Improving the Security and Programmability of Persistent Memory Objects

Derrick Greenspan, Naveed Ul Mustafa, Zoran Kolega, Mark Heinrich, Yan Solihin
University of Central Florida
{derrick.greenspan, unknown.naveedulmustafa, heinrich, yan.solihin}@ucf.edu
kolegazoran@knights.ucf.edu

***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. This paper addresses three problems with PM for the first time. First, we design a *persistent memory object* (PMO) abstraction that allows data to be retained in memory across process lifetimes and system power cycles. Second, we address the security of PMOs while at rest against corruption and disclosure attacks. Third, we address 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. Our PMO design outperforms NOVA-Fortis, a memory-mapped file-based approach providing crash consistency, by $3.61\times$ and $3.2\times$ for two sets of evaluated workloads. Adding protection for at-rest data to the design incurs a modest overhead, between $3-46\%$, depending on the level of protection.**

I. 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 [4], [5], [11], [23]. 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 data. Alternatively, the *persistent memory object* (PMO) approach [2], [12], [24] 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 the attacker in one run becomes persistent, enabling it to affect future runs of the same, or even different, applications [20]. 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.

In this work, we address some of the security and programmability challenges of PMOs. We begin with a discussion on the lifespan of a PMO, which can be conceptualized as a combination of two distinct periods: *In-use*, when a PMO is mapped (or attached) into the address space of a user process and accessible to it, and *At-rest*: when a PMO is not mapped to the address space of any process i.e. 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 [24], [25], [26]. However, there have been no defenses proposed for at-rest PMOs. As with files, we expect that a typical PMO spends most of its life at-rest. While at-rest, it is vulnerable to disclosure or corruption that may be caused by the system software. To protect against this, we propose 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 work also addresses the programmability challenge of achieving crash consistency. Prior work with PMOs, such as MERR [24] and Twizzler [2] do not address crash consistency, and assumes 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) [18], or through transactions in Mosiqs [12]. 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 [16].

Hence, we propose a different approach that does not rely on transactions to establish crash consistency. Instead, our approach introduces *psync*, a system call that provides a simple primitive to the programmer to achieve crash consistency. Psync decouples *when* data in a PMO reaches a crash consistent state from *how* the state is achieved. The programmer specifies points in the code where data are in a crash consistent state, e.g., when a node has been inserted into a linked list, a tree has been balanced, etc. The system ensures that all stores prior to the psync are rendered durable prior to any store subsequent to the psync. The system hides the mechanism for achieving crash consistency from the programmer. In addition, psync is different from a transaction in that it is object-specific rather than thread-specific, e.g. stores to non-PMO data (or to a different PMO) are not governed by psync: they can be performed ahead of psync completion. Furthermore, unlike transactions, atomicity is not guaranteed between two consecutive psyncs, thus no abort and rollback are needed.

Finally, this work presents the design and implementation of an actual PMO abstraction on a real Linux kernel with Intel Optane PMem. The following properties guide our design:

1) **Secure:** An abstraction should protect not only PMOs *in-use*, but also PMOs *at-rest*.
2) **Crash consistent**: To be broadly applicable, the PM abstraction itself should provide a simple and intuitive method for programmers to manage crash consistency.
3) **Simple and efficient**: The programmer should not be expected to rewrite their applications in a major way to benefit from the PM data abstraction. Proposed primitives should be implemented efficiently.

```
1 struct node *c = attach(head, 'w');
2 while(c->next != NULL && c->data < data) c = c->next;
3 if(c->next == NULL) c->next = new_node;
4 else{tmp = c->next; c->next = new_node;
5     c->next->next = tmp;}
6 psync(head);
7 detach(head);
```

Listing 1: Linked list node insertion using PMOs.

Listing 1 illustrates how a programmer may use our PMO abstraction. The programmer first attaches a PMO as they would map a file, and then inserts a psync call on line 6, when data have reached a consistent point, e.g. after a node is fully inserted. The programmer may also combine N operations (insertions, deletions, etc.) together before calling psync to reduce overhead. Multiple threads within the same process may read or write to a PMO as it would any other shared memory address, but psync is process-wide; hence the programmer should invoke it after threads synchronize.

Overall, this paper makes the following **contributions**:

- We identify a new threat model for PMOs and present an example security attack for exploiting PMOs at-rest.
- We present our defense mechanism against attacks on PMOs at-rest through encryption and integrity verification.
- To support crash consistency, we introduce the *psync* primitive, which relieves the programmer from the burden of relying on transactions or using low-level flushing and fencing to manage persistency.
- We propose a new PMO system abstraction, and implement it in the Linux kernel.
- We evaluate the PMO system using a set of microbenchmarks and workloads from FileBench [19]. Our 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, our approach is $1.6\times$ and $3.2\times$ faster for microbenchmarks and FileBench, respectively. We also evaluate our secure at-rest design against an insecure baseline and found that it incurs a tolerable $3-46\%$ performance overhead, depending on the level of security enabled, for the executed workloads.

This paper is organized as follows: Section II discusses background knowledge. Section III discusses the threat model that our PMO encryption and integrity verification defends against. Section IV discusses our PMO design. Section V discusses our extensions to the design to provide for security at-rest. Section VI discusses the kernel modifications required to implement PMOs. Section VII describes our evaluation methodology for our PMO system. Section VIII discusses our evaluation results. Finally, Section IX concludes the paper.

II. Background and Related Work

A. 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 [5], Libnvmmio [4], splitFS [11], and NOVA-Fortis [23]. Alternatively, PM may be used as a repository of *persistent memory objects* (PMOs). Examples include MERR [24], Twizzler [2], and Mosiqs [12]. Our 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 [24], Twizzler [2], Mosiqs [12], and TERP [25]. 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. [26] 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 paper.

B. 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 inserted 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 [2], [12], [24], [25] 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, whom 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.

C. 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 [21], 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) [24] 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.

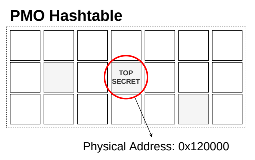
A relative pointer is a combination of a PMO ID and offset in format of `object:offset`. It can use a regular 64-bit format or use a fat pointer format where a pointer is represented by multiple fields. To dereference a pointer, a translation table is used to translate the system-wide unique PMO ID to its base virtual address [22], 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 leverages PMDK. 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.

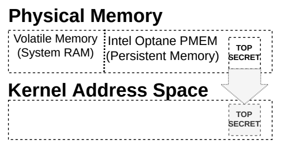
III. Threat Model

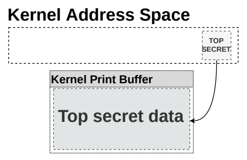
We consider a threat model where a PMO is not attached to any user process and so is not accessible in the user space (i.e., the PMO is at-rest). We assume that PMO-resident data-structures may contain buffers and pointers. We do not trust system software except for a subset, specifically the Linux Kernel Crypto API [14], critical kernel memory functions such as memcpy and memset, and our PMO kernel subsystem. We assume that all of these components are free of any code vulnerabilities. This assumption is reasonable, as the code-size of these components are small enough to be formally verified. For example, the Linux kernel Crypto API for version 5.14.18 contains about 82,500 source lines of code, our 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 entire kernel which contains about 2.2 million lines. That means our kernel subsystem, the critical memory functions (which are architecture specific, and written in assembly), and the Crypto API contribute to 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, we assume that the attacker knows the location and layout of a PMO in persistent memory. Our threat model is different from both [20], [24], since their threat model requires that the PMO be in-use by a user process before it can be exploited.



(a) Step 1: Discover PMO address.


Physical Memory
Volatile Memory (System RAM)
Intel Optane PMEM (Persistent Memory)
TOP SECRET
Kernel Address Space
TOP SECRET
(b) Step 2: Map physical memory into kernel space.



(c) Step 3: Read mapped data.

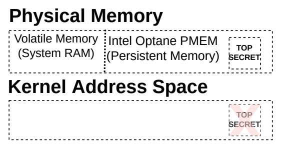

Physical Memory
Volatile Memory (System RAM)
Intel Optane PMEM (Persistent Memory)
TOP SECRET
Kernel Address Space
TOP SECRET
(d) Step 4: Unmap kernel space memory.

Fig. 1: Steps of PMO example attack.

Figure 1 illustrates an example PMO data-disclosure attack. We assume 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 Section IV-A). 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 [5]). 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 in Step 3 write and then persist invalid or malicious data into the PMO. If a user process attaches the PMO in the future, and relies on its data, this could potentially alter its control flow.

IV. Persistent Memory Object Design

We envision PMOs to be a contiguous region of memory. To achieve atomic crash consistent updates, there are generally two possible approaches: *logging* and *shadowing*. Logging (undo/redo) requires the creation and management of an undo/redo log, which is expensive to achieve at the OS level due to the need to intercept every first store (undo log) or every store (redo log). The kernel can only intercept a store by marking a page read only, such that the store incurs a trap to the kernel.

Therefore, we rely on 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, we create 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.

A. PMO System Layout

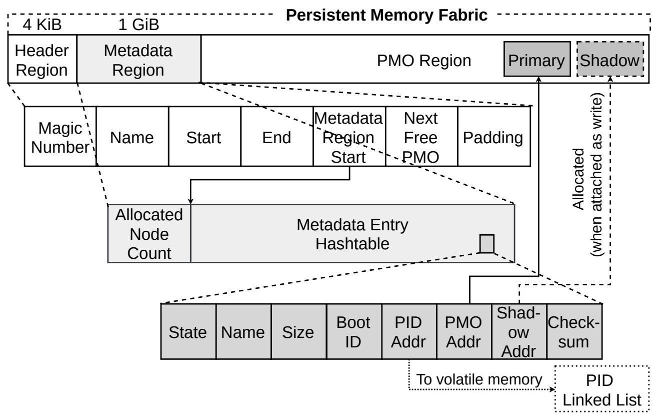


Fig. 2: The layout of a PMO system.

As shown in Figure 2, our 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, we use a 4KiB-size (1 page) header. 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). We keep the header mapped as uncacheable 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 Section IV-D2), 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[1], and the address of the PMO and its shadow copy (if any). The PID and Boot ID are used in combination to ensure that a PMO is attached to only one one process with write permissions at a time, both in normal operation and upon crash recovery, as described in Section IV-B2. The checksum field is used for PMO integrity verification as described in Section V. 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.

B. Programming Interface

TABLE I: 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.

1) pcreate/pdestroy: The **pcreate** primitive creates a PMO of a specified name, size, and key. Once created, a PMO exists in the PM until 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). 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

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

Free PMO" field is updated, and an entry is created in the PMO metadata hashtable.

2) attach: Successful invocation of the **attach** primitive maps a PMO (of specified name) into the virtual address (VA) space of the calling process, rendering it accessible to the process. We only allow 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)

Fig. 3: PMO inter-process sharing semantics.

Figure 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 linked-list of PIDs. A PID entry is added to the linked-list on successful attach and removed on detach. On an attach request for a PMO, a boot ID field (in the corresponding hashtable entry) which is different from the current 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.

3) detach: 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 VA has an invalid mapping to the physical address (PA) of the PMO. Successfully detaching a PMO attached as write sets the boot ID to sentinel value and removes PID 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.

4) psync: 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. We design psync to have similar semantics to the POSIX **msync** and **fsync** [9], 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 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.

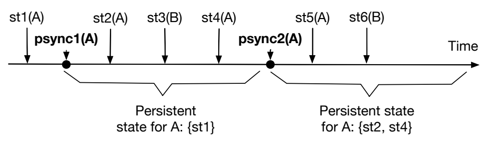

st1(A)
psync1(A)
st2(A)
st3(B)
st4(A)
psync2(A)
st5(A)
st6(B)
Time
Persistent state for A: {st1}
Persistent state for A: {st2, st4}

Fig. 4: psync transactional semantics.

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.

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 [23], can reduce the blocking interval as it blocks a thread only when it requests access to unmarked pages. For our current design, and for performance reasons, we expect the programmer to avoid this situation by synchronizing threads accessing the same PMO, although we intend to investigate and evaluate the two approaches in a future work.

C. 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:

1) 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, we advocate for an approach where a PMO is a contiguous region of memory with a static size set at the PMO's creation. 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.

2) 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 [24] 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 our PMO system must work with commodity hardware, we use a different solution, *demand paging* [8].

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 VA 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.

3) Low-Latency Pointer Dereferencing: A key challenge of PMO-resident (i.e. persistent) pointers is that the VA that a pointer refers to must be associated with the PA of a PMO beyond the process lifetime [1]. This is required to ensure that pointers within and across PMOs are always valid. As described in Section II, one solution to this challenge is to use *relative pointers* in `object:offset` format [3], 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-VA 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 [22]. While hardware supported pointer translation [22] 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, we propose to use *static pointers*. Static pointers are already in VA 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 VA. Therefore, we assign the VA range to PMOs at creation time such that no two addresses overlap. To achieve this, we use several techniques. First, to avoid an overlap between PMOs and non-persistent data, we split the effective virtual user space address space 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 VA x. The kernel maps a PMO into the persistent-half of the VA 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 VA range, we assign PMOs globally unique VAs.

With such an approach, we run the risk of running out of

VA space if there are too many large PMOs in a static VA allocation scheme. 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). We identify a possible mitigation strategy: we can use static pointers for small to medium PMOs (KiBs to MiBs), and use relative pointers for larger PMOs (GiBs and above).

Since the VA of a PMO is determined by its location in PM, our approach makes it costlier to perform the PMO address randomization used in MERR [24]; moving a PMO to a different VA 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, makes our approach attractive.

D. *Design for Crash-Consistency*

1) 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, our 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, we 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, we follow a different approach 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, our design allocates the size of the PMO again and designates it as the shadow copy. Our approach can potentially result in primary and shadow copies non-contiguous to each other. Therefore, we track the shadow copy through the "PMO Shadow Addr" entry pointing to the location of shadow (see Figure 2). A null entry indicates that the PMO is either detached or attached as read only.

2) PMO state transitions: To achieve crash consistent updates, a PMO is always in one of the five states shown in Figure 5 (consider only the solid/black and dashed/blue part). 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-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 Ⓓ → Ⓦ

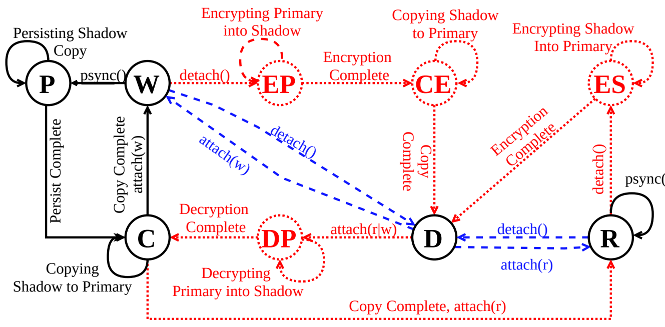


Fig. 5: PMO state transitions. Dashed are for the crash consistent design without encryption. Dotted are for the crash consistent design with encryption. Solid are for both.

(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 [7] 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.

3) 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 state of each PMO to determine which copy to rely on as consistent, illustrated in Figure 5. 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.

V. Security Protection for At-Rest PMO

In this section, we present a design that protects against the threat model and example attack described in Section III. The design provides a defense of at-rest PMOs against corruption and disclosure through integrity verification and encryption, respectively.

A. Protection From Corruption

To protect a PMO at-rest from corruption, we rely on a checksum computed over the PMO at detach time, and durably store it in the checksum field of the corresponding hashtable entry (Figure 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. To address this challenge, we identify two optimizations. First, for detach, we can compute the checksum 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. We leave solving the problem of verification-latency for future work.

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.

B. Protection From Disclosure

Recall that in Section III, we explained that we consider the Kernel Crypto API, certain kernel memory routines such as memcpy and memset, and the PMO Subsystem to be 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, our crash consistency approach utilizes a shadowing approach, including when a PMO is attached for read-only access. Since we already employ a shadow copy to manage crash consistency, we repurpose the shadow copy 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 5 shows the state-transition diagram for our design (consider only the solid/black and dotted/red components). A PMO is initially in the $\textcircled{D}$ state. On an attach call, irrespective of the permissions, it transitions $\textcircled{D} \rightarrow \textcircled{DP}$ (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 $\textcircled{DP} \rightarrow \textcircled{C}$, 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 $\textcircled{R}$ or $\textcircled{W}$, depending on attach permissions.

In $\textcircled{R}$, psync is ignored, while the invocation of detach transitions $\textcircled{R} \rightarrow \textcircled{ES}$ (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 $\textcircled{R} \rightarrow \textcircled{D}$.

In $\textcircled{W}$, the invocation of psync triggers state transitions in the same way as in the basic crash consistent design. In case of invoking detach in $\textcircled{W}$, the PMO transitions $\textcircled{W} \rightarrow \textcircled{EP}$ (Encrypt Primary) where the kernel encrypts the primary copy to the shadow and persists it. Note that detach does not automatically persist modifications and we expect the programmer to 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 $\textcircled{EP} \rightarrow \textcircled{CE}$ (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 $\textcircled{CE} \rightarrow \textcircled{D}$. Note that in the $\textcircled{D}$ state, both the primary and shadow copy are encrypted, so it is safe to free 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. We apply this optimization for PMOs that are read-only, since psync does not occur.

VI. IMPLEMENTATION

A. Persistent Memory Provisioning

Intel Optane PMem supports namespaces which expose the memory as a logical device [15] with different modes. We provision the namespace in devdax mode, providing direct access (DAX) to the underlying PM [10]; this mode emits a *character device* (as opposed to a block device). Analogous to block devices being formatted with `mkfs`, we introduce a similar utility for formatting PMO systems, `mkpmo`, that zeroes a given namespace and writes the PMO header and metadata structures.

B. Kernel Modifications

We make several modifications to the Linux kernel, and add attach, detach, and psync as system calls. In addition, we modify the Linux per-process memory descriptor (`mm_struct`), and we modify the virtual address page fault handler to use demand paging for PMOs, adopting the `VM_SOFTDIRTY` flag [7] 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, we extend the existing `vm_area_struct` in the Linux kernel to include new fields needed only when mapping PMOs in the address space of a process, which we call a `vpm_area_struct`. To allow for quick access, we introduce a new red-black tree to `mm_struct` with `pmo_rb` as its root.

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

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

C. Attach/Detach and Psync

We organize the metadata about the PMO 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, we use SHA256, and XTS-AES-256 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.

VII. EVALUATION METHODOLOGY

A. Correctness and Crash Consistency

To verify that our implementation is crash consistent, we inserted `panic()` 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, we reattach the PMO and examine its content. We verified that the data from the previous psync are in the PMO, as expected. We also tested inserting `panic()` immediately before the persist stage. After restart, we verified that the data from before the psync are there at attach.

B. Performance Assessment

In order to test our PMO system performance, we compare it against two schemes. The baseline scheme has *no crash consistency* (NCC) and represents an ideal performance case that is functionally incorrect. For NCC, we use 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 we compare our scheme against is the state-of-the-art crash consistent filesystem, NOVA-Fortis [23], that employs snapshots to support crash consistency. We note that NOVA-Fortis only guarantees crash consistency of the file system, but does not guarantee crash consistency of application data. We compare our PMO system against NCC and NOVA-Fortis in terms of execution time and I/O bandwidth of specified workloads. We also evaluate the thread scalability and synchronization-rate sensitivities of our PMO system. Finally, we determine the overhead caused by adding integrity checking and/or encryption to the PMO system.

For our evaluation, we used the system described in Table II. We implemented our PMO system on a modified version of Linux Kernel v5.14.18; we used stock v5.14.18 for evaluating

TABLE II: System used for evaluation.

Component	Specifications
Motherboard	Dual socket Supermicro X11DPi-NT (w/ADR)
CPU	2×Intel Xeon Gold 6230, 20 cores, 40 threads
CPU Clock	2.1GHz (3.9GHz Boost)
DRAM	4 × 32GiB DDR4 @ 2666MHz
PMEM	4 × 128GiB Intel Optane DC
Kernel	Linux 5.14.18 (PMO, NCC), 5.1.0 (NOVA)

NCC ext4 with dax. We use Linux Kernel v5.1.0 for NOVA-Fortis, as it is the latest version NOVA-Fortis supports. The Linux distribution was Fedora 33.

C. *Microbenchmarks*

In order to measure PMO performance from the perspective of *execution time*, we use an OpenMP version of LU decomposition provided by [13], (and originally from the SPLASH benchmark suite [17]), a 2D-Convolution (2dConv) benchmark and a Tiled Matrix Multiplication (TMM) benchmark, taken from a recent PM study [6]. We run LU, 2dConv and TMM with matrix sizes of 3584×3584 doubles, 4096×128 integers, and 3072×3072 integers, respectively.

We ported these benchmarks 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 Δ has elapsed from the previous invocation of the synchronization, we insert a synchronization point (i.e., psync in case of PMO, generating a snapshot in case of NOVA-Fortis, and invoking pmem_persist in case of NCC). Varying Δ varies the synchronization rate.

D. *Filebench*

We rely on FileBench benchmarks [19], which represent I/O intensive real-world applications, for measuring I/O bandwidth performance. We ported these benchmarks to use PMOs by replacing files with PMOs of respective sizes. For ext4-dax (i.e., NCC) and NOVA-Fortis we mapped files via DAX. Furthermore, we insert synchronization points in the benchmarks after **every** update (i.e., *append* or *wholefilewrite* flowop, in Filebench's terminology). Also, to avoid races between threads, we emit a pthread barrier 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%.

E. *Encryption and Memory Integrity*

In addition to testing our PMO system performance with crash consistency, we also attempt to determine the overhead compared to a baseline crash consistent design when adding encryption and data integrity. As with the crash consistency evaluation above, we evaluate encryption and integrity with Filebench; and we do 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. Note that adding attach/detach calls between each operation makes the microbenchmarks too slow to obtain useful results, so we do not perform this evaluation for them.

VIII. Evaluation Results

For this section, we want to answer several questions: How much performance overhead does our PMO system incur in comparison with a non crash consistent (NCC) and a crash consistent system? 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 on performance? Since there are no existing object-based abstractions providing intrinsic support for crash consistency, we compare our PMO abstraction with NOVA-Fortis, a state-of-the-art file system.

A. *Performance Evaluation of Crash Consistency*

Figure 6 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, our PMO system slows down the execution time by only $\approx 27.8\%$ (geometric mean) vs. $\approx 55.1\%$ with NOVA-Fortis. This indicates that our PMO system is not much slower than NCC and is $\approx 1.61\times$ faster than NOVA-Fortis.

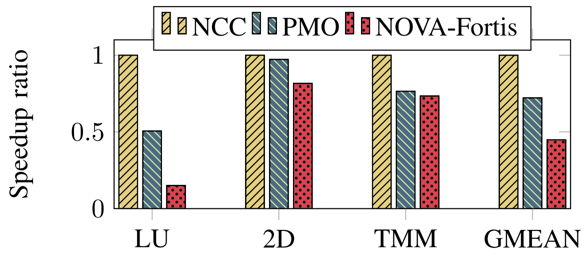


Fig. 6: Performance of crash consistent PMO and NOVA-Fortis normalized to Non-Crash-Consistent (NCC) ext4-dax.

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, our 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.

B. *I/O Bandwidth and Crash Consistency*

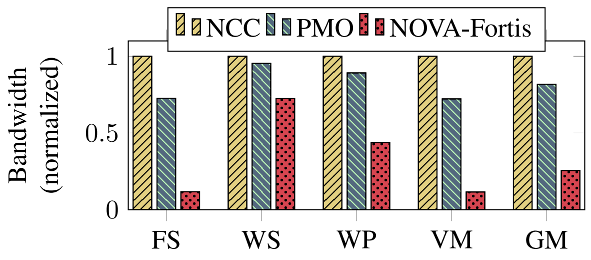


Fig. 7: Bandwidth comparison of crash consistent PMO and NOVA-Fortis to Non-Crash-Consistent (NCC) ext4-dax.

Figure 7 compares the I/O bandwidth of different FileBench workloads achieved by our 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 our 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 synchronization points in accordance with their write percentage. More frequent synchronization incurs more overhead and thus lower bandwidth performance.

C. *Thread-scalability*

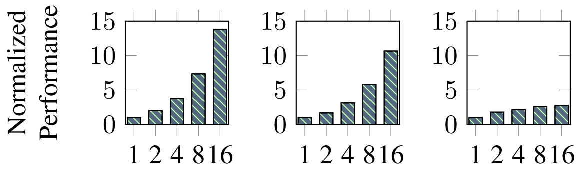


Fig. 8: Thread-scalability of the PMO system. From left to right: 2DConv, TMM, and LU.

Figure 8 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.

D. *Synchronization Frequency Sensitivity*

Figure 9 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

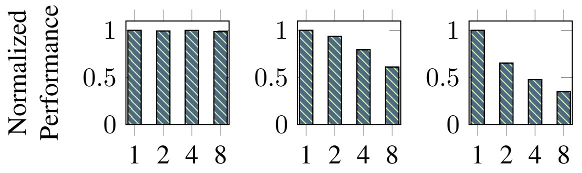


Fig. 9: Synchronization-sensitivity of the PMO system. From left to right: 2DConv, TMM, and LU.

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 8: more pages are updated per synchronization operation with LU and TMM.

E. *I/O Bandwidth of Encryption and Memory Integrity*

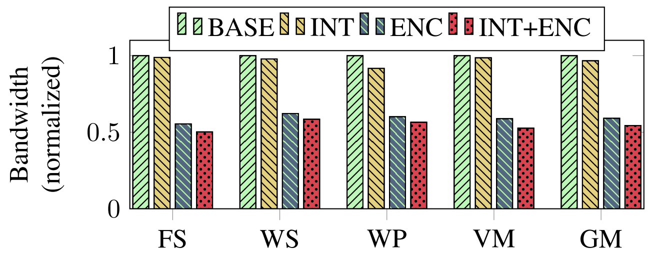


Fig. 10: Bandwidth comparison of attach/detach PMO, with different modes: baseline (BASE), Integrity (INT), Encryption (ENC), and both (ENC+INT).

Figure 10 shows the impact of our 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 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 both encryption/decryption affects both primary and shadow copies of a PMO, and is performed on the whole PMO. This result indicates that we should apply only what is needed, e.g. 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; we leave this for future work.

F. *Security Evaluation of Encryption and Memory Integrity*

To evaluate the strength of our approach to protect against at-rest PMO attacks, we design an experiment to detect whether our approach can prevent unauthorized disclosure and detect unauthorized modifications of at-rest PMO data. We initially create 1,000 PMOs and write a secret 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. We determine the effectiveness of our approach by calculating the ratio between the number of detected unauthorized modifications or unauthorized disclosures, divided by the total number of attacks.

We find that 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 our design protects against both at-rest disclosures and at-rest modifications of data.

IX. Conclusion

Security and programmabity are two important requirements for the design of a widely acceptable crash consistent object-based abstraction of persistent memory. We discussed the design and implementation of a secure persistent memory objects (PMO) system with intrinsic support for crash consistency. Results show that our 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.

Acknowledgements

This work is supported in part by the ONR under grant N00014-20-1-2750, and by the NSF under grant 1900724. We thank Dr. Paul Gazzillo, the reviewers, and our shepherd Dr. Michel Kinsy, for providing feedback to improve this paper.

References

[1] Alexandro Baldassin, João Baretto, Daniel Castro, and Paolo Romano. Persistent memory: A survey of programming support and implementations. 2021.

[2] Daniel Bittman, Peter Alvaro, Pankaj Mehra, Darrell DE Long, and Ethan L Miller. Twizzler: a data-centric {OS} for non-volatile memory. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, pages 65–80, 2020.

[3] Daniel Bittman, Peter Alvaro, and Ethan L Miller. A persistent problem: Managing pointers in nvm. In *Proceedings of the 10th Workshop on Programming Languages and Operating Systems*, pages 30–37, 2019.

[4] Jungsik Choi, Jaewan Hong, Youngjin Kwon, and Hwansoo Han. Libnvmmio reconstructing software {IO} path with failure-atomic memory-mapped interface. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 1–16, 2020.

[5] Linux Kernel Source Documentation. From https://kernel.org/.

[6] H. Elnawawy, M. Alshboul, J. Tuck, and Y. Solihin. Efficient checkpointing of loop-based codes for non-volatile main memory. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 318–329, 2017.

[7] Pavel Emelyanov. Soft-dirty ptes, Apr 2013. From https://kernel.org/.

[8] Mel Gorman. *Understanding the Linux virtual memory manager*. Prentice Hall Upper Saddle River, 2004.

[9] Austin Common Standards Revision Group. *IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7.* 2018.

[10] Intel. *Persistent Memory Programming*. August 2016.

[11] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. Splitfs: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 494–508, 2019.

[12] Awais Khan, Hyogi Sim, Sudharshan S Vazhkudai, Jinsuk Ma, Myeong-Hoon Oh, and Youngjae Kim. Persistent memory object storage and indexing for scientific computing. In *2020 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, pages 1–9. IEEE, 2020.

[13] Christian Klauser. Lu decomposition and matrix multiplication with openmp, 2011.

[14] James Morris. Kernel korner: the linux kernel cryptographic api. *Linux Journal*, 2003(108):10, 2003.

[15] Ivy B Peng, Maya B Gokhale, and Eric W Green. System evaluation of the intel optane byte-addressable nvm. In *Proceedings of the International Symposium on Memory Systems*, pages 304–315, 2019.

[16] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutiu. Thynvm: Enabling software-transparent crash consistency in persistent memory systems. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 672–685. IEEE, 2015.

[17] Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. Splash-3: A properly synchronized benchmark suite for contemporary research. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 101–111, 2016.

[18] Steve Scargall. *Programming persistent memory: A comprehensive guide for developers*. Springer Nature, 2020.

[19] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX; login*, 41(1):6–12, 2016.

[20] Naveed Ul Mustafa, Yuanchao Xu, Xipeng Shen, and Yan Solihin. 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.

[21] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 91–104, New York, NY, USA, 2011. ACM.

[22] Tiancong Wang, Sakthikumaran Sambasivam, Yan Solihin, and James Tuck. Hardware supported persistent object address translation. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 800–812. IEEE, 2017.

[23] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. Nova-fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 478–496, 2017.

[24] Yuanchao Xu, Yan Solihin, and Xipeng Shen. Merr: Improving security of persistent memory objects via efficient memory exposure reduction and randomization. New York, NY, USA, 2020. Association for Computing Machinery.

[25] Yuanchao Xu, Chencheng Ye, Xipeng Shen, and Yan Solihin. Temporal exposure reduction protection for persistent memory. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 908–924. IEEE, 2022.

[26] Yuanchao Xu, ChenCheng Ye, Yan Solihin, and Xipeng Shen. Hardware-based domain virtualization for intra-process isolation of persistent memory objects. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 680–692. IEEE, 2020.