge volatile memory pool.

### *Cache and Page Prediction*

Most prior work involving data prefetching are at hardware cache level rather than software page level, and involve either stream buffers or correlation tables. Stream buffers work by copying contiguous blocks of memory into the local cache after a previous cache miss, which provides

spatial locality. In contrast, correlation tables attempt to correlate cache misses with accesses to predict ahead of time when a page might be needed, which provides temporal locality [29]. Most prefetching designs are hardware-based, such as with a prefetching engine [59] or a correlation table that intercepts L2 Cache calls [64]. The idea is to predict ahead of time if data are needed and put the predicted data into a separate buffer that is adjacent to the cache. Both stream buffers and correlation tables alike suffer from a limited cache size making prediction accuracy important: mispredicting multiple addresses not only creates unnecessary work, but the limited size also limits how many predictions can exist within the buffer at any one time, meaning inaccurate predictions are worse than useless. This chapter takes inspiration from stream buffers for prediction, but at the software level rather than hardware, and at the level of pages rather than cache lines.

## Design

### *DRAM as PM cache*

Consider a PM system using $SP/I_d$. By itself, this system has reads and writes occurring directly on the PM. However, as demonstrated in the introduction, performance suffers as the number of threads increases, due to contention and the physical properties of the PM system. In addition, recall from the previous section that PM has a limited write endurance. Both of these problems limit the ability of the programmer to use PMOs.

A solution to this problem is to map pages residing in DRAM into userspace and allow the program to read and write to it instead of directly on the PM substrate. While the previous chapters indirectly assumed that a PM system would not have significant amounts of volatile memory, current systems with PM have a mixture of both PM and DRAM, so DRAM can be used as a cache for PM.

Using DRAM as cache introduces two complexities. First, `psync` becomes slightly more com-

plicated since the volatile copy is in DRAM, not in the shadow. Hence, psync now takes slightly longer, since the persist step is now a copy (from DRAM to the shadow) followed by another copy (from shadow to primary).

Second, PM systems with DRAM will likely have far more PM than DRAM; for example, the PM system used throughout this dissertation has $124$ GiB DRAM, but has $504$ GiB PM, a ratio of $1:4$. Hence, if a system is actively using large amounts of PM, the system must either reclaim unused DRAM pages, or when the DRAM is exhausted, future page faults must map directly into the shadow PMO rather than volatile memory. This chapter does not discuss this in depth, but the reclamation process and DRAM exhaustion handler is a topic for future work.

Figure 4.2, adapted from Chapter 3, illustrates the modified state transitions required to support predictive decryption and DRAM as cache. Although (a) is unchanged from prior chapters, (b), (c), and (d) are different. The Page Fault state diagram (b) shows that the PMO system now decrypts the primary page into DRAM and verifies the checksum of the page (if enabled). Once the page is decrypted, the kernel initiates the nonblocking prediction handler and then immediately returns.

Psync (c) no longer focuses on the BP design, but *only* dirty pages (rather than all extant working pages like before) are encrypted from DRAM into the shadow (EP). This is in contrast to the prior design where the shadow was simply persisted. The rest of the process is otherwise identical to last chapter's deisgn, but psync is now slightly slower since two copies are performed, rather than a persist followed by a copy. Finally, Detach (d) updates the checksum (UC) and then frees and clears *all* DRAM pages (DS), including pages that were predicted but never used. Only then are the shadow pages destroyed, and the detach handler returns.

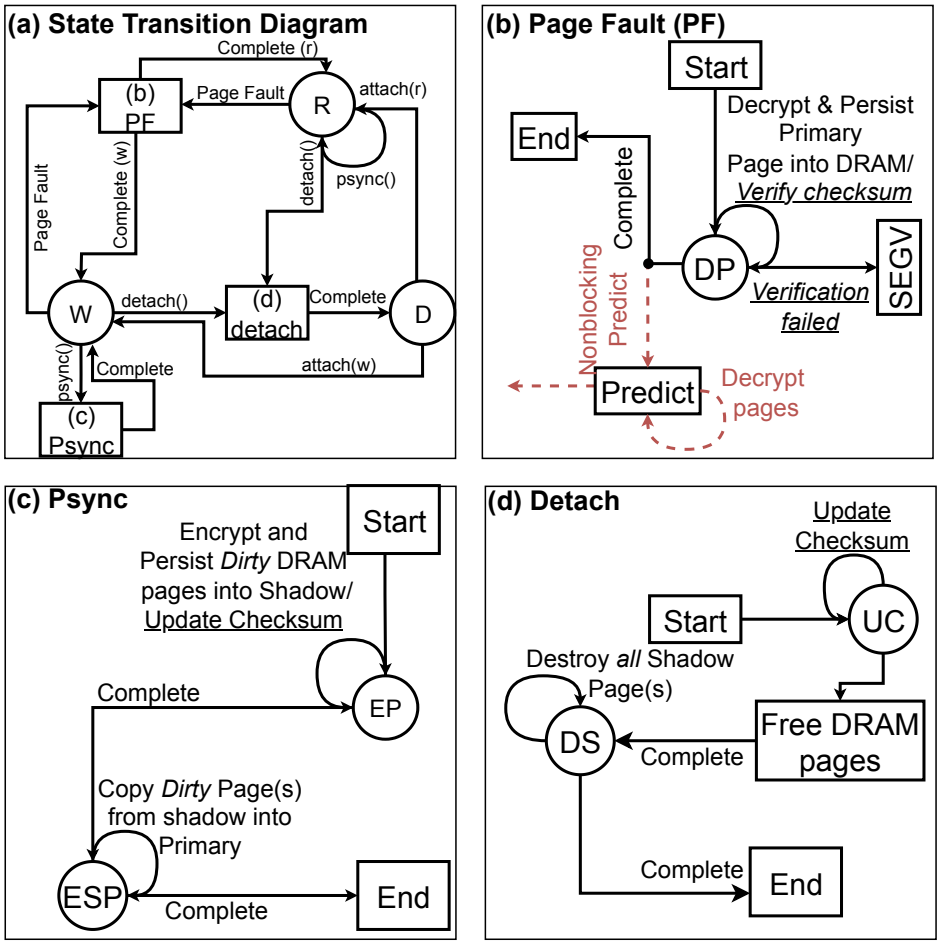



Figure 4.2: The updated state diagram with DRAM and prediction.

*Per-page Prediction*

As mentioned earlier, the problem of determining which data will be accessed and copying them into a fast cache ahead of time is well studied. To improve the performance of a DRAM cache even further, this section proposes decrypting pages into DRAM ahead of time via the use of a stream buffer-like design

Since DRAM is much larger than the (generally) limited space of a stream buffer, the penalty for

wrongfully predicting a page is significantly lessened. Rather than have a separate buffer, data can be decrypted directly into DRAM and used at page fault time.

There are three questions impacting the efficiency of the predictor. First, does a stream buffer improve or worsen performance? If the standard assumption of spatial data locality (i.e., that if a page is accessed, its neighbor will likely be accessed too) is incorrect, then the stream buffer has the potential to worsen performance because the system will be stuck servicing pages that are not needed; this slows down detach because it now will have to clear unaccessed pages, it also slows down the general operation of the system because it consumes the write bandwidth of DRAM and the read bandwidth of PM. In addition, even if the decryption occurs off the critical path, the process will still occupy CPU utilization that could be performing useful work. How many predictions were later found to be accurate are referred to as *accuracy*.

Second, how many accessed pages will be decrypted beforehand? A page that is not decrypted beforehand will experience all of the latency of the page fault from the original Chapter 3 when the fault occurs. The kernel can remap pages from kernel to userspace quickly, but it cannot decrypt them quickly. This is referred to as *coverage*.

Finally, how often is the page fault handler stuck waiting for decryption to complete? There may be pages that have been successfully predicted ahead of time, but the page fault handler nonetheless cannot map the page to userspace since the decryption is ongoing. This is *timeliness*.

All three questions have high impact on the overall effectiveness of the predictor. To be useful, all three metrics should be very high, although the penalty is significantly lessened compared to mispredictions occurring with a cache predictor. The results of each of these metrics are reported in Figure 4.5, although the question of whether other, more complicated, predictors such as Markov predictors [35, 45] can improve performance is left for future work.

## Implementation

Analogous to the Virtual Memory Areas (VMAs) that the Linux Kernel uses to keep track of memory mappings [47] (See also Chapters 2 and 3 for further elaboration), the PMO subsystem maintains VPMAs (*Virtual Persistent Memory Areas*). To keep track of pages from DRAM, the VPMA is extended by adding two additional structs, one for tracking pages in DRAM, and another for handling prediction.

### *DRAM Page Tracking*

To handle decryption into DRAM, each page within a PMO has an associated structure containing the scatterlists for the primary and DRAM pages as well as additional flags indicating whether the page has been predicted, faulted, or handled by the kernel. When a PMO is attached for the first time, the associated scatterlists are initialized and mapped to their respective pages. When a page fault occurs, the kernel checks whether the page has been handled in a previous prediction, and if so, maps the DRAM page into the appropriate userspace virtual addresses, otherwise it performs the page fault handler routine and then the mapping. At psync, the PMO renders the synchronized pages durable, as described in the previous section. When a PMO is detached, the data in DRAM are discarded; when using DRAM as a buffer, the data in shadow are only discarded at the processes' termination, since the data in shadow are encrypted. The first time the page is either predicted or faulted, the kernel serves a page from DRAM via `__get_free_page()`.

### *Prediction*

To handle prediction, the kernel calls the prediction handler at page fault time. The prediction handler then decrypts the next $X$ sequential pages (where $X$ is the stream depth). If the atomic

flag indicating that the page has already been handled is set, the prediction handler skips that page, otherwise it performs the decryption. A key challenge with prediction is ensuring that the prediction and page fault code do not occur at the same time. To do this, the kernel locks the page via a mutex, future page faults on the page then wait until the decryption completes, and then skips the decryption step (calling `remap_pfn_range()`).

## Evaluation Methodology

This section discusses how the PMO system with prediction was evaluated, as well as the testing platform used. The original PM design from Chapter 3 is compared with the base DRAM design, as well as with the stream predictor. The system used to evaluate these designs is found in Table 4.1, which is a modified version of the Linux Kernel Version 5.14.18.

Table 4.1: Configuration of the PM system used for evaluation. Note that the configuration is identical to that used in Chapter 3's Table 3.3.

| **Component** | **Specifications** |
|---|---|
| Motherboard | Dual socket Supermicro X11DPi-NT (w/ADR) |
| CPU and Clock | 2×Intel Xeon Gold 6230, 20 cores, 40 threads; 2.1GHz (3.9GHz Boost) |
| CPU Cache | L1: 32KiB; L2: 1MiB; L3: 27.5MiB |
| DRAM and PM | $4 \times 32$GiB DDR4 @ 2666MHz; $4 \times 128$GiB Intel Optane DC (PMem) |
| OS and Kernel | 64-bit AlmaLinux 9.0; Linux 5.14.18 |

As in Chapter 3, this section utilizes the procfs subsystem and exposes multiple procfs files. In addition to `/proc/pmo/pmo` that allows for changing different PMO encryption and checksum designs, the directory now exposes `dram`, `pred`, and `depth`. The different configuration options exposed by these files are shown in Table 4.2.

Also like in Chapter 3, this section evaluates four microbenchmarks and four additional Filebench benchmarks.

Table 4.2: Procfs files and description

| File | Description |
|---|---|
| /proc/pmo/pmo | Switches between different PMO primary/shadow designs (from Chapter 3) |
| /proc/pmo/dram | Enables/disables the DRAM caching system |
| /proc/pmo/pred | Enables/disables Stream predictor |
| /proc/pmo/depth | Allows for specifying stream predictor depth |

*Evaluated Benchmarks*

The following microbenchmarks are used for this evaluation: 2d Convolution (2dConv), Gaussian Elimination (Gauss), LU Decomposition (LU) and Tiled Matrix Multiplication (TMM). Figure 4.6 depicts the different access patterns of each benchmark, with the dotted line indicating different attach/detach boundaries. The configurations of the microbenchmarks are found in Table 4.3.

Table 4.3: Microbenchmark Configurations

| Benchmark | Configuration |
|---|---|
| 2dConv | $256 \times 1024$ |
| LU Decomp | 6144 |
| Gauss | 18432 |
| TMM | 4096 16 |

This chapter also adopts the Filebench benchmarks from previous chapters. Each workload has different percentages of read/write operations and is run $5$ times for $1$ minute, and the result is the average between the runs . FileServer (FS) has $67\%$ writes, VarrMail (VM) has $50\%$, WebProxy (WP) has $16\%$, and WebServer (WS) is only $9\%$.

This section attempts to address multiple questions. How much more performant DRAM is compared to the prior PM only design? What are the access patterns for the different microbenchmarks? How timely are the predictions with the stream buffer design, and what is the prediction accuracy and coverage? How does timeliness, prediction, accuracy and coverage affect the performance of the predictor? The legend for the PMO execution time breakdowns for this section can be found in Figure 4.3.

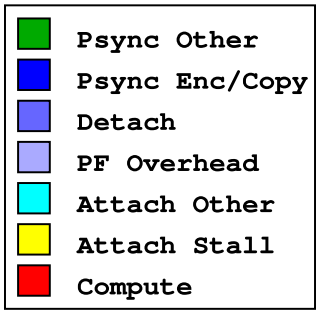



Figure 4.3: Legend for PMO execution time breakdowns.

*Microbenchmark performance evaluation*

Figure 4.4 compares the execution times of the original, fully PM $SP_g$ design from Chapter 3 with the total execution times of the $DSP_g$ design (DRAM). Like before, the results are normalized to 8 threads and the total execution is split into different categories: **Psync Other**, **Psync Enc/Cpy**, **Detach** , **PF Overhead**, **Attach Other**, **Attach Stall**, and **Compute**.

**Psync Other** consists of the components of `psync()` that are not rendering the primary PMO crash consistent; i.e., they include invalidating the cache lines and updating checksums, but they do not include copying the shadow (or DRAM) copy into the primary copy of the PMO; those are

included in **Psync Enc/Cpy**. **Detach** is the time between when `detach()` is invoked and when it returns (since the system call is nonblocking, the total time spent performing it is negligible). **PF Overhead** is the overhead cost (normalized to the original $SP_g$ design) for performing page faults. **Attach Other** is the time spent performing the operations required for `attach()` to complete, excluding **Attach Stall**, which is the time spent waiting for detach to complete before the attach process can begin. Finally, **Compute** is all other time: generally, it is the time performing "useful" operations and should be the same regardless of the type of system being evaluated.

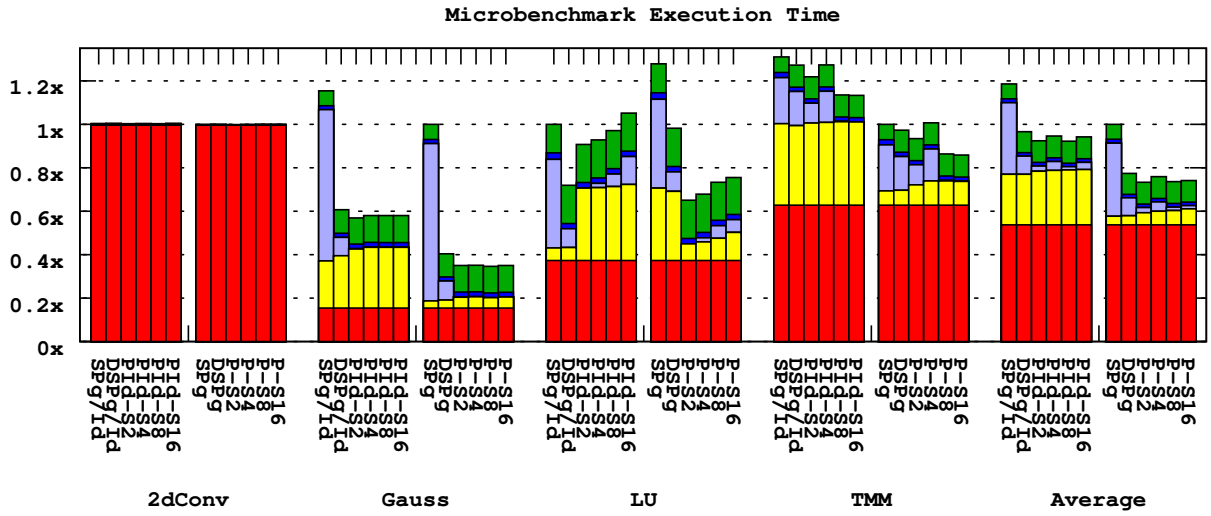



Figure 4.4: Execution time with and without DRAM prediction at 8 threads.

When only considering $SP_g$ vs. $DSP_g$, there are no real surprises in Figure 4.4. 2d Convolution has consistently been compute heavy throughout all the previous chapters, while Gaussian Elimination has always been a write heavy benchmark (and hence benefits from DRAM's higher bandwidth and faster writes).

When looking at the impact of $PDSP_g$, Integrity Verification with prediction is on average actually *faster* than PM without Integrity Verification! With the best depth, (8) it is $7.8\%$ faster. This implies that the performance penalty of Integrity Verification that was elucidated in the previous chapter

can be nullified. It is likely that if TMM had better accuracy (see Figure 4.5), its performance with prediction would be higher. In general, the performance improvement on average is $1.7\times$ faster than the previous best-case design.

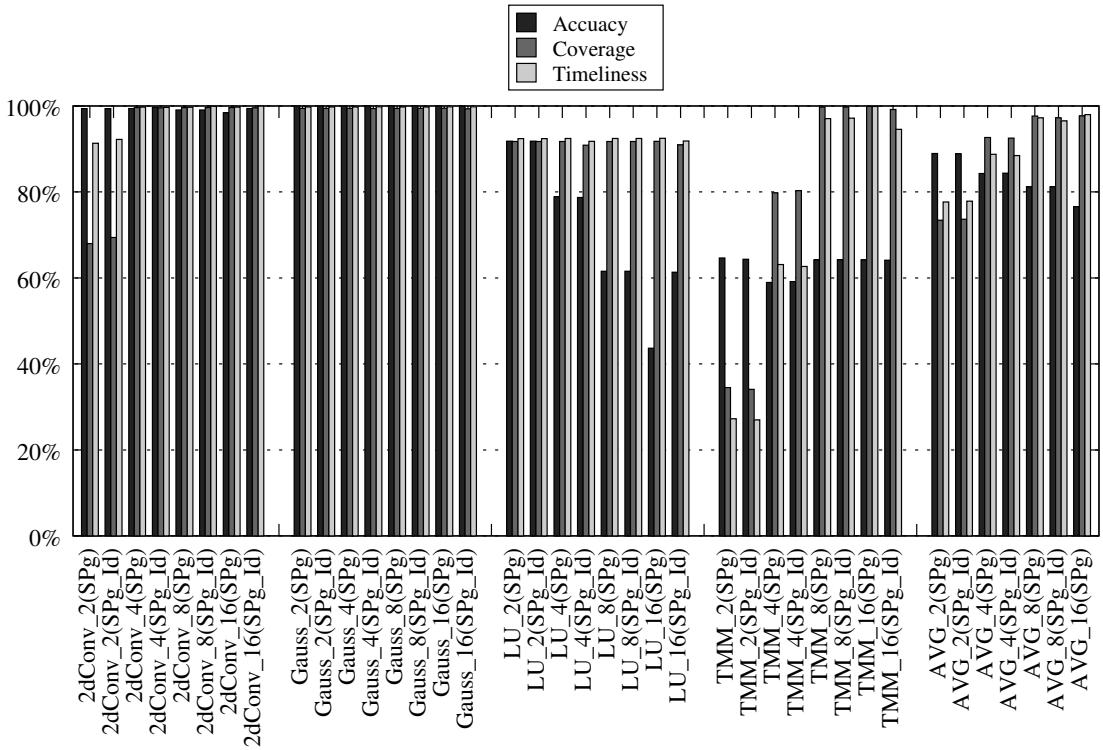



Figure 4.5: Prediction Accuracy, Coverage, and Timeliness for the four microbenchmarks with ($SP_g/I_d$) and without ($SP_g$) integrity verification

Figure 4.5 provides results for prediction accuracy, coverage, and timeliness for the four microbenchmarks with and without Integrity Verification. 2d Convolution, LU Decomposition, and Gaussian Elimination have good accuracy, coverage and timeliness. Interestingly although TMM has good coverage and timeliness when using stream buffers of size 16, its accuracy is consistently around $60\%$. Also, across all benchmarks, prediction accuracy, coverage, and timeliness are not significantly different with and without Integrity Verification.

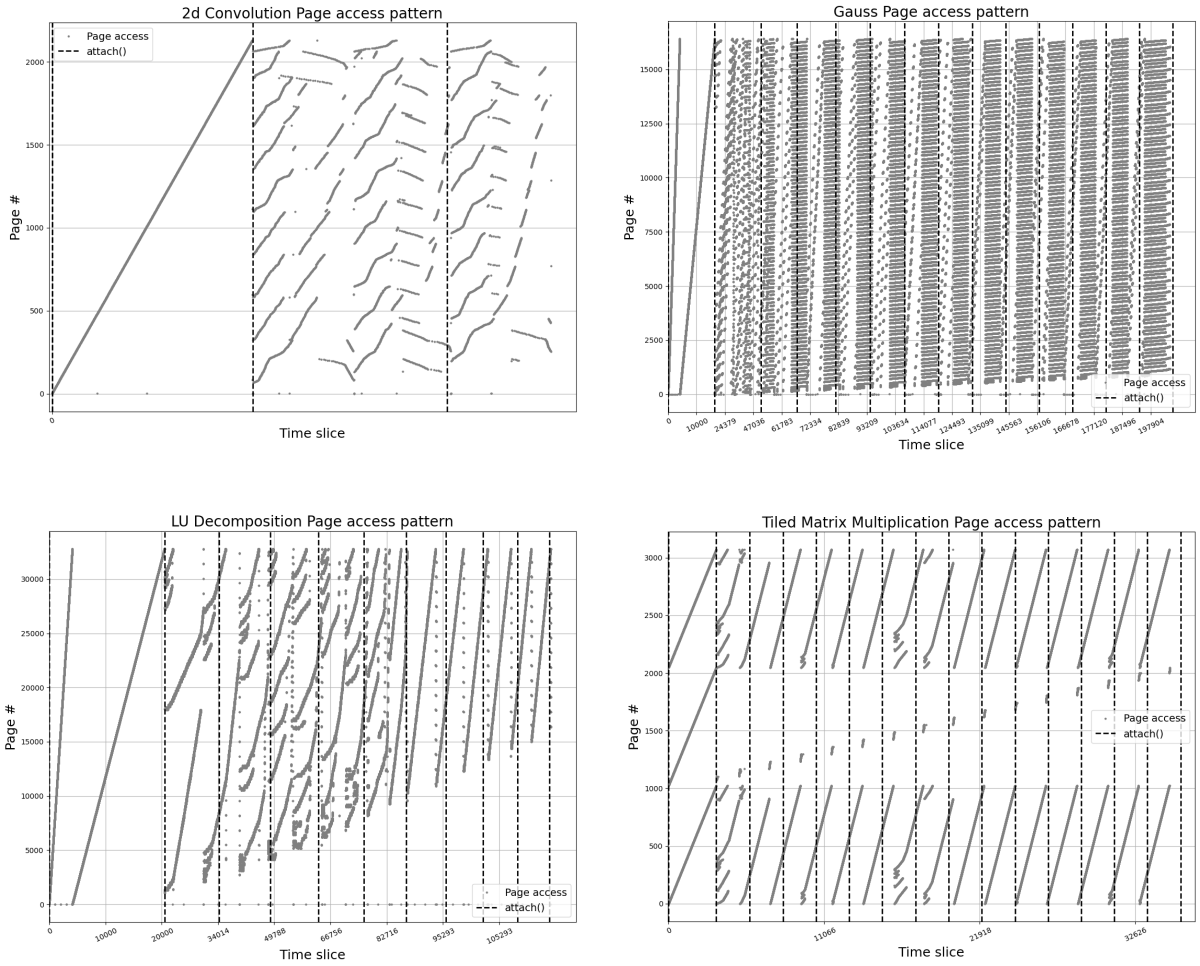



Figure 4.6: Access patterns. Clockwise from top left: 2d Convolution, Gaussian Elimination, Tiled Matrix Multiplication, and LU Decomposition

To determine whether these characteristics are expected for these workloads, Figure 4.6 shows the access patterns for each[1]. Each gray dot represents a page fault, while the dashed lines represent different attach calls.

It should be noted that the access pattern might be slightly different compared to the pattern that

[1]For performance reasons, the configurations for each benchmark are reduced from performance evaluations. 2d Convolution is $256$, Gauss is $4096$, LU is $2048$, and TMM is $1024 \times 4$

occurs when these benchmarks run without being observed, this is because because the page faults are printed to the kernel ring buffer, which is a slow operation [16]. Furthermore, the time stamps are numbered by page access, not the actual time. Finally, the access pattern will only print a page at first access after an attach call; this is because the kernel cannot see accesses to pages that have already been faulted (these pages are irrelevant since a faulted page is also a decrypted one).

Nevertheless, 2d Convolution, Gaussian Elimination, and LU Decomposition all have access patterns that stream buffers catch. Tiled Matrix Multiplication however, does not have a linear $1:1$ access pattern that stream buffers are optimized for, which is why its accuracy suffers.

Returning to Figure 4.4, the results now make more sense. LU's performance is best with a depth of 2, where all three metrics (timeliness, accuracy, and coverage) are around $91.7\%$, but accuracy starts to decline past that, which impacts the total execution time, and lengthens the Attach Stall time (which makes sense, since now unnecessarily decrypted pages need to be destroyed before the detach thread can complete). It also explains why TMM's total execution time never declines by any significant amount: the accuracy is always fairly low. In fact, the drop in accuracy is closely related to TMM's increase in execution time.

### *Filebench*

Figure 4.7 compares the I/O bandwidth of different Filebench workloads. Results are normalized to $SPg$ and reported for 8 threads with synchronization performed after every write/append operation. DRAM alone is $1.19\times$ faster than PM, while DRAM with prediction is $1.81\times$ faster on average. With Integrity Verification, on average, DRAM does not improve performance enough to make it faster than the PM only design, but prediction changes that: with prediction, it's $1.37\times$ faster on average compared to the original PM only design, meaning that across all the workloads, the performance penalties of including Integrity Verification are completely eliminated.

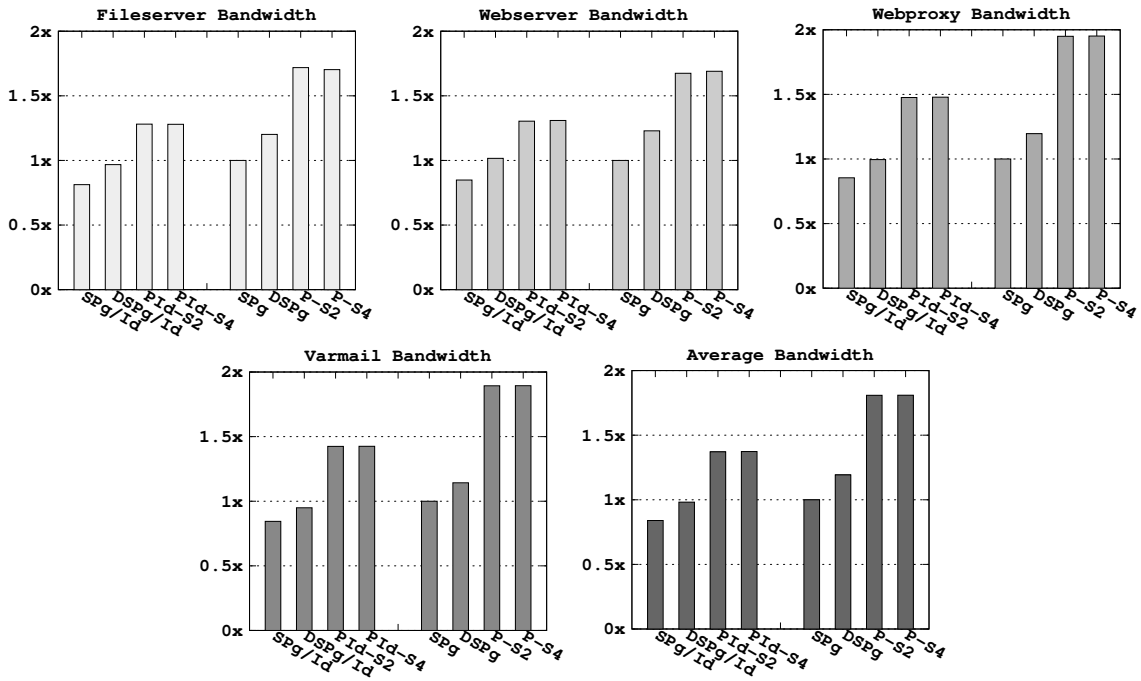


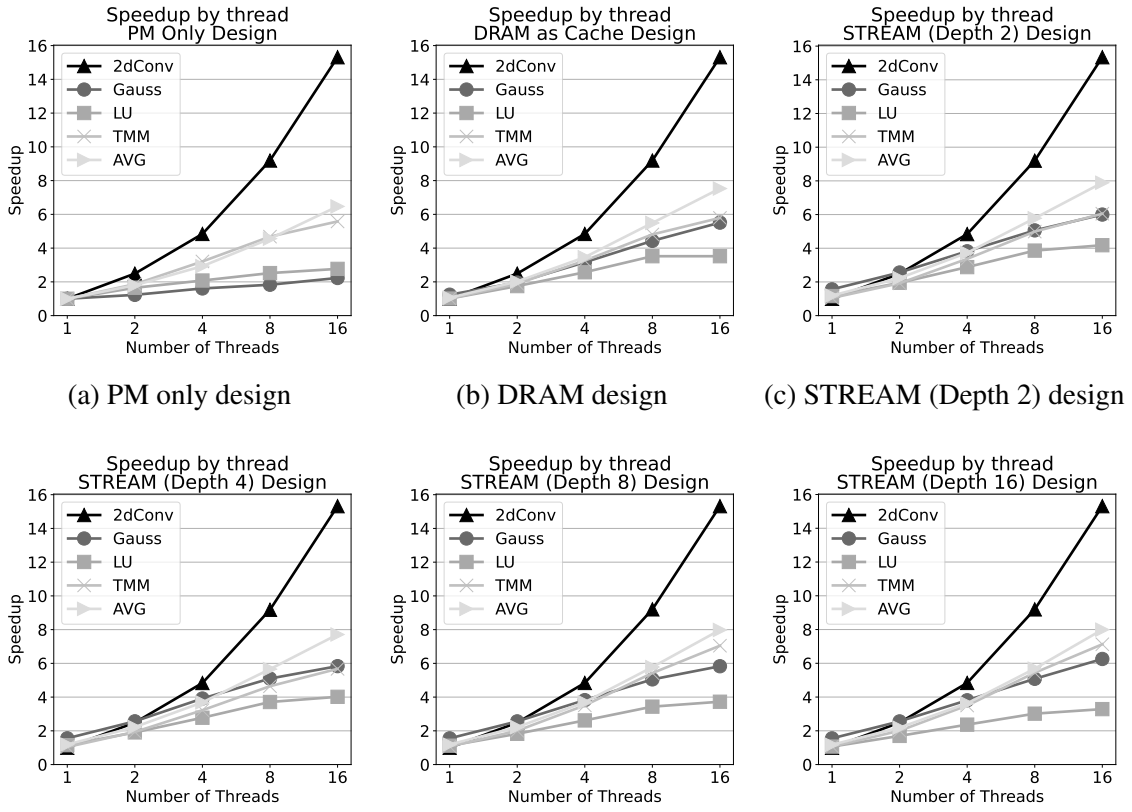

Figure 4.7: Filebench bandwidth for PM, DRAM and Prediction, with and without integrity verification

In contrast to the microbenchmarks, the Filebench workloads have nearly $100\%$ coverage, timeliness, and accuracy[2]; this is due to their workloads simulating I/O access patterns, which are usually sequential [46]. Stream buffers, therefore, perform well with these workloads.

### *Thread scalability*

Recall that one of the motivating factors for using DRAM as cache was thread scalability. As illustrated in the introduction, as the number of threads increase, performance does not follow. Figure 4.8 shows the thread-scalability of the PMO system's performance when using the best-case design, synchronized once per second, with an attach/detach size of $2$.

[2]All three have accuracy, coverage, and timeliness at $> 99.9\%$, regardless of prediction depth.




(a) PM only design (b) DRAM design (c) STREAM (Depth 2) design

(d) STREAM (Depth 4) design (e) STREAM (Depth 8) design (f) STREAM (Depth 16) design

Figure 4.8: Thread speedup by design

The results show that excessive numbers of writer threads continue to slow the fabric down, but the effect is blunted somewhat with $DSP_g$, as the average performance improvement at 16 is $7.53\times$ (compared to $6.47\times$ with $SP_g$). When using the $PDSP_g$, the speedup increases to $7.89\times$. With that design, the benchmarks are on average $25\%$ faster across all thread sizes, while they are only $16\%$ faster when using $DSP_g$. Therefore, prediction improves scaling based on the number of threads.

Figure 4.9 further breaks down the results, normalized to $SP_g$ (single thread) for Gauss, LU, TMM,

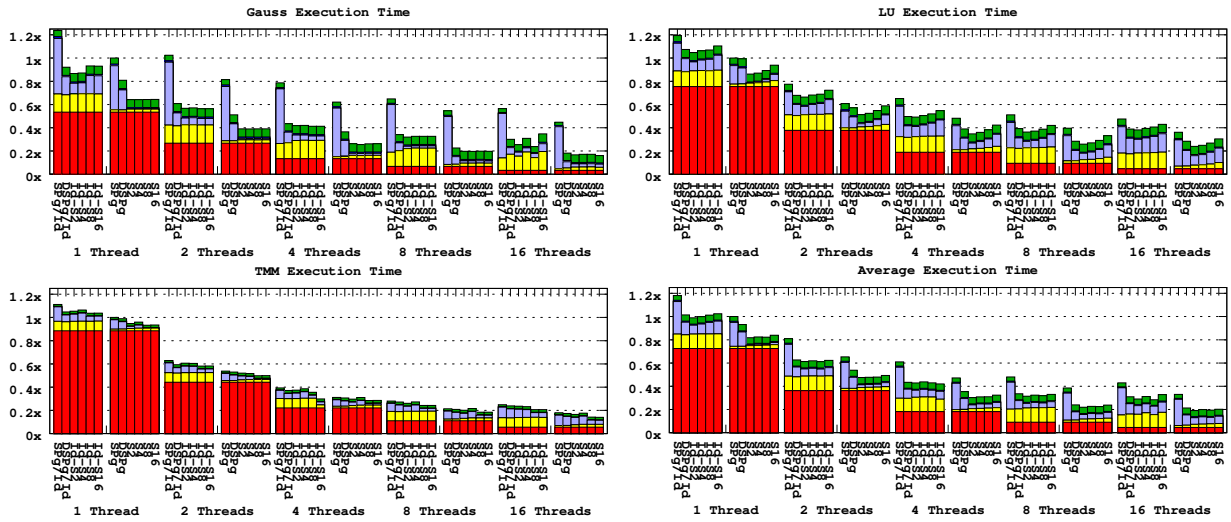



Figure 4.9: Thread scalability, by prediction depth.

and the averages of all of them[3]. The results show that the majority of the latency of PM (and to a lesser extent, DRAM) are from the overhead of page faults. Figure 4.9 also breaks down the results into different stream prediction sizes, from 1 to 16. It is clear that prediction greatly reduces the time spent waiting for page faults to complete. This behavior is expected since most of that time was spent performing page decryption. With a single thread, the latency caused by decrypting pages is almost completely eliminated; even at 16 threads the latency is significantly lessened. The figure also shows that the inclusion of Integrity Verification does not impact thread scalability by a significant amount.

### *Psync sensitivity*

Figure 4.10 evaluates the impact of psync on the various prediction designs. This figure varies the number of psyncs between an attach/detach session from 1 to 8, so that although there are an

[3] 2dConv is excluded in the interest of space and since it is compute-dominated.

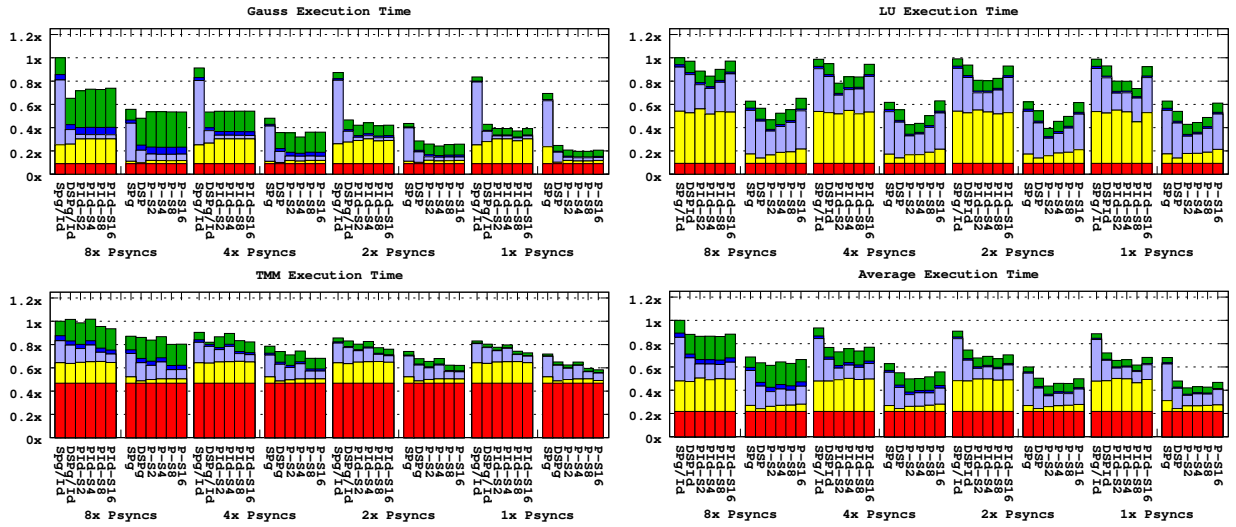



Figure 4.10: Psync sensitivity

equal amount of attach/detach calls in each run, psync occurs more or less often. The results are normalized to $SP_g$ at 8 psync calls per attach/detach. What's most interesting about these results is that the impact of prediction is lessened. For example, at $\times 2$ psyncs per attach/detach, the most performant depth size has an improvement of $10\%$ compared to the DRAM only design, but at $\times 8$ psyncs per attach/detach, the average speedup is only $2\%$. This implies that less frequent psync invocations will see more benefit from prediction. This makes sense: while page fault overhead is lessened by prediction, invocations of psync increase the amount that psync consumes as a total portion of the execution time. Although psync only persists those pages that are dirtied, in many cases most pages are dirtied between psync invocations. Also, using $DSP_g$ with TMM produces a counter-intuitive result: there is worse performance at $8\times$ psync invocation for $DSP_g$ compared to $SP_g$. This is because $SP_g$ encrypts the page from the shadow into the primary, while $DSP_g$ must both encrypt the page from the DRAM copy into the shadow and then copy from the shadow into the primary. As expected, psync is broadly slower with $DSP_g$ compared to $SP_g$ alone. A programmer using $DSP_g$ will therefore want to be mindful of the fact that psync is a

more expensive operation and avoid invoking it excessively.

## Conclusion

This chapter introduced DRAM-as-Cache and predictive decryption to PMOs, with the goal of further accelerating their performance without impacting security or reliability. Results show that compared to the prior most-performant design, introducing DRAM-as-Cache and predictive decryption has the potential to improve performance by $1.7\times$ (for DRAM-as-Cache alone) and $1.9\times$ (with predictive decryption). The performance impact of Integrity Verification is also lessened, and in some cases completely eliminated, when using DRAM and predictive encryption together.

## Acknowledgments

The material from this chapter was supported in part by the Office of Naval Research (ONR) under grant N00014-20-1-2750, and by the National Science Foundation (NSF) under grant 1900724.

When the material in this chapter is peer-reviewed and accepted for publication, the associated implementation of the PMO system, library, and related benchmarks will be found at `https://github.com/derrickgreenspan/Persistent-Memory-Objects`.

# CHAPTER 5: CONCLUSION

This chapter consists of two sections. The first section discusses possible avenues of future work. The second section concludes this dissertation.

## Future Work

This dissertation identifies three avenues for future work.

First, future work on PMOs should extend the work of Chapter 4 with more advanced page predictors, as well as more benchmarks. Stream buffer-like predictors work well for page access patterns that are linear, but many other workloads (and TMM) do not benefit from stream buffers.

Second, future work on PMOs should move away from Intel Optane as a base, and towards CXL [36] which is likely to be the locus of future work with persistent memory since Intel has ended production of Optane PMem. One avenue of research is to investigate memory-semantic SSDs which use CXL memory; one possible optimization is to seamlessly perform shadow writes via a form of logging that the SSD uses.

Finally, there is potential to optimize PMOs via a compiler pass such as an LLVM compiler pass. For example, Xu proposed [75] performing automatic attach/detach calls rather than rely on the programmer to manually insert them.

# Conclusion

This dissertation has described a persistent-memory abstraction called Persistent Memory Objects (PMO) and identifies a new threat model for PMOs. This dissertation presents an example security attack for exploiting PMOs at-rest and a defense mechanism against these types of attacks. Furthermore this dissertation describes a method of making PMOs crash-consistent through the use of the psync abstraction. This dissertation also implements the PMO abstraction within the Linux kernel on real Persistent Memory (Intel Optane PMem) hardware. This dissertation then goes on to extend the original PMO system with Low-Overhead at Rest PMO Protection (LOaPP) to provide for per-page encryption to protect at-rest PMOs and improve their performance, and introduces per-page integrity verification as well. Finally, this dissertation extends LOaPP with Predictive DRAM LOaPP (PD-LOaPP), which uses DRAM-As-Cache, and performs decryption ahead of time rather than on demand.

# LIST OF REFERENCES


[1] AKRAM, S. Performance evaluation of intel optane memory for managed workloads. *ACM Transactions on Architecture and Code Optimization (TACO) 18*, 3 (2021), 1–26.

[2] BALDASSIN, A., BARETTO, J., CASTRO, D., AND ROMANO, P. Persistent memory: A survey of programming support and implementations.

[3] BENSON, L., WEISGUT, M., AND RABL, T. What we can learn from persistent memory for cxl.

[4] BITTMAN, D., ALVARO, P., MEHRA, P., LONG, D. D., AND MILLER, E. L. Twizzler: a data-centric {OS} for non-volatile memory. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)* (2020), pp. 65–80.

[5] BITTMAN, D., ALVARO, P., MEHRA, P., LONG, D. D., AND MILLER, E. L. Twizzler: a data-centric os for non-volatile memory. *ACM Transactions on Storage (TOS) 17*, 2 (2021), 1–31.

[6] BITTMAN, D., ALVARO, P., AND MILLER, E. L. A persistent problem: Managing pointers in nvm. In *Proceedings of the 10th Workshop on Programming Languages and Operating Systems* (2019), pp. 30–37.

[7] BOLES, D., WADDINGTON, D., AND ROBERTS, D. A. Cxl-enabled enhanced memory functions. *IEEE Micro 43*, 2 (2023), 58–65.

[8] CHAUHAN, H., CALCIU, I., CHIDAMBARAM, V., SCHKUFZA, E., MUTLU, O., AND SUBRAHMANYAM, P. {NVMOVE}: Helping programmers move to {Byte-Based} persistence. In *4th Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 16)* (2016).

[9] CHEN, G., CHEN, S., XIAO, Y., ZHANG, Y., LIN, Z., AND LAI, T. H. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)* (2019), IEEE, pp. 142–157.

[10] CHOI, J., HONG, J., KWON, Y., AND HAN, H. Libnvmmio: Reconstructing software {IO} path with {Failure-Atomic}{Memory-Mapped} interface. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (2020), pp. 1–16.

[11] COMMUNITY, K. D. Block Cipher Algorithm Definitions. https://www.kernel.org/doc/html/v5.14/crypto/api-skcipher.html#symmetric-key-cipher-api .

[12] COSTAN, V., AND DEVADAS, S. Intel sgx explained. *Cryptology ePrint Archive* (2016).

[13] CUTRESS, I., AND TALLIS, B. Intel intel launches optane dimms up to 512gb: Apache pass is here! https://www.anandtech.com/show/12828/intel-launches-optane-dimms-up-to-512gb-apache-pass-is-here, Mar. 2018.

[14] DESNOYERS, P., ADAMS, I., ESTRO, T., GANDHI, A., KUENNING, G., MESNIER, M., WALDSPURGER, C., WILDANI, A., AND ZADOK, E. Persistent memory research in the post-optane era. In *Proceedings of the 1st Workshop on Disruptive Memory Systems* (2023), pp. 23–30.

[15] DOCUMENTATION, L. K. S. From https://kernel.org/.

[16] EDGE, J. A discussion on printk(), 2022. From LWN.net.

[17] ELKHOULY, R., ALSHBOUL, M., HAYASHI, A., SOLIHIN, Y., AND KIMURA, K. Compiler-support for critical data persistence in nvm. *ACM Transactions on Architecture and Code Optimization (TACO) 16*, 4 (2019), 1–25.

[18] ELNAWAWY, H., ALSHBOUL, M., TUCK, J., AND SOLIHIN, Y. Efficient checkpointing of loop-based codes for non-volatile main memory. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)* (2017), pp. 318–329.

[19] ELNAWAWY, H., ALSHBOUL, M., TUCK, J., AND SOLIHIN, Y. Efficient checkpointing of loop-based codes for non-volatile main memory. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)* (2017), IEEE, pp. 318–329.

[20] ELNAWAWY, H., ALSHBOUL, M., TUCK, J., AND SOLIHIN, Y. Efficient checkpointing of loop-based codes for non-volatile main memory. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)* (2017), pp. 318–329.

[21] EMELYANOV, P. Soft-dirty ptes, Apr 2013. From https://kernel.org/.

[22] FIALA, D., MUELLER, F., FERREIRA, K., AND ENGELMANN, C. Mini-ckpts: Surviving os failures in persistent memory. In *Proceedings of the 2016 International Conference on Supercomputing* (2016), pp. 1–14.

[23] GARCIA, A. M., GRIEBLER, D., SCHEPKE, C., AND FERNANDES, L. G. Spbench: a framework for creating benchmarks of stream processing applications. *Computing 105*, 5 (2023), 1077–1099.

[24] GORMAN, M. *Understanding the Linux virtual memory manager*. Prentice Hall Upper Saddle River, 2004.

[25] GREENSPAN, D. Llama-automatic memory allocations: an llvm pass and library for automatically determining memory allocations. In *Proceedings of the International Symposium on Memory Systems* (2019), pp. 363–372.

[26] GREENSPAN, D., MUSTAFA, N. U., DELGADO, A., BRAMHAM, C., PRATS, C., WALLACE, S., HEINRICH, M., AND SOLIHIN, Y. Loapp: Low-overhead at-rest pmo protection.

In *To appear in the 2024 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)* (2024), p. TBD.

[27] GREENSPAN, D., MUSTAFA, N. U., KOLEGA, Z., HEINRICH, M., AND SOLIHIN, Y. Improving the security and programmability of persistent memory objects. In *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)* (2022), pp. 157–168.

[28] GROUP, A. C. S. R. *IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7*. 2018.

[29] HENSON, M., AND TAYLOR, S. Memory encryption: A survey of existing techniques. *ACM Computing Surveys (CSUR) 46*, 4 (2014), 1–26. Publisher: ACM New York, NY, USA.

[30] HILL, J., BLANCO, J., SHEY, J., RAKVIC, R., AND WALKER, O. Toward classification of phase change memory and 3d nand flash ssds using power-based side-channel analysis in the time-domain. In *Proceedings of the 2022 International Symposium on Memory Systems* (2022), pp. 1–7.

[31] INTEL. *Persistent Memory Programming*. Aug. 2016.

[32] INTEL, R. . Intel architecture memory encryption technologies, 2022.

[33] INTEL, R. . 12th generation intel core processors, 2023.

[34] IZRAELEVITZ, J., YANG, J., ZHANG, L., KIM, J., LIU, X., MEMARIPOUR, A., SOH, Y. J., WANG, Z., XU, Y., DULLOOR, S. R., ET AL. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714* (2019).

[35] JOSEPH, D., AND GRUNWALD, D. Prefetching using markov predictors. In *Proceedings of the 24th annual international symposium on Computer architecture* (1997), pp. 252–263.

[36] JUNG, M. Hello bytes, bye blocks: Pcie storage meets compute express link for memory expansion (cxl-ssd). In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems* (2022), pp. 45–51.

[37] KADEKODI, R., LEE, S. K., KASHYAP, S., KIM, T., KOLLI, A., AND CHIDAMBARAM, V. Splitfs: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (2019), pp. 494–508.

[38] KANNAN, S., GAVRILOVSKA, A., AND SCHWAN, K. pvm: persistent virtual memory for efficient capacity scaling and object storage. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), pp. 1–16.

[39] KAPLAN, D., POWELL, J., AND WOLLER, T. Amd memory encryption. *White paper* (2016).

[40] KAWAHARA, T., ITO, K., TAKEMURA, R., AND OHNO, H. Spin-transfer torque ram technology: Review and prospect. *Microelectronics Reliability 52*, 4 (2012), 613–627.

[41] KHAN, A., SIM, H., VAZHKUDAI, S. S., AND KIM, Y. Mosiqs: Persistent memory object storage with metadata indexing and querying for scientific computing. *IEEE Access 9* (2021), 85217–85231.

[42] KHAN, A., SIM, H., VAZHKUDAI, S. S., MA, J., OH, M.-H., AND KIM, Y. Persistent memory object storage and indexing for scientific computing. In *2020 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)* (2020), IEEE, pp. 1–9.

[43] KLAUSER, C. Lu decomposition and matrix multiplication with openmp, 2011.

[44] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., ET AL. sel4: For-

mal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), pp. 207–220.

[45] LAGA, A., BOUKHOBZA, J., KOSKAS, M., AND SINGHOFF, F. Lynx: A learning linux prefetching mechanism for ssd performance model. In *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)* (2016), IEEE, pp. 1–6.

[46] LI, C., SHEN, K., AND PAPATHANASIOU, A. E. Competitive prefetching for concurrent sequential i/o. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (2007), pp. 189–202.

[47] MADIEU, J. *Linux Device Drivers Development: Develop Customized Drivers for Embedded Linux*. Packt Publishing Ltd, 2017.

[48] MIRONOV, V., CHERNYKH, I., KULIKOV, I., MOSKOVSKY, A., EPIFANOVSKY, E., AND KUDRYAVTSEV, A. Performance evaluation of the intel optane dc memory with scientific benchmarks. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)* (2019), IEEE, pp. 1–6.

[49] MORRIS, J. Kernel korner: the linux kernel cryptographic api. *Linux Journal 2003*, 108 (2003), 10.

[50] MUSTAFA, N. U., AND SOLIHIN, Y. Persistent memory security threats to inter-process isolation. *IEEE Micro* (2023).

[51] MUSTAFA, N. U., XU, Y., SHEN, X., AND SOLIHIN, Y. Seeds of seed: New security challenges for persistent memory. In *2021 International Symposium on Secure and Private Execution Environment Design (SEED)* (2021), IEEE, pp. 83–88.

[52] NILSSON, A., BIDEH, P. N., AND BRORSSON, J. A survey of published attacks on intel sgx. *arXiv preprint arXiv:2006.13598* (2020).

[53] PATIL, O., IONKOV, L., LEE, J., MUELLER, F., AND LANG, M. Performance characterization of a dram-nvm hybrid memory architecture for hpc applications using intel optane dc persistent memory modules. In *Proceedings of the International Symposium on Memory Systems* (2019), pp. 288–303.

[54] PATTERSON, L., PIGOROVSKY, D., DEMPSEY, B., LAZAREV, N., SHAH, A., STEINHOFF, C., BRUNO, A., HU, J., AND DELIMITROU, C. Hivemind: a hardware-software system stack for serverless edge swarms. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (2022), pp. 800–816.

[55] PENG, I. B., GOKHALE, M. B., AND GREEN, E. W. System evaluation of the intel optane byte-addressable nvm. In *Proceedings of the International Symposium on Memory Systems* (2019), pp. 304–315.

[56] PLANK, J. S., KIM, Y., AND DONGARRA, J. J. Algorithm-based diskless checkpointing for fault tolerant matrix operations. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers* (1995), IEEE, pp. 351–360.

[57] REN, J., ZHAO, J., KHAN, S., CHOI, J., WU, Y., AND MUTIU, O. Thynvm: Enabling software-transparent crash consistency in persistent memory systems. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2015), IEEE, pp. 672–685.

[58] ROBERTS, P. Mit: Discarded hard drives yield private info. *ComputerWorld 16* (2003).

[59] ROGERS, B., SOLIHIN, Y., AND PRVULOVIC, M. Memory predecryption: hiding the latency overhead of memory encryption. *ACM SIGARCH Computer Architecture News 33*, 1 (2005), 27–33.

[60] SAKALIS, C., LEONARDSSON, C., KAXIRAS, S., AND ROS, A. Splash-3: A properly synchronized benchmark suite for contemporary research. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2016), pp. 101–111.

[61] SANFILIPPO, S., AND NOORDHUIS, P. The redis documentation, 2016.

[62] SCARGALL, S. *Programming persistent memory: A comprehensive guide for developers*. Springer Nature, 2020.

[63] SOLIHIN, Y. Persistent memory: Abstractions, abstractions, and abstractions. *IEEE Micro 39*, 1 (2019), 65–66.

[64] SOLIHIN, Y., LEE, J., AND TORRELLAS, J. Using a user-level memory thread for correlation prefetching. *ACM SIGARCH Computer Architecture News 30*, 2 (2002), 171–182.

[65] SYSTEMS, C. CyberPower PR3000 Sinewave UPS Specifications. https://www.cyberpowersystems.com/product/ups/smart-app-sinewave/pr3000/.

[66] TARASOV, V., ZADOK, E., AND SHEPLER, S. Filebench: A flexible framework for file system benchmarking. *USENIX; login 41*, 1 (2016), 6–12.

[67] UL MUSTAFA, N., XU, Y., SHEN, X., AND SOLIHIN, Y. Seeds of seed: New security challenges of persistent memory. In *IEEE International Symposium on Secure and Private Execution Environment Design (SEED)* (Virtual, 2021), SEED Organizing Committee 2021.

[68] VOLOS, H., TACK, A. J., AND SWIFT, M. M. Mnemosyne: Lightweight persistent memory. *ACM SIGARCH Computer Architecture News 39*, 1 (2011), 91–104.

[69] WANG, T., SAMBASIVAM, S., SOLIHIN, Y., AND TUCK, J. Hardware supported persistent object address translation. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2017), IEEE, pp. 800–812.

[70] XU, J., AND SWANSON, S. {NOVA}: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)* (2016), pp. 323–338.

[71] XU, J., ZHANG, L., MEMARIPOUR, A., GANGADHARAIAH, A., BORASE, A., DA SILVA, T. B., SWANSON, S., AND RUDOFF, A. Nova-fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), pp. 478–496.

[72] XU, J., ZHANG, L., MEMARIPOUR, A., GANGADHARAIAH, A., BORASE, A., DA SILVA, T. B., SWANSON, S., AND RUDOFF, A. Nova-fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), pp. 478–496.

[73] XU, Y., SOLIHIN, Y., AND SHEN, X. Merr: Improving security of persistent memory objects via efficient memory exposure reduction and randomization. Association for Computing Machinery.

[74] XU, Y., XU, W., KEETON, K., AND CULLER, D. E. Softpm: Software persistent memory. In *13th Non-Volatile Memories Workshop (NVMW)* (2022).

[75] XU, Y., YE, C., SHEN, X., AND SOLIHIN, Y. Temporal exposure reduction protection for persistent memory. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)* (2022), IEEE, pp. 908–924.

[76] XU, Y., YE, C., SHEN, X., AND SOLIHIN, Y. Temporal exposure reduction protection for persistent memory. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)* (2022), IEEE, pp. 908–924.

[77] XU, Y., YE, C., SOLIHIN, Y., AND SHEN, X. Hardware-based domain virtualization for intra-process isolation of persistent memory objects. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)* (2020), IEEE, pp. 680–692.

[78] YANG, J., KIM, J., HOSEINZADEH, M., IZRAELEVITZ, J., AND SWANSON, S. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)* (Santa Clara, CA, Feb. 2020), USENIX Association, pp. 169–182.

[79] YANG, J., LI, B., AND LILJA, D. J. Exploring performance characteristics of the optane 3d xpoint storage technology. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS) 5*, 1 (2020), 1–28.

[80] YOUNG, V., NAIR, P. J., AND QURESHI, M. K. DEUCE: Write-Efficient Encryption for Non-Volatile Memories. *ACM SIGARCH Computer Architecture News 43*, 1 (Mar. 2015), 33–44.