Persistent Memory Objects on the Cheap

Derrick Greenspan
derrick.greenspan@ucf.edu
University of Central Florida
College of Engineering and Computer Science
Orlando, Florida, USA

Naveed Ul Mustafa
num@nmsu.edu
New Mexico State University
Department of Computer Science
Las Cruces, New Mexico, USA

Jongouk Choi
jongouk.choi@ucf.edu
University of Central Florida
College of Engineering and Computer Science
Orlando, Florida, USA

Mark Heinrich
heinrich@ucf.edu
University of Central Florida
College of Engineering and Computer Science
Orlando, Florida, USA

Yan Solihin
yan.solihin@ucf.edu
University of Central Florida
College of Engineering and Computer Science
Orlando, Florida, USA

ABSTRACT

Persistent Memory Objects (PMOs) are the state-of-the-art OS-based approach for persistent memory (PM) management. Recent PMO designs have limited performance due to the properties of the PM substrate. To address this challenge, this paper introduces LPMO, or lightweight PMOs, that enables two key performance optimization techniques: software-based DRAM caching and prediction. First, through DRAM caching, LPMO moves reads/writes to a faster medium, while retaining crash consistency. Second, LPMO introduces software-based predecryption to predict when pages might be used and decrypt them ahead of time.

Our evaluation shows that software-based DRAM Caching and software-based predecryption with LPMO can improve the performance of a PMO system by up to 1.25× compared to the prior state-of-the-art implementations when using LPMO locally. When bundled with a stream predictor, the improvement reaches 1.81×, depending on the workload.

To further demonstrate the flexibility and performance benefits of our LPMO design, we evaluated our solution in a CXL memory system and introduce a CXL memory hierarchy that our LPMO system can configure. In such a CXL system, when integrated with CXL Enhanced Memory Functions (EMFs) that perform encryption in hardware, LPMO is capable of performing comparably or in some workloads *faster* than the prior state-of-the-art design, despite the added latency of CXL memory.

CCS CONCEPTS

• Software and its engineering → Memory management; **• Hardware → Non-volatile memory**.

KEYWORDS

Persistent memory, CXL, memory abstractions

ACM Reference Format:
Derrick Greenspan, Naveed Ul Mustafa, Jongouk Choi, Mark Heinrich, and Yan Solihin. 2025. Persistent Memory Objects on the Cheap. In *2025 International Conference on Supercomputing (ICS '25), June 8–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3721145.3734533

1 INTRODUCTION

Persistent Memory (PM) allows fast byte-addressable data access (like DRAM) but retains data on power loss (like storage). Examples include memory-semantic SSDs and the discontinued [30] Optane Pmem [28]. PM enables utilizing persistent data in a byte-addressable fashion, either to host a file system [40], or to use file-less abstractions such as the Persistent Memory Objects (PMOs) [37, 41, 42]. PMOs hold (potentially pointer-rich) data structures that can be directly accessed through `load`/`store` instructions, but have naming, permission, and crash consistency managed by the Operating System (OS). After being created, a user process must `attach()` a PMO into its address space to successfully access PMO data, and can later unmap it by calling `detach()`. All updates to the PMO data are made persistent at explicit points in the

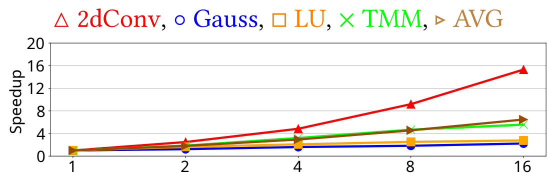


Figure 1: Microbenchmarks [12] scalability by thread count using PM (with the x-axis representing the number of threads).

program via `psync()` system calls. The semantics of psync are atomic and crash consistent: it fully succeeds or fully fails, and if a crash happens, the PMO state is guaranteed to revert to one from the last successful psync. This is achieved through *shadowing*, where modifications are performed on a shadow PMO copy and merged into the primary copy upon psync.

During its lifetime, a PMO is either *in-use* (i.e., attached to a process) or at *at-rest* (i.e., detached from any processes). Since many PMOs are expected to spend most of their lifetime at-rest (like files), similar to the file system, they need to be encrypted and integrity-protected while at-rest [14]. Therefore, PMOs must first be decrypted and their integrity verified at attach time, and encrypted and protected by a message authentication code (MAC) at detach time. However, the attach-time decryption and integrity check introduce substantial performance penalties, both in terms of the critical-path delay and scalability.

To analyze this problem, we evaluated the state-of-the-art LOaAPP PMO design [14] in a real system with Intel Pmem. We ran four benchmark applications: 2d Convolution, Gaussian Elimination, LU Decomposition, and Tiled Matrix Multiplication, varying the number of threads from 1 to 16. Figure 1 shows the speedup ratios of several benchmarks from 1 to 16 threads with the state-of-the-art Linux-based PMO [14]. Except for 2d Convolution, which is not particularly memory intensive, all benchmarks show poor speedup scaling, e.g. only $\approx 3\times$ at 16 threads for LU.

There are several reasons for the high performance overheads and low scalability. PMOs are hosted entirely in the PM [14, 15] where access latencies are high (especially writes) and write bandwidth is low, while DRAM is completely unutilized. This makes sense to ensure no data is lost on a crash. However, PMOs are particularly challenging to use beyond PM-based memory systems. Adapting PMOs to disaggregated memory or server systems, such as those leveraging CXL with heterogeneous memory devices [13, 25, 39], is difficult. In these environments, PMOs storage of data directly in PM not only introduces the inherent overhead of the CXL controller but also adds additional switching overheads. Furthermore, this approach underutilizes both CXL-attached DRAM and local DRAM, leading to inefficient memory usage and diminished performance.

To address these performance and scalability challenges, we found that leveraging the semantics of psync, which allows for *some* data loss up to the last successful psync, is critical. By understanding this, we can optimize the usage of DRAM with PMOs. In particular, we propose placing shadow pages on DRAM where modifications are performed, and at psync, these modifications are then merged to the primary copy of the PMO. If a crash occurs, pages in DRAM are lost, but we can still adhere to the crash consistency semantics of psync [15] by ensuring that there is always a valid copy in PM during the process.

While placing shadow pages in DRAM improves performance thanks to faster accesses, there are new challenges that arise. One challenge is that accessing a DRAM-resident shadow page still requires decryption and integrity verification following a page fault; unless this critical-path delay is addressed, its performance gain may be severely constrained. Second, it is unclear whether the shadow page should be kept in plaintext or ciphertext in DRAM. To address these two challenges, we propose a *predictor* whereby the next *n* pages that are likely accessed are faulted, decrypted, and verified in advance. This not only can potentially hide decryption and integrity verification latencies, it can also hide page fault delay. We refer to our solution for more performant security of at-rest PMOs as *Light PMO*, or LPMO.

The LPMO system must be designed carefully. Since the predictor itself runs in the kernel, it must be lightweight and quick to generate predictions. At the same time, the predictor must achieve high *accuracy* (i.e., how many predicted decryptions are later found to be correct), high *coverage* (i.e., what percentage of decrypted pages were decrypted ahead of time by the predictor), and *timeliness* (i.e., the percentage of predicted pages that were fully decrypted by the time the page fault occurs).

Finally, LPMO leverages a software-based memory tiering mechanism to allow the designation of any memory devices as software-managed (exclusive) caches, offering (logically) reconfigurable memory hierarchies. This property is ideal for disaggregated memory systems such as the types of systems that CXL envisions; it allows us to leverage local DRAM or CXL-attached DRAM as caches for free, thereby improving performance by reducing the need for frequent access to slower memory tiers. We demonstrate that LPMO is viable on CXL devices, despite their added latency. To the best of our knowledge, LPMO is the first work to enable secure handling of persistent data in CXL-enabled PM.

We implemented LPMO on a real OS (Linux) and evaluated it on a real system that has a mixture of DRAM and Intel Optane Pmem as main memory, in both local and (simulated) CXL-memory configurations. We found that LPMO achieves up to 1.25× speedup over a state-of-the-art PMO. When combined with page prediction, the speedup improves further to 1.81×. In CXL-memory configurations, LPMO can improve the performance by up to 3× compared to the local PM-based PMO systems, despite the added latency of CXL. Overall, we make the following **contributions** in this work:

(1) We propose utilizing DRAM and leverage psync semantics to place shadow pages in DRAM.
(2) We propose a predictor to perform page faults, decryptions, and integrity verification for pages for which their shadows are likely to be created in the near future.
(3) We implement LPMO and integrate it into a real OS.
(4) We extensively evaluate LPMO's performance for several workloads and perform a thorough design space exploration. We show that LPMO achieves up to 1.25× speedup over a state-of-the-art PMO system (LOaPP [14]). When combined with page prediction, the speedup improves further to 1.81×. We also demonstrate that LPMO achieves better scaling as the number of threads increases.
(5) We show that LPMO can be used with CXL-enabled PM devices and negate the performance penalty from CXL switch latency, and that LPMO lends itself well to different CXL architecture configurations.

2 BACKGROUND AND MOTIVATION

2.1 Persistent Memory Objects

In contrast to hosting a file system on PM, PMOs host data structures in PM without file backing. They allow direct access using loads/stores and avoids the complexity of reconciling file metadata semantics and virtual memory semantics. Furthermore, PMOs can use the entire PM device's write bandwidth [21], which is important given the limited write bandwidth in PM when compared to DRAM.

The PMO design treats PM as a collection of byte-addressable *objects*. Persistent data within a PMO are stored in structures that have the potential to be pointer-rich, and the required metadata are handled by the kernel. A PMO is created with `pcreate()`, attached into the address space of the calling process with `attach()`, severed from the calling process with `detach()`, and synchronized with `psync()`. We adapt the Greenspan PMO (GPMO) [15, 31] system with LOaPP [14]'s optimizations, as it is the only available PMO system that works on real hardware and has been implemented on a real system.

2.2 Crash Consistency

A technical failure can cause data stored within a PM system to be corrupted as a result of partial or improperly ordered writes. To prevent this, PM systems must ensure *crash consistency*. Logging and shadowing are two methods for achieving crash-consistency; logging tends to be used with filesystem approaches [40], 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.

2.3 Per-page encryption

Prior works with PMOs observe that their performance when using integrity verification and encryption are lacking, especially with larger PMOs, because the entire PMO is encrypted/decrypted and its integrity is validated at every attach/detach [15]. To fix this, a recent work, LOaPP [14] breaks PMOs into separate pages to improve performance. However, that work does not utilize DRAM and does not perform prediction or perform decryption ahead of time.

2.4 Cache and Page Prediction

Most prior work involving data prefetching are performed in hardware for the cache, and involve either stream buffers or correlation tables. Stream buffers provide spatial locality by copying contiguous blocks of memory into the local cache after a previous cache miss. In contrast, correlation tables provide temporal locality by correlating cache misses with accesses to predict when a page might be needed [17]. Most prefetching designs are hardware-based, such as with a prefetching engine [36] or a correlation table that intercepts L2 Cache calls [38]. While we leverage such prediction techniques, our design distinguishes itself by running the prediction in software, integrated into the kernel and its page fault handlers. As such, our design needs to be is attractive since it requires no hardware changes, though it needs to be lightweight to be effective.

2.5 Compute Express Link Memory

Compute Express Link (CXL) has obtained significant research interest thanks to its exceptional capabilities in managing hardware heterogeneity and enabling resource disaggregation. In this context, researchers have also introduced CXL-attached PM for secondary memory expansion [1, 9, 45]. CXL memory devices use the CXL protocol which physically uses the PCIe interface [9] and allows for attaching/detaching arbitrary devices to a system while retaining cache coherency between the device and the processor. There are three types of CXL devices. CXL Type 1 devices possess cache coherency with host memory but do not have their own separate memory, only a separate cache and the ability to directly access host memory. CXL Type 2 devices provide bidirectional cache coherency and usually possess their own device memory; both the host and the device itself can access each other's memories directly. CXL Type 3 devices operate as memory-expander devices. These devices have no cache, only memory that can be accessed and cached by the host CPU. For this work, we assume our *CXL-based PM uses Type 3 Memory.*

2.5.1 Flat Memory Hierarchy. PM devices that use CXL are often directly connected to the CXL controller, presenting itself as flat memory to the host [45]. Due to this flat memory hierarchy, applying PMO to CXL-attached PM inherently exposes the long latency of PM operations to the host. This configuration forces the host to treat CXL-attached PM as main memory, resulting in unavoidable latency overheads, such as a 3x slowdown in memory operations [25]. This is compounded by the fact that PM has pathologies that cause performance degradation when multiple threads attempt to write to it at the same time [20]; some applications are impacted by this more than others.

2.5.2 CXL Protocols. CXL contains three separate protocols:

CXL.io*.* The CXL.io protocol manages device configuration, memory-mapped I/O (MMIO), interrupts, and DMA operations, and monitors the status of devices. CXL.io ensures the system is aware of the available devices and can communicate with them. While it handles I/O tasks and device management, it does not directly participate in memory operations but remains essential for setting up and maintaining the operational state of CXL devices.

CXL.cache*.* The CXL.cache protocol facilitates cache coherency between the host and CXL-attached devices. It is designed to enable devices, such as accelerators and network adapters, to cache data from the host's memory, ensuring that data remains consistent across multiple caches. This protocol allows devices to read, modify, and write back cache lines while maintaining coherency, similar to traditional cache-coherency protocols like MESI.

CXL.mem*.* The CXL.mem protocols enables direct access to memory on CXL-attached devices by mapping remote memory into the system address space. It allows the host processor to treat CXL memory devices as an extension of its own memory, similar to accessing local memory. The protocol ensures that memory devices attached via CXL can be virtualized using a hypervisor, making it compatible with virtualization technologies and useful for cloud and data center environments.

CXL.mem allows for dynamic memory expansion and the creation of large, shared memory pools that can be accessed by multiple hosts. In particular, the protocol specifies how data are transacted between the CPU and memory devices on the CXL bus, and is broken up into memory requests from the master to subordinate (M2S) and memory responses from the subordinate to master (S2M).

2.5.3 CXL Access Latency. The performance characteristics of CXL devices show that CXL memories are significantly faster than alternatives such as RDMA [13] (8.3× faster), but still far slower than local DRAM (5.5× slower). In general, the latency of accessing memory through CXL memory pools are higher than the latency of accessing memory through NUMA [39], but it can vary significantly depending on the number of switches used within the pool. For our work, we use only one, which makes it easier for us to treat NUMA as a proxy for CXL devices.

2.5.4 CXL at-rest encryption. To the best of our knowledge, there have been no proposed work for at-rest encryption for CXL-attached PMs. The closest work is ShieldCXL [3], which utilizes CXL's support for at-rest encryption to produce Oblivious RAM (ORAM). However, ORAM focuses on mitigating side-channel attacks, which is not our threat model (see Section 3).

One might suggest utilizing enhanced memory functions (EMFs) as articulated by [1] and provided by the CXL protocol. These EMFs include encryption/decryption of memory at rest. In particular, CXL 3.1 recently introduced the Trusted Execution Environment (TEE) Security Protocol (TSP) for CXL devices. The CXL TSP provides for, among other things, at-rest memory encryption that are within the target memory device [6].

There are two types of at-rest memory encryption on the target device that can be used with CXL. First, there is *Range-based Memory Encryption*, where the initiating device configures a specific Host Physical Address (HPA) range to use a specific encryption key. Second, there is *Context Key Identifier (CKID)-based Memory Encryption*, where a specific key is utilized for a given CXL transaction, rather than for an entire memory range. However, this has not yet been utilized for at-rest memory security; we discuss it in more detail in Subsection 5.2.1.

2.5.5 Integrity Verification. Unfortunately, the CXL specification 3.1 *does not* provide at-rest integrity verification within the TEE; rather, it only provides assurances of integrity verification as the data moves down the wire from within the CXL domain. Although data integrity is important for at-rest PM security, the integrity verification is outside of the scope of the CXL TEE, in Subsection 5.2.2, we discuss how LPMO achieves data integrity on CXL devices.

2.5.6 Persistent Memory Objects in CXL. In this work, we argue that we can unlock the full potential of CXL memory by 1) allowing for a reconfigurable memory hierarchy for LPMO systems, and 2) for the first time, utilizing the CXL TEE for at-rest encryption.

3 THREAT AND TRUST MODEL

Similar to files, PMOs keep persistent data for a long time, and most of them will spend most of their lifetime at rest. Our threat model assumes 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. We assume that the attacker knows or has the capability to find the location of the target PMO in memory.

One attack we consider is data remanence, where a stolen or improperly disposed of PM may be analyzed by the attacker to obtain sensitive data. Such attacks have long been documented for files in hard drives [35] and prompted filesystem encryption which is widely used today. Likewise, PMO encryption was motivated by the same concerns [14].

Another attack we consider has the adversary compromise a user account to steal or corrupt PMO data. But without having the correct key, the attacker cannot read the plaintext of PMO data, or modify PMO data without being discovered later. PMO encryption keys can be further managed by a Trusted Platform Module (TPM) to avoid the attacker reading keys from memory.

Since our design adopts the threat model of filesystem encryption (i.e., protecting data at-rest rather than in-use.), protecting plaintext data in DRAM from data remanence attacks is out of our scope. However, our LPMO system *can* be integrated with well-known DRAM protection mechanisms such as MBIST for memory resetting, zeroization [26], Intel Total Memory Encryption (TME) [2], or AMD Secure Memory Encryption (SME) [22], which have low overheads since they rely on hardware. Similarly, side-channel attacks are out of scope for our work too, as our threat model follows that of prior PMO works [14, 15, 31, 32, 42, 43].

Our trust is limited to specific components of the system software, Linux Kernel Crypto API [5], crucial kernel memory functions like memcpy and memset, and our PMO kernel subsystem. We assume that these components are devoid of any code vulnerabilities, which is plausible because the total code size is small enough for formal verification [23]. Finally, we trust the encryption hardware of the CPU.

We describe the steps of the attack. Lacking either encryption or integrity verification, it proceeds as follows: 1) The attacker discovers the physical address (PA) of the PMO. 2) The attacker maps the PMO to its address space, and 3) performs the attack by either corrupting the PMO by writing incorrect data, which will not be caught in the absence of integrity verification [32], or alternatively, in the absence of encryption, the attacker can read from the PMO and steal secrets from it. In either case, the attacker is able to do these actions silently, by simply 4) unmapping the address space of the PMO from the attacker process's address space, which leaves no evidence that the PMO was accessed or modified.

4 LPMO DESIGN

4.1 DRAM Shadow Paging

Prior PMO systems placed each PMO entirely in the PM, both shadow and primary pages. While in this approach the PMO retains data on power loss (or crashes), it suffers from high access latencies (especially writes) and low write bandwidth. Furthermore, a typical memory system with PM has a mixture of DRAM and PM, and the DRAM is underutilized. To benefit from DRAM, we could consider placing both primary and shadow pages in DRAM. However, while accesses are fast, data in such pages will be lost on crashes. Thus, we propose a solution where only the shadow pages are placed in DRAM, while primary pages are placed in PM. On a crash, only the shadow pages are lost.

Although our approach incurs some risk of data loss, we note that as long as the data loss is limited to any modifications after the last successful `psync()`, the PMO semantics are not violated. To benefit from DRAM while adhering to the PMO semantics, our solution requires that data in shadow pages are merged into the primary pages on each psync (after being flushed from volatile caches). The cost of this merging is that psync takes slightly longer to complete. However, considering that reads and writes occur much more frequently than psync, this trade-off is worth it.

Another design issue is whether the shadow and primary pages should stay encrypted or decrypted. First, if both pages stay encrypted, the shadow will be in DRAM while the primary remains in PM. This approach is infeasible (at least without special hardware support) because PMO data is accessed using loads/stores which are in plaintext. The second possible approach is to keep both pages in plaintext. Here, every time a PMO page is written, the page fault decrypts the primary page, and allocates a decrypted shadow page in DRAM. This allows the page to be accessible using regular loads/stores, and psync involves simple copying from shadow to primary. However, it requires the primary pages to be encrypted in PM immediately upon a detach or crash. While possible, this requires two new hardware supports: a reliable crash detector, and spare energy to complete encryption post-crash. Instead, our approach keeps the shadow in plaintext (accessible to loads/stores) while the primary is in ciphertext (so no special hardware for post-crash processing is needed). The cost is minor: a psync involves encrypting the shadow into the primary instead of copying it.

To elaborate on our design, Figure 2 illustrates the state transitions required to support DRAM caching. The overall state transition diagram is provided in (a). A LPMO is initially in the Ⓓ (detached) state. Upon an `attach()` with read (r) or write (w) permissions, the LPMO transitions Ⓓ → Ⓦ (write) or Ⓓ → Ⓡ (read) state. Upon a page fault (b), the

LPMO transitions → $\textcircled{Dc}$ (decrypt and copy to DRAM). In this state, the kernel decrypts and copies a faulted LPMO page into a DRAM cache and verifies the checksum of the page (if enabled). If the decrypted checksum and the expected checksum do not match, the kernel causes the process to generate a segmentation fault.

A psync call (c) on a LPMO in a $\textcircled{R}$ (read) state causes nothing to happen, but a psync call on a LPMO in a $\textcircled{W}$ (write) state causes the PMO to transition → $\textcircled{EP}$ (encrypt and persist), where all pages in DRAM that are marked as dirty are encrypted and then copied and persisted (via `memcpy_flushcache()`) into a temporary shadow copy (TSC) that exists in the PM. The LPMO then transitions → $\textcircled{Tc}$ (TSC copy). These pages are then copied from the TSC into the primary, ensuring that the data are always in a crash consistent state. Compared to prior work, psync is now slightly slower since two copies are performed, rather than a persist followed by a copy, but this should be offset by the fact that DRAM is significantly faster than PM.

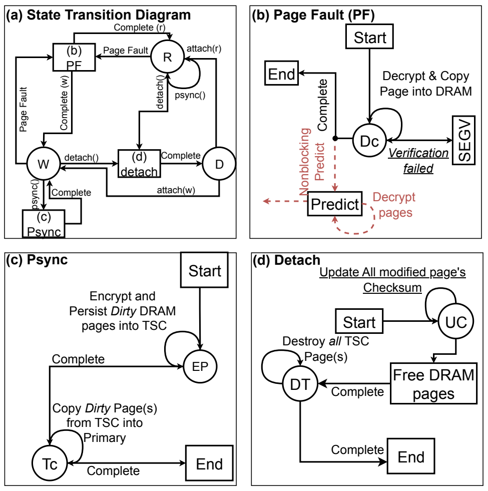


Figure 2: LPMO state diagram.

Finally, on detach (d), the LPMO transitions → $\textcircled{UC}$ (Update Checksum), which, if Integrity Verification is enabled, updates the checksum of those pages that have been modified. The kernel then frees the DRAM pages allocated for the PMO, the PMO then transitions → $\textcircled{DT}$ (Destroy TSC), the kernel erases the TSC pages, and the call returns.

4.2 Page Access Prediction

While our DRAM shadow paging helps performance, accessing the page for the first time still requires a page fault (to place the shadow page into DRAM), decryption, and integrity verification. Together, these incur a substantial critical-path delay that places an upper-bound on performance improvements. To tackle such delays, we propose predicting access patterns, decrypting them into DRAM, and verifying their integrity, ahead of time.

For this to work, several things must be achieved simultaneously. First, the predictor itself is software that runs in the kernel and occupies CPU time, hence it must be lightweight and fast in generating predictions. At the same time, the predictor must achieve high **accuracy** (what percentage of pages decrypted ahead of time are later found to be correct) to avoid excessive PM read and DRAM write bandwidth consumption and CPU occupancy. Second, the predictor must achieve high **coverage** (what percentage of accessed pages were decrypted ahead of time) to produce meaningful performance improvements. Finally, the predictor must exhibit high **timeliness** (what percentage of predicted pages that were fully decrypted by the time the page fault occurs) to actually be beneficial.

To satisfy all of these requirements, we use a page-usage pattern predictor based on stream buffers [33]. Stream buffer predictors are lightweight and can achieve sufficiently high coverage, accuracy, and timeliness. We try several different prediction depths to evaluate all three properties.

Returning to Figure 2, the dashed lines are the added component for handling page prediction. When a page fault occurs, the kernel checks whether the page is already decrypted from a previous prediction, and if so, skips the decryption and copy step. In either case, the prediction handler (which is nonblocking) is invoked. The prediction handler predicts the next X sequential pages that are likely to be accessed next, then runs X number of decrypt, verify, and copy operations on X number of pages ahead of the faulting page, depending on its depth. We refer to X as the **stream depth**.

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 (by calling `remap_pfn_range()`)

4.3 Page Access Predictor Evaluation

To determine how effective our stream buffer design is, we evaluate our predictor based on different properties (accuracy, coverage, timeliness) and then analyze the access pattern. Figure 3 illustrates the properties of our predictor for four benchmarks. Assuming decryption on page fault in software (as done by LOaPP's authors [14]) and denoted by S, we see that 2d Convolution, LU Decomposition, and Gaussian Elimination have good accuracy, coverage, and timeliness (although LU's accuracy drops as the size of the stream increases). Interestingly, although Tiled Matrix Multiplication has good coverage and timeliness when using stream buffers

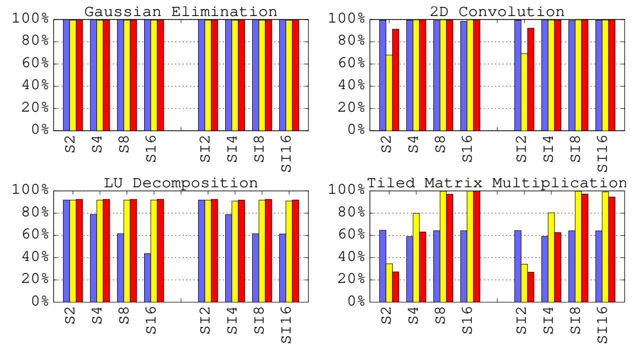


Figure 3: Prediction Accuracy, Coverage, and Timeliness for the four microbenchmarks with (*SI*) and without (*S*) integrity verification.

of sizes 8 and 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.

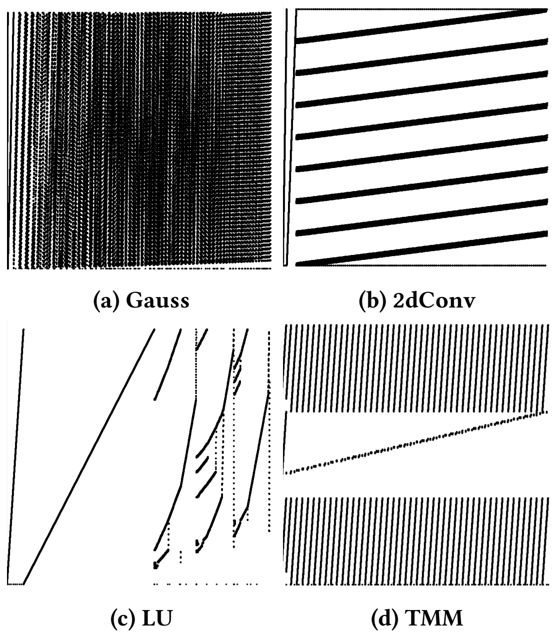
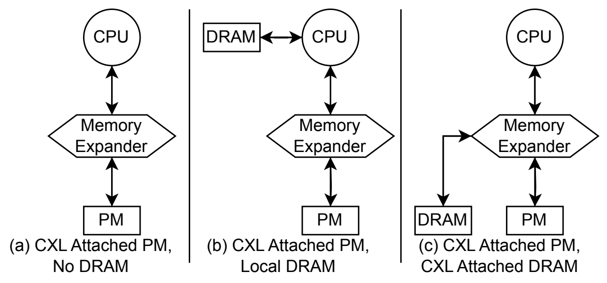


Figure 4: Access patterns.

To determine whether these characteristics are expected for these workloads, Figure 4 shows the overall access patterns for each. Each gray dot represents a page fault. 2d Convolution, Gaussian Elimination, LU Decomposition all have access patterns that stream buffers catch. Tiled Matrix Multiplication however suffers in accuracy regardless of the stream depth size.

5 LPMO WITH CXL



CPU
Memory Expander
PM
(a) CXL Attached PM, No DRAM
DRAM
CPU
Memory Expander
PM
(b) CXL Attached PM, Local DRAM
CPU
Memory Expander
DRAM
PM
(c) CXL Attached PM, CXL Attached DRAM

Figure 5: (a): Design C, (b): Design CL, (c): Design CF.

We envision that the LPMO abstraction can be applied both to local and CXL-attached PM systems, enabling new memory hierarchies such as those shown in Figure 5. Figure 5(a) depicts a system with PM attached to the CXL memory expander and no DRAM cache. Figure 5(b) depicts a LPMO system with attached PM and a DRAM cache that is local to the processor, and not associated with CXL. The final Figure 5(c) depicts a LPMO system with both DRAM and PM attached to the CXL Memory Expander. This paper refers to the design choices depicted in Fig. 5(a), (b), and (c) as **Design C**, **Design CL**, and **Design CF**, respectively. Table 1 summarizes the different advantages and disadvantages of the three designs depicted in Figure 5.

5.1 Reconfigurable Memory Hierarchy

Table 1: CXL Architecture design trade-offs.

Design	Advantages	Disadvantages
C	Faster psync Simple to implement	CXL latency Slow write speed Poor multithreading
CL	Symmetric read/write No additional CXL latency	Uses local DRAM Limited cache size Psync slower
CF	Symmetric read/write Flexible memory config	CXL latency Psync slower

5.1.1 ***Design C.*** Pond [25] demonstrated that CXL adds ≈ 75ns relative to local DRAM, due to the overhead from the external memory controller. In the first design, the CXL-attached PM will therefore have an additional latency of ≈ 75ns, and all reads/writes to any data within a PMO will incur at least that latency, along with the latency of the PM fabric. In addition, since PM has asymmetric read/writes, write speed to the PM fabric will be even further reduced. Finally, multithreading performance will also be poor [20]. The main benefit of this design is that there is no need for additional memory devices to be used as a cache (e.g., DRAM) to handle the LPMO system, which makes it simple to implement. Psync is also slightly faster under this system than the other design choices.

5.1.2 ***Design CL***. In this design, the CXL-attached PM will retain the extra $\approx$ 75ns latency upon the first fault, but read/write to PMO data after the page has been faulted in will be significantly faster since the DRAM used as a cache with the PMO system *will be on the local node*. The main advantage of this design is that read/write operations on cached pages will experience no further CXL access latency, and there will be symmetric read/write performance to PMOs after the first fault. The main disadvantages are that 1) psync will be somewhat slower, since it will require two copies to PM rather than just a persist followed by the copy, 2) this will use the same pool of local DRAM as other components of the system, and 3) since the size of PM on the memory expander might be significantly larger than the size of available local DRAM, the amount of PMO data that is cacheable might be a much smaller fraction of the total size of the LPMO system.

5.1.3 ***Design CF***. In this design, the CXL-attached PM and DRAM share a memory expander. The added latency will therefore be $\approx$ 75 ns for both PM and DRAM. This design splits the difference between designs **C** and **CL**: DRAM will provide symmetric read/write latencies, and we can include significantly higher amounts of DRAM than what can be included locally, but the trade-off is that the CXL access latency is still higher than the local access latency.

5.1.4 Discussion. There could be a possible fourth design **Design CA**, that we do not evaluate, but we mention for completeness and as consideration for future work. This design would use a CXL Type 2 Accelerator device instead of a Type 3 Memory Expander. The device could have its own local memories and directly cache data from PM without relying on the host.

5.1.5 ***Psync***. As mentioned above, designs **CL** and **CF** introduce a minor complexity in handling synchronization. Since the working copy is in DRAM and not in the shadow PM page, `psync()` takes slightly longer to complete, since the persistent step is now a copy (from DRAM into the shadow PM page) followed by another copy (from DRAM into the primary PM page).

5.1.6 ***Memory Tiering***. The ability to reconfigure memory means that it is possible to configure it to use hierarchal or flat memory ad hoc, depending on the needs of the programmer. It is also important to note that, unlike prior software-managed tiering approaches [11, 27, 44], LPMOs do not rely on tracking memory hotness. Instead, it employs a one-to-one mapping of PM to the designated cache, under the assumption that the cache size matches the PM size. This design minimizes overhead on the host CPU, making it lightweight and efficient; we leave software-based hotness tracking and fine-grained data placement in the context of LPMOs as future work.

5.2 Hardware Support for Memory Security

5.2.1 Memory Encryption Design considerations. As mentioned in Section 2, the CXL TEE [6] specification describes two different types of at-rest memory encryption for CXL Memory. The first type of at-rest memory encryption is *CKID-based Encryption*, the second type is *Range-based Encryption*.

CKID-based Encryption. CKID-based encryption uses the CKID field within the CXL Transaction Layer for the CXL Memory protocol (CXL.mem). The CKID field is used to identify a specific key to be utilized for decrypting or encrypting the contents of a given M2S memory transaction. The maximum number of CKID keys envisioned by the CXL Specification is 2^{13}, or 8192 keys, as specified in the "Number of CKIDs" field reported by "Get Target Capabilities Response", per the CXL TSP specification. This would mean that the LPMO system could only support up to 8192 PMOs without having to reuse CKID keys, which is clearly too small; a device hosting a PMO system could easily exhaust this limit.

Range-based Encryption. Range-based encryption uses memory range registers that are configured by the device to associate a specific encryption key with a specific host physical address range. Read/write operations to that memory range use that encryption key, and there is no need for the memory transaction to explicitly invoke it: it is done automatically. The maximum range ID the CXL system can support is 2^{16}, or 65000 keys. This would mean that the system would only support up to 65000 PMOs. This is a significantly larger number compared to CKID encryption. In our design, we use range-based encryption.

Encryption Standards. According to the CXL specification, CXL devices may support two different types of encryption standards [6], AES-XTS-128 and AES-XTS-256. Although the specification states that the device can use alternative algorithms, only these two are explicitly supported. However, like with CKID and Range-based encryption, these two encryption standards are mutually exclusive; only one bit for the "Memory Encryption Algorithm Select" may be active at one time. Prior PMO implementations used AES-XTS-256 [15]; we assume the same in this work, but AES-XTS-128 could also be used for performance reasons.

5.2.2 Integrity Verification. As mentioned in Section 2.5.5, Integrity Verification (IV) of at-rest data is outside of the scope of the CXL TEE specification. Therefore, the LPMO system can have the kernel perform integrity verification at detach or integrity verification at psync. The implications of this were explored in prior work with PMOs [14]; the authors of that work argued that IV at detach was valid so long as the system could perform recovery in case of a crash or transient fault. In Section 8, we evaluate the performance impact of performing integrity verification in the CPU at detach time, while keeping at-rest encryption within the TEE.

6 IMPLEMENTATION

6.1 LPMO Local Memory Implementation

To handle local memory, each page within a LPMO 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, it maps the DRAM page into the appropriate userspace virtual addresses, otherwise it performs the PMO 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. The first time the page is either predicted or faulted, the kernel serves a page from DRAM via `__get_free_page()` [8].

To handle prediction, the kernel calls the prediction handler at page fault time. The prediction handler then copies the next X sequential pages (where X is the stream depth) to DRAM. If the atomic flag indicating that the page has already been handled is set, the prediction handler skips that page, otherwise it performs the copy to DRAM. A key challenge with prediction is ensuring that the prediction and page fault code do not cause data races by occurring at the same time. To do this, the kernel locks the page via a mutex, and future page faults on the page then wait until the decryption completes, and then skips the decryption step (by calling `remap_pfn_range()`).

6.2 CXL LPMO Implementation

To evaluate the performance of the LPMO design **C** and **CL**, we use `taskset` to host the Optane memory of our PM system on the opposite NUMA socket from the CPU, as per Pond [25]. To evaluate the performance of LPMO design **CF**, we mimic both DRAM and PM being hosted in CXL by forcing *all* memory allocations in DRAM related to PMO page faults to occur on the opposite socket. We do this by calling in the kernel `__alloc_pages_node()` and force it to always allocate on the opposite node from the node reported by `numa_mem_id()`, by using the Get Free Page (GFP) flag `__GFP_THISNODE`. Because our system only has two NUMA nodes, this always results in the far NUMA node's memory being accessed.

7 EVALUATION METHODOLOGY

Table 2: Configuration of the PM system used for evaluation.

Component	Specifications
MB	Supermicro X11DPi-NT
CPU	2×Intel Xeon Gold 6230 (20 cores)
Clock	2.1GHz (3.9GHz Boost)
Cache	L1: 32KiB; L2: 1MiB; L3: 27.5MiB
DRAM	4 × 32GiB DDR4 @ 2666MHz
PM	4 × 128GiB Intel Optane DIMM
OS	AlmaLinux 9.0; Linux 5.15.157

We evaluated LPMO using the system described in Table 2. We use the following microbenchmarks taken from prior PMO works [14], and originally from [12]: 2d Convolution (2dConv), LU Decomposition (LU), Gaussian Elimination (Gauss), and Tiled Matrix Multiplication (TMM); we also evaluate our work with Filebench [29] modified to support PMOs. In addition, we evaluate a modified version of Lightning Memory-Mapped Database (LMDB) [4], which is a key-value store that utilizes a B+Tree to store its data [16]; we utilize the Yahoo! Cloud Serving Benchmark (YCSB) [7] workloads with LMDB to provide a large set of different, non-regular access patterns.

Table 3 provides the configuration of the evaluated microbenchmarks, the write percentages for the Filebench workloads, and a description of the YCSB workloads.

8 EVALUATION

In this section, we want to evaluate our LPMO implementation. We are interested in the answers to two questions: 1) How much more performant is the LPMO design compared to the original design? 2) How do our LPMO optimizations improve the thread scalability and psync sensitivity of PMOs compared to the original design?

In this section, O refers to the PMO design from [14], D refers to the non-CXL LPMO design with DRAM. C refers to the CXL design without DRAM (see Figure 5a), L refers to the CXL design with local DRAM (see Figure 5b), F refers to the CXL design with far DRAM (see Figure 5c). Finally, the numbers after $D/L/F$ refer to the stream depth when using prediction (see Subsection 4.2).

Table 3: Evaluated Benchmarks.

Microbenchmarks		Filebench		YCSB LMDB			
Benchmark	**Configuration**	**Workload**	**Write %**	**Workload**	**Description**	**Workload**	**Description**
2dConv	256x1024	Fileserver	67% writes	A	Update Heavy	D	Read Latest
Gauss	18432	VarMail	50% writes	B	Read Mostly	E	Short Ranges
LU	6144	WebProxy	16% writes	C	Read Only	F	RMW Heavy
TMM	4096 16	WebServer	9% writes				

Figure 6 depicts the legend for our microbenchmark results. P Other represents the time spent performing psync that do not involve rendering the primary PMO copy crash consistent (i.e., invalidating the cache lines to render the shadow copy durable) while P Data represents the time spent performing psync that involve copying the data from the shadow copy to the primary. Detach is the time between when the detach is invoked and when the system call returns; since detach is a nonblocking system call, this is a very short time period. PF Overhead represents all time spent on servicing page faults (i.e., the time spent rendering the page available to the calling process). A Other is the time spent performing all attach operations that are not being blocked by detach, while A Stall is the time the Attach system call spends waiting for the detach thread to complete (since detach is nonblocking, but the detach operations must complete before attach can occur). Finally, Compute is all of the other time not captured by the other categories, primarily, this is the time spent performing computation.

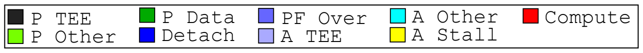


Figure 6: Microbenchmark Legend.

We note that that enabling encryption on a CXL device may have a slight performance cost that is not captured by our simulated system. However, this performance cost should be negligible: for example, the performance impact of Intel Total Memory Encryption-Multi Key (TME-MK) [19] is only ever at most 2.2%; we assume that CXL TEE encryption should be similar. To emulate this latency, we apply a 1% overhead to attach and psync when simulating LPMO on CXL (we refer to this as A TEE and P TEE).

8.1 LPMO Performance

8.1.1 Microbenchmark Results. Figure 7 compares the execution times of various microbenchmarks and their average for the state-of-the-art GPMO system (denoted as *O*, for origianal) from [14], versus our LPMO design with DRAM shadow paging (denoted as *D*), and our design with stream buffer prefetching with stream depths of 2, 4, and 8 (denoted as *D*2, *D*4, and *D*8). The execution times are all normalized to *O* for each benchmark, and are broken into various components.

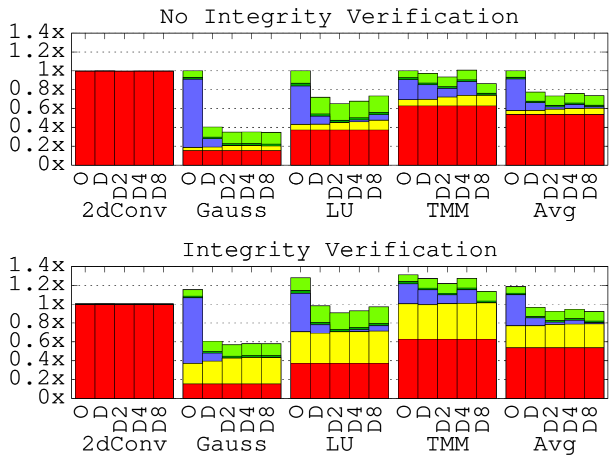


Figure 7: Execution time with and without DRAM prediction.

When comparing *O* vs. *D* our scheme reduces execution time by ≈ 21% on average, due to a large reduction in the page fault delay from much faster page faults because of DRAM's higher bandwidth and faster writes. Furthermore, it affects various benchmarks differently. For example, 2d Convolution is unaffected because it is compute heavy (consistent with prior works [14, 15]), while Gaussian Elimination is a write heavy benchmark that incurs page faults often, so its execution time decreases the most. Looking at page usage predictions, it further reduces page fault times across all benchmarks. Across various predictor stream depths, a shallow depth of 2 slightly outperforms other depths. Note that while the impact of page prediction is small, the result from Filebench, which uses sequential access patterns will show a much bigger impact (to be covered later). Finally, it is likely that if TMM had better accuracy (see Figure 3), its performance with prediction would be higher. It is notable that even with integrity verification, LPMOs are actually *faster* than the LOaPP design *without* integrity verification (*O*) (faster by 7.8% with a depth of 8). The performance improvement on average is 25% faster than the previous best-case design.

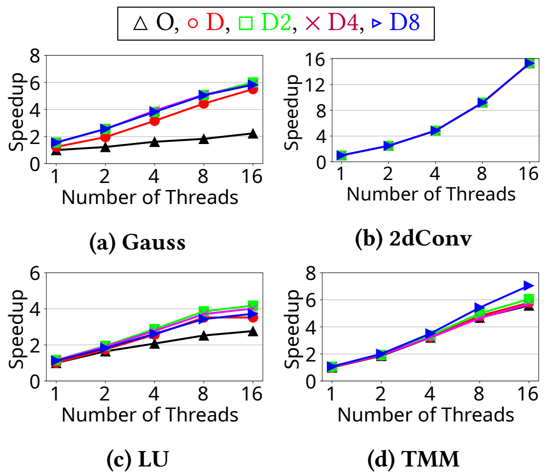


Figure 8: Thread speedup.

Thread scalability. Figure 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. The results show that excessive numbers of writer threads continue to slow the fabric down, but the effect is blunted somewhat with D, as the average performance improvement at 16 is 7.53× (compared to 6.47× with P). When using $S2$ or $S4$, the speedup increases to 7.89×. With that design, the benchmarks are on average 25% faster across all thread sizes, while they are only 16% faster when using D. Therefore, prediction improves scaling based on the number of threads. Figure 9 breaks down the results, normalized to P (single thread) for 2d Convolution, Gauss, LU, and TMM. 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 9 breaks down the results into different stream prediction sizes, from 2 to 8. It is clear that prediction greatly reduces the time spent waiting for page faults to complete.

Psync sensitivity. Figure 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 equal amount of attach/detach calls in each run, psync occurs more or less often. The results are normalized to D with Integrity Verification 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 2× psyncs per attach/detach, the most performant depth size has an improvement of 10% compared to D, but at 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 of time that psync consumes

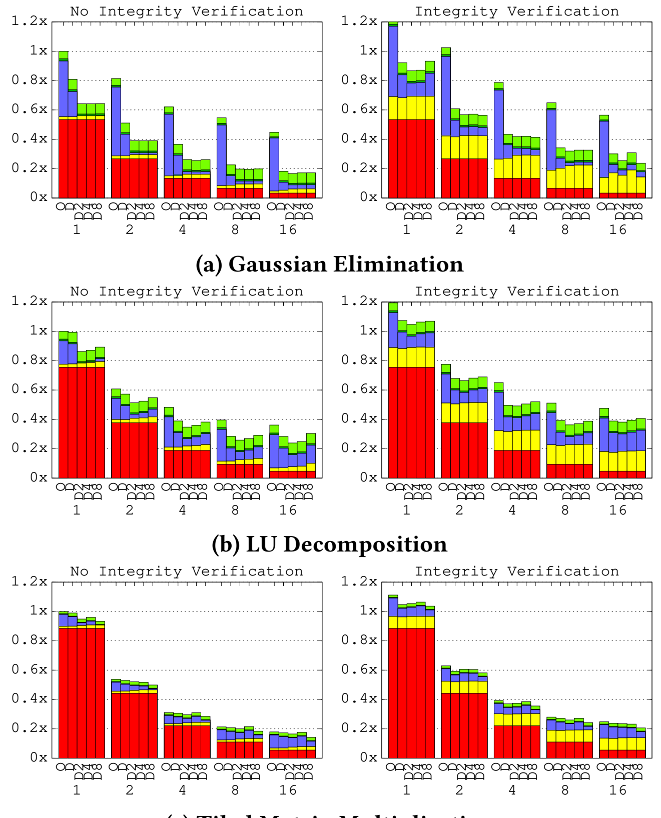
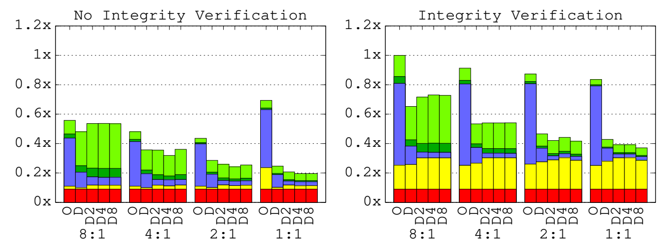


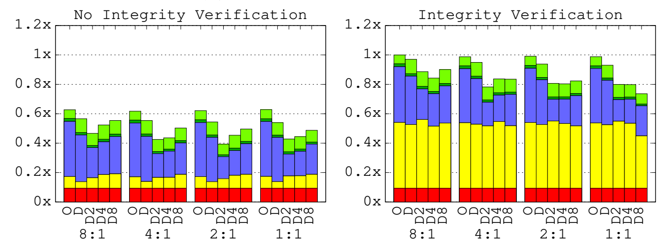
Figure 9: Thread scalability by stream prediction size.

as a total portion of the execution time. Although psync only persists those pages that are dirtied, in many cases, a large number of pages are dirtied between psync invocations. Also, using D with TMM produces a counter-intuitive result: there is worse performance at 8× psync invocation for D compared to O. This is because O encrypts the page from the shadow into the primary, while D 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 D compared to O alone. A programmer using D will therefore want to be mindful of the fact that psync is a more expensive operation and avoid invoking it excessively.

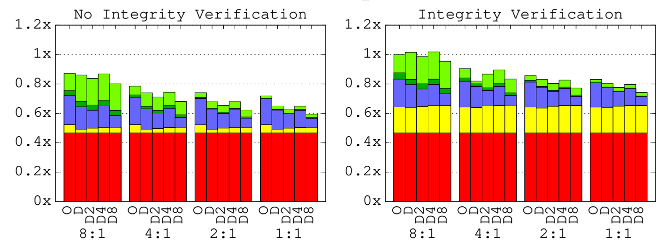
8.1.2 Filebench Results. Figure 11a compares the I/O bandwidth of different Filebench workloads, higher is better. The results are normalized to O and reported for 8 threads with synchronization performed after every write/append operation. The figure shows that D is 1.19× faster than O, but increases to 1.81× faster when page prediction is used. This result contrasts to the microbenchmarks' where the impact



(a) Gaussian Elimination

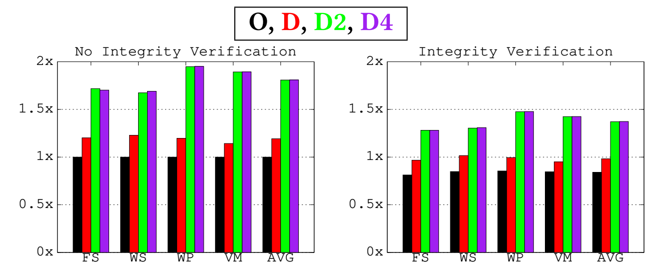


(b) LU Decomposition

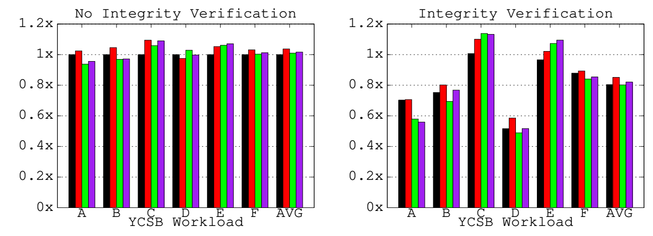


(c) Tiled Matrix Multiplication

Figure 10: Psync sensitivity by prediction depth.



(a) Normalized Filebench bandwidth.



(b) Normalized YCSB LMDB bandwidth.

Figure 11: Bandwidth for PM (*O*), DRAM (*D*), and Prediction (*D*2, *D*4).

of prediction on performance is smaller. With Integrity Verification, on average, *D* does not improve performance enough to make it faster than *O*, but prediction changes that: with prediction, it's 1.37× 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. In contrast to the microbenchmarks, the Filebench workloads have nearly 100% coverage, timeliness, and accuracy due to their workloads simulating real I/O access patterns, which are usually sequential [24]. Our PMO stream buffers perform well with these workloads.

8.1.3 LMDB YCSB Workloads. Figure 11b depicts the bandwidth of various YCSB workloads using an LMDB backend. As with the Filebench workloads, a higher bandwidth is better, and the results are normalized to *O*. The results are reported for 8 threads, and to represent a realistic persist cadence for key-value stores and read workloads, we call psync after every 10000 operations, and after 4 psync invocations, we invoke a detach and attach call.

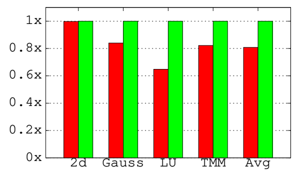
In contrast to the Filebench workloads, most of YCSB workloads do not benefit from the stream predictor. Without integrity verification, the predictor is about 1% slower than just using DRAM alone on average, and approximately 1.6% faster than the original design. With integrity verification, the predictor is $\approx$ 6% slower on average, and is negligibly faster (< 1%) than the original design.

However, Workloads *C* and *E*, when using Integrity Verification, present an exception to this. For these workloads, which are described as "Read Only" and "Short Ranges" respectively, the predictor is beneficial. For *C*, this is is because psync has nothing to do and there are no pages to have their checksum updated at detach time. For *E*, this is because the items accessed within the key-value store are nearby each other and so the pages the stream predictor decrypts ahead of time are almost always the ones needed by the workload. This is also why the predictor is faster at a depth of four than two. As the performance of the stream predictor is workload dependent, the system can disable the predictor once the miss-rate exceeds a specified value. We leave for future work the possibility of using a more complicated predictor.

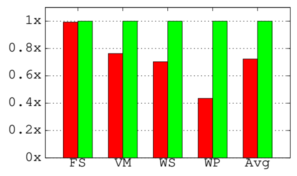
8.2 CXL Performance

In this section, we are interested in the implications of using CXL with our LPMO abstraction. We utilize our LPMO abstraction and compare it to the performance of the GPMO system.

8.2.1 Software Encryption. To illustrate the problem of software-based encryption with CXL, Figure 12 compares the total execution time at 8 threads of LPMO with software encryption (and no IV) to the execution times of LPMOs with CXL



(a) Microbenchmarks



(b) Filebench

Figure 12: Speedup (higher is better) of software encryption compared to CXL TEE.

TEE. As can be seen, LPMO devices utilizing software encryption is up to 56.5% slower (with WP) than without.

8.2.2 Microbenchmark Evaluations. In this subsection, we evaluate all of the microbenchmarks described earlier. First, we break down the execution of each of our microbenchmarks based on our memory configuration. The results are normalized to the original (*O*) GPMO design without integrity verification or CXL.

Gaussian Elimination. Figure 13a depicts the execution time of the Gaussian Elimination benchmark. This benchmark suffers from the added latency of CXL; it's nearly twice as slow as the original system. However, when we add DRAM to CXL, the trend reverses and the overall execution time is approximately 40% lower compared to the original design, because the page fault overhead is reduced (since only the first fault must access PM). As observed earlier, prediction does not improve the performance of the benchmark much, but our CXL design is nearly twice as slow as the local design, assuming we have DRAM. This is likely because Gaussian Elimination is a write heavy benchmark with a high number of page faults from shadow page allocations. The trend is the same when using Integrity Verification.

2d Convolution. Figure 13b depicts the execution time of the 2d Convolution benchmark. This benchmark is highly CPU bound [14] and so the overhead is relatively small. Nonetheless, there is very interesting behavior here. Adding integrity verification increases the execution time of the original PMO design by 23.5%, while it increases the execution time of the LPMO design with no DRAM by 20%. On the other hand, the LPMO design with EMFs are 3% faster, despite the added latency of CXL!

LU Decomposition. Figure 13c depicts the execution time of the LU Decomposition benchmark. In these benchmarks, no integrity verification is significantly faster. Like with the earlier benchmarks, we see that CXL is slower than the original, but adding a local DRAM cache makes it faster. When adding Integrity Verification, our CXL designs are slower than the original design, but with prediction, they are slightly faster depending on the depth. The page fault overhead (from faulting unnecessary pages) is very high for our CXL system at a depth of 8, which is surprising. It is likely because of the added latency of CXL NUMA nodes and the fact that LU is heavily memory bound.

Tiled Matrix Multiplication. Figure 13d depicts the execution time of the Tiled Matrix Multiplication benchmark. As before, the predictor does not change the performance much.

8.2.3 LMDB YCSB Workloads. Finally, we evaluate the same LMDB YCSB workloads we described in Section 8.1.3. Figure 14 normalizes our results to the original design both with and without integrity verification. The most interesting observation here is that despite the additional latency of CXL, the lack of software encryption/decryption means that the original design is almost always slower than even a CXL design without DRAM Caching (*C*). This means that for key-value stores, using our CXL Design improves performance dramatically compared to the original GPMO design. This behavior is true whether or not Integrity Verification is enabled.

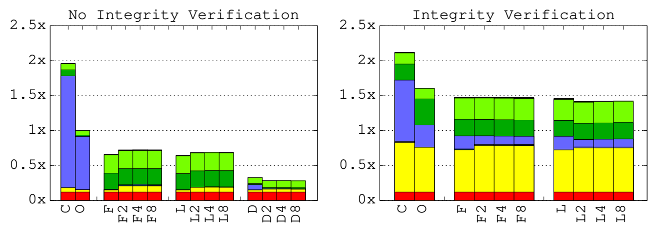
The second critical observation is that certain YCSB workloads (*A* and *D*) heavily punish the predictor when using integrity verification; Workload *A* is "Update Heavy", and Workload *D* are "Read Latest" workloads. It's important to note that Workload *D* does perform insertions, and the insertions are hashed rather than ordered purely by time (i.e., the "latest" items are not sequential, but are randomly placed within the B+Tree). Since the checksum is updated on detach on those pages that are marked as dirty, but the checksum is verified at page fault (which in our design, is verified ahead of time), the system's high misprediction rate of approximately $15-25\%$ leads to unnecessary integrity verification checks.

9 RELATED WORK

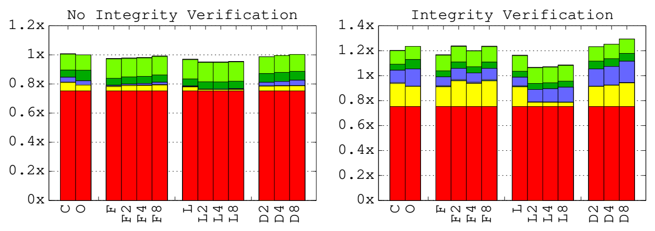
9.1 DRAM Caching for PM devices

HUNTER [34] proposes moving the metadata of PM File Systems to DRAM, to alleviate performance bottlenecks caused by updating PM metadata directly in the PM. As PMOs reduce metadata as much as possible by design, this optimization is not needed. HUNTER also utilizes dynamic arrays to perform fast indexing, but this optimization is again for metadata updates. Further, HUNTER's assurances of crash-consistency only apply to these metadata updates. On the other hand, our design, following the PMO approach to crash consistency, ensures crash-consistency for all data within the PM.

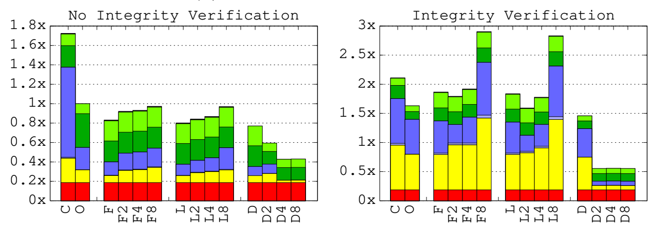
One possibility to avoid the requirement to perform psync to ensure crash-consistency is to utilize battery-backed RAM. In its presence, LPMO could abrogate performing psync entirely. However, Intel processors no longer support eADR [18] because of the discontinuation of Optane, making battery-backed RAM difficult to obtain.



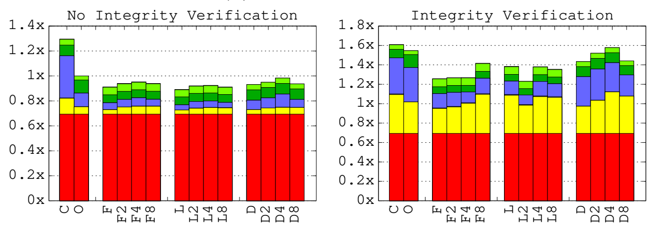
(a) Gaussian Elimination



(b) 2D Convolution



(c) LU Decomposition



(d) Tiled Matrix Multiplication

Figure 13: CXL Microbenchmark Evaluation

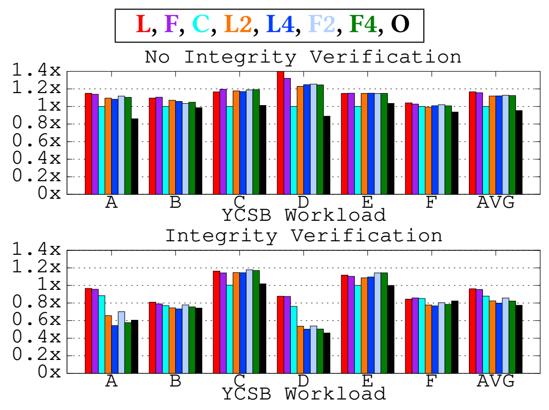


Figure 14: Normalized YCSB LMDB bandwidth.

9.2 CXL-attached PM devices

Prior work has discussed the implications of CXL-attached PM devices. For example [10] argues that the existence of CXL PM devices is not required for persistent memory research, because hybrid solutions that combine flash, volatile memory, and battery-backed buffers can enable persistent-memory systems. Another work, Memstrata [45] propose memory tiering modes for CXL devices and mentions how Optane PM can be configured to neglect the properties of crash consistency through the use of Memory Mode. Neither of these works, however, directly address byte-addressable persistent memory that treats both persistence and byte-addressability in equal importance; to the best of our knowledge, our design is the first to utilize CXL PM persistence.

10 CONCLUSION

We introduced LPMO, a lightweight PMO solution that enables software-based DRAM caching and predictive decryption, with the goal of further accelerating their performance without impacting security or reliability. Results show that compared to the prior most-performant design, these optimizations have the potential to improve performance by up to 1.25× (for DRAM caching alone) and 1.81× (with predictive decryption), depending on the workload. The performance impact of Integrity Verification is also lessened, and in some cases completely eliminated, when using DRAM and predictive encryption together.

We also demonstrated that our LPMO abstraction can apply to CXL-attached PM systems with a reconfigurable memory hierarchy, and we discussed the implications of such a design. We evaluated the same benchmarks and demonstrated that despite the added latency of CXL, reconfiguring the CXL architecture to support different memory configurations allows CXL attached devices using the LPMO abstraction to have performance comparable to, and sometimes faster than, the prior most-performant PMO design.

ACKNOWLEDGMENTS

This work was funded in part by multiple grants, including the National Science Foundation grants CNS-2314680 and CNS-2106629, as well as the Office of Naval Research grant N00014-23-1-2136. Furthermore, we greatly appreciate the invaluable feedback from the anonymous reviewers and shepherd. We also greatly appreciate the assistance of the other members of the CompArch and ARPERS research laboratories at the University of Central Florida.

REFERENCES

[1] David Boles, Daniel Waddington, and David A Roberts. 2023. CXL-enabled enhanced memory functions. *IEEE Micro* 43, 2 (2023), 58–65.

[2] Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. 2024. Intel TDX Demystified: A Top-Down Approach. *ACM Comput. Surv.* 56, 9, Article 238 (April 2024), 33 pages. https://doi.org/10.1145/3652597

[3] Kwanghoon Choi, Igjae Kim, Sunho Lee, and Jaehyuk Huh. 2025. ShieldCXL: A Practical Obliviousness Support with Sealed CXL Memory. *ACM Trans. Archit. Code Optim.* 22, 1, Article 13 (March 2025), 25 pages. https://doi.org/10.1145/3703354

[4] Howard Chu. 2011-2015. Lightning Memory-Mapped Database (LMDB). https://www.symas.com/lmdb.

[5] Kernel Development Community. 2023. Block Cipher Algorithm Definitions. https://www.kernel.org/doc/html/v5.14/crypto/api-skcipher.html#symmetric-key-cipher-api https://www.kernel.org/doc/html/v5.14/crypto/api-skcipher.html#symmetric-key-cipher-api.

[6] COMPUTE EXPRESS LINK CONSORTIUM, INC. 2023. *Compute Express Link 3.1 Specification.* COMPUTE EXPRESS LINK CONSORTIUM, INC. Revision 3.1.

[7] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[8] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. 2005. *Linux device drivers.* " O'Reilly Media, Inc.", Sebastopol, CA, USA.

[9] Debendra Das Sharma, Robert Blankenship, and Daniel Berger. 2024. An Introduction to the Compute Express Link (CXL) Interconnect. *Comput. Surveys* 56, 11 (2024), 1–37.

[10] Peter Desnoyers, Ian Adams, Tyler Estro, Anshul Gandhi, Geoff Kuenning, Mike Mesnier, Carl Waldspurger, Avani Wildani, and Erez Zadok. 2023. Persistent Memory Research in the Post-Optane Era. In *Proceedings of the 1st Workshop on Disruptive Memory Systems* (Koblenz, Germany) *(DIMES '23)*. Association for Computing Machinery, New York, NY, USA, 23–30. https://doi.org/10.1145/3609308.3625268

[11] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. 2023. Towards an Adaptable Systems Architecture for Memory Tiering at Warehouse-Scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Vancouver, BC, Canada) *(ASPLOS 2023, Vol. 3)*. Association for Computing Machinery, New York, NY, USA, 727–741. https://doi.org/10.1145/3582016.3582031

[12] Hussein Elnawawy, Mohammad Alshboul, James Tuck, and Yan Solihin. 2017. Efficient Checkpointing of Loop-Based Codes for Non-volatile Main Memory. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, Portland, Oregon, USA, 318–329. https://doi.org/10.1109/PACT.2017.58

[13] Donghyun Gouk, Miryeong Kwon, Hanyeoreum Bae, Sangwon Lee, and Myoungsoo Jung. 2023. Memory pooling with cxl. *IEEE Micro* 43, 2 (2023), 48–57.

[14] Derrick Greenspan, Naveed Ul Mustafa, Andres Delgado, Connor Bramham, Christopher Prats, Samu Wallace, Mark Heinrich, and Yan Solihin. 2024. LOaPP: Improving the Performance of Persistent Memory Objects via Low-Overhead at-Rest PMO Protection. In *2024 International Symposium on Secure and Private Execution Environment Design (SEED)*. IEEE, IEEE, Orlando, Florida, USA, 131–142.

[15] Derrick Greenspan, Naveed Ul Mustafa, Zoran Kolega, Mark Heinrich, and Yan Solihin. 2022. Improving the Security and Programmability of Persistent Memory Objects. In *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*. IEEE, Storrs, USA, 157–168. https://doi.org/10.1109/SEED55351.2022.00021

[16] Gavin Henry. 2019. Howard chu on lightning memory-mapped database. *Ieee Software* 36, 06 (2019), 83–87.

[17] Michael Henson and Stephen Taylor. 2014. Memory encryption: A survey of existing techniques. *ACM Computing Surveys (CSUR)* 46, 4 (2014), 1–26. Publisher: ACM New York, NY, USA.

[18] Intel. 2021. eADR: New Opportunities for Persistent Memory Applications. https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html

[19] Intel. 2023. Intel® Total Memory Encryption - Multi-Key - 009 - ID:655258 | 12th Generation Intel® Core™ Processors.

[20] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. arXiv:1903.05714 [cs.DC] https://arxiv.org/abs/1903.05714

[21] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. 2019. SplitFS: reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 494–508. https://doi.org/10.1145/3341301.3359631

[22] David Kaplan, Jeremy Powell, and Tom Woller. 2016. AMD memory encryption. *White paper* 13 (2016), 12.

[23] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 207–220. https://doi.org/10.1145/1629575.1629596

[24] Chuanpeng Li, Kai Shen, and Athanasios E. Papathanasiou. 2007. Competitive prefetching for concurrent sequential I/O. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (Lisbon, Portugal) *(EuroSys '07)*. Association for Computing Machinery, New York, NY, USA, 189–202. https://doi.org/10.1145/1272996.1273017

[25] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. 2023. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. ACM New York, New York, USA, Vancouver, British Columbia, Canada, 574–587.

[26] Jubayer Mahmod and Matthew Hicks. 2022. Sram has no chill: exploiting power domain separation to steal on-chip secrets. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Lausanne, Switzerland, 1043–1055.

[27] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent

Page Placement for CXL-Enabled Tiered-Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 742–755. https://doi.org/10.1145/3582016.3582063

[28] Tony Mason, Thaleia Dimitra Doudali, Margo Seltzer, and Ada Gavrilovska. 2020. Unexpected performance of Intel® Optane™ DC persistent memory. *IEEE Computer Architecture Letters* 19, 1 (2020), 55–58.

[29] Richard McDougall and Jim Mauro. 2005. FileBench Tutorial, In 2024 NFS Conference. *URL: http://www. nfsv4bat. org/Documents/nasconf/2004/filebench. pdf*, 56.

[30] Sebasian Moss. 2022. Intel kills off Optane Memory, writes off $559 million inventory. https://www.datacenterdynamics.com/en/news/intel-kills-off-optane-memory-writes-off-559-million-inventory/.

[31] Naveed Ul Mustafa and Yan Solihin. 2023. Persistent Memory Security Threats to Interprocess Isolation. *IEEE Micro* 43, 5 (2023), 16–23.

[32] Naveed Ul Mustafa, Yuanchao Xu, Xipeng Shen, and Yan Solihin. 2021. Seeds of SEED: New Security Challenges for Persistent Memory. In *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*. IEEE, IEEE, Virtual, 83–88.

[33] S. Palacharla and R. E. Kessler. 1994. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21st Annual International Symposium on Computer Architecture* (Chicago, Illinois, USA) *(ISCA '94)*. IEEE Computer Society Press, Washington, DC, USA, 24–33. https://doi.org/10.1145/191995.192014

[34] Yanqi Pan, Yifeng Zhang, Wen Xia, Xiangyu Zou, and Cai Deng. 2025. HUNTER: Releasing Persistent Memory Write Performance with A Novel PM-DRAM Collaboration Architecture. In *Proceedings of the 60th Annual ACM/IEEE Design Automation Conference* (San Francisco, California, United States) *(DAC '23)*. IEEE Press, New York, NY, USA, 1–6. https://doi.org/10.1109/DAC56929.2023.10247940

[35] P Roberts. 2003. MIT: Discarded hard drives yield private info. https://www.computerworld.com/article/1334164/mit-discarded-hard-drives-yield-private-info.html

[36] Brian Rogers, Yan Solihin, and Milos Prvulovic. 2005. Memory predecryption: hiding the latency overhead of memory encryption. *ACM SIGARCH Computer Architecture News* 33, 1 (2005), 27–33.

[37] Yan Solihin. 2019. Persistent memory: Abstractions, abstractions, and abstractions. *IEEE Micro* 39, 1 (2019), 65–66.

[38] Yan Solihin, Jaejin Lee, and Josep Torrellas. 2002. Using a user-level memory thread for correlation prefetching. *ACM SIGARCH Computer Architecture News* 30, 2 (2002), 171–182.

[39] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. 2023. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto, ON, Canada) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 105–121. https://doi.org/10.1145/3613424.3614256

[40] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. 2017. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 478–496. https://doi.org/10.1145/3132747.3132761

[41] Yuanchao Xu, Yan Solihin, and Xipeng Shen. 2020. MERR: Improving Security of Persistent Memory Objects via Efficient Memory Exposure Reduction and Randomization. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 987–1000. https://doi.org/10.1145/3373376.3378492

[42] Yuanchao Xu, Chencheng Ye, Xipeng Shen, and Yan Solihin. 2022. Temporal Exposure Reduction Protection for Persistent Memory. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, IEEE, Seoul, South Korea, 908–924.

[43] Yuanchao Xu, ChenCheng Ye, Yan Solihin, and Xipeng Shen. 2020. Hardware-Based Domain Virtualization for Intra-Process Isolation of Persistent Memory Objects. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. ACM,IEEE, Virtual, 680–692. https://doi.org/10.1109/ISCA45697.2020.00062

[44] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Nimble Page Management for Tiered Memory Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 331–345. https://doi.org/10.1145/3297858.3304024

[45] Yuhong Zhong, Daniel S. Berger, Carl Waldspurger, Ryan Wee, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D. Hill, Mosharaf Chowdhury, and Asaf Cidon. 2024. Managing Memory Tiers with CXL in Virtualized Environments. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 37–56. https://www.usenix.org/conference/osdi24/presentation/zhong-yuhong