DaxVM: Stressing the Limits of Memory as a File Interface

Chloe Alverti[*] Vasileios Karakostas[†] Nikhita Kunati[‡] Georgios Goumas[*] Michael Swift [§]
[*] *National Technical University of Athens* [†] *University of Athens*
[‡] *NVIDIA* [§] *University of Wisconsin-Madison*
{*xalverti, goumas*}*@cslab.ece.ntua.gr* *vkarakos@di.uoa.gr* *nikhitak@nvidia.com* *swift@cs.wisc.edu*

***Abstract*—Persistent memory (PMem) is a low-latency storage technology connected to the processor memory bus. The Direct Access (DAX) interface promises fast access to PMem, mapping it directly to processes' virtual address spaces. However, virtual memory operations (e.g., paging) limit its performance and scalability. Through an analysis of Linux/x86 memory mapping, we find that current systems fall short of what hardware can provide due to numerous software inefficiencies stemming from OS assumptions that memory mapping is for DRAM.**

In this paper we propose DaxVM, a design that extends the OS virtual memory and file system layers leveraging persistent memory attributes to provide a fast and scalable DAX-mmap interface. DaxVM eliminates paging costs through pre-populated file page tables, supports faster and scalable virtual address space management for ephemeral mappings, performs unmappings asynchronously, bypasses kernel-space dirty-page tracking support, and adopts asynchronous block pre-zeroing. We implement DaxVM in Linux and the ext4 file system targeting x86-64 architecture. DaxVM mmap achieves 4.9$\times$ higher throughput than default mmap for the Apache webserver and up to 1.5$\times$ better performance than read system calls. It provides similar benefits for text search. It also provides fast boot times and up to 2.95$\times$ better throughput than default mmap for PMem-optimized key-value stores running on a fragmented ext4 image. Despite designed for direct access to byte-addressable storage, various aspects of DaxVM are relevant for efficient access to other high performant storage mediums.

***Keywords*-virtual memory; persistent memory; file systems;**

I. Introduction

Persistent memory (PMem) is a new storage technology [43], [67] that is connected to the system via the memory bus, like DRAM, and is accessible via CPU load and store instructions. The technology uniquely combines four characteristics: (i) scaling capacities, (ii) byte-addressability, (iii) latency/bandwidth close to DRAM, and (iv) non-volatility, blurring the decades-old distinction between slow but persistent storage and fast but volatile memory.

With PMem, storage accesses can be cheaper than OS invocations, so reducing the OS overheads is a strong requirement. The DAX (Direct Access) interface [3] can map persistent memory directly to user-space, enabling applications to access storage via regular load/store instructions. Multiple works [7], [27], [36], [47], [54], [58], [76], [79], [80], [81], [82] attempt to reduce PMem software stack overheads e.g., by optimizing file systems [22], [78], or designing them from scratch as PMem-aware to optimize metadata operations [47], [81], [82], [85]. Our work focuses on a different part of the stack: the performance of *DAX memory-mapped file access*. We refer to this as the memory-mapped (MM) file interface.

Prior research focuses on MM overheads deriving from the until-recently necessary DRAM buffering of the data (page cache) [60], [61]. PMem and DAX-mmap remove this necessity, as *files are already stored in byte-addressable memory*. Despite the true zero-copy access that they provide, we find that memory-mapping can still be significantly slower than system call file access. The overheads have two sources. First, fast storage exposes the overheads of Linux mmap operations. Second, direct access to fast storage enables new use cases for mmap; e.g. replacing read system calls with mmap operations for applications that access numerous small files (e.g., web and mail servers). Such new use-cases, change the traditional mmap workload and expose new overheads not previously important.

In this work, we analyze how Linux behaves when it maps files stored in PMem and observe that the interface's generic design often assumes that file mappings refer to DRAM resources. For example, all file mappings are populated lazily via page-faults and deleted synchronously to save scarce volatile memory. Such savings are irrelevant with PMem and DAX-mmap. In addition, hardware-maintained metadata – page access and dirty bits – target and drive efficient volatile memory management. By design they assist the selection of victim pages to reclaim page cache memory. With PMem and DAX, page cache management is no longer necessary. From this analysis, we identify multiple opportunities to remove unnecessary overheads targeting a design that comes close to the limits of what hardware can provide for MM direct access to byte-addressable storage.

We propose *DaxVM*, an efficient interface to byte-addressable storage, that extends the Linux virtual memory and file system layers. To reduce latency, DaxVM maintains shareable *pre-populated page tables* per file (file tables) and (de) attaches them to processes' address spaces during mmap. This eliminates paging costs and enables fast *O(1) operation* [72]. To improve scalability, DaxVM provides support for *ephemeral* file access patterns, e.g. opening multiple small files, reading/processing the data once, and closing them. Such concurrent access usually dictates the use of read/write system calls, as contention over virtual memory locks makes MM access prohibitive. DaxVM introduces a dedicated lightweight virtual address space manager that enables *ephemeral MM access* scaling to many cores. DaxVM also provides batched, *asynchronous unmapping operations* to minimize TLB coherence overheads. It also minimizes and potentially eliminates *kernel-space dirty tracking* overheads for applications that manage durability from user-space. Finally, DaxVM introduces *asynchronous background block zeroing* on PMem file systems to deal with the inherent double-writing costs of MM append operations.

DaxVM combines these techniques under a new *high-performance interface* for persistent memory operations providing new m(un)map calls. Separating the current unified volatile and non-volatile interface supports the observation that usage patterns for persistent and volatile data may differ substantially, enabling distinct optimizations.

Some of DaxVM's mechanisms are inspired by prior works; however, those works targeted different setups (e.g., block mapping for flash storage [39] or lazy unmapping for volatile memory [53]), required hardware extensions [39], or described high-level ideas [72], [83]. Our work combines these into an interface to persistent memory, implements them in a real operating system—enabling us to study the complexity and the details of their realization and synergy—and evaluates them with commodity hardware.

Despite tailoring for byte-addressable storage and PMem, various DaxVM aspects, e.g., file tables and virtual memory scalability optimizations, are relevant for fast access to other high-performance storage mediums (Section VI).

We implement DaxVM in Linux 5.1.0 with the ext4-DAX [78] and we make it publicly available[1]. For multi-threaded workloads operating for short time intervals over multiple small files, e.g., Apache [1], DaxVM improves standard mmap performance up to 4.9×. It also reverses the trend that favors read for such setups, outperforming it by up to 1.5×. It comparably boosts the performance of other applications with ephemeral access patterns that do not move data out of PMem (e.g., text search like ag [9]),. DaxVM also increases system availability, providing fast boot times for PMem databases [8]. It can finally provide up to 2.95× better throughput than baseline MM for PMem-optimized key-value stores [42] running on a fragmented ext4 image. This paper makes the following contributions:

- We detail the inherent costs of MM file access and we identify virtual memory features that assume data are always buffered in DRAM. We show how byte-addressable storage attributes can be leveraged to control the costs.
- We integrate file tables with a well-tested kernel-space file system (ext4) and show how they can eliminate paging costs via O(1) mmap. We study the address translation overhead implications of placing page tables on a slower medium (PMem) and show mitigations.
- We exploit the potentially ephemeral lifetime of DAX mappings and their access characteristics for fast address space (de)allocation and lazy unmappings, significantly improving DAX MM scalability to many cores.
- We show that block pre-zeroing is an inherently different requirement for MM access than write syscalls which undermines MM benefits. We demonstrate how asynchronous pre-zeroing in the file system removes this cost.
- We show that kernel's dirty page tracking harms performance even for applications that manage durability from user-space, and enable bypassing these costs.
- We show that a dedicated interface for DAX mappings unleashes optimization opportunities not possible with POSIX strict semantics.
- We combine all the above in DaxVM, an interface close to the limit of what hardware can provide for MM access. We provide an end-to-end implementation in Linux and evaluate it on a real system.

[1]DaxVM is available at https://github.com/cslab-ntua/DaxVM-micro2022

II. Background

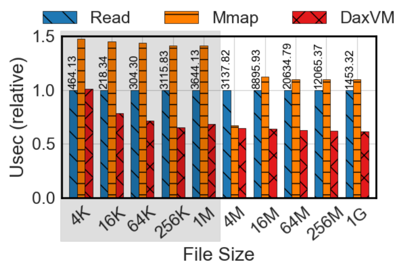
DAX [3] mechanism enables PMem file access without buffering data through DRAM (e.g., copying files pages in page cache). With DAX-mmap, virtual pages are directly mapped to PMem physical locations. The OS virtual memory subsystem creates virtual-to-pmem translations and persistent data can be accessed via load/store CPU instructions.

Memory-mapped and system call file access. Intuitively, memory mapping files holds important advantages compared to system call access (read/write), even for traditional block storage. Memory mapping files spares crossing the user/kernel boundary on multiple same file access, and always avoids at least one data copy. However, with block devices, file data must still be copied from the device to the volatile page cache to be mapped. In addition, the avoided extra copy (compared to read/write) is relatively cheap; from page cache to private per-process memory locations. Hence, for decades the guidance has been to use memory-mapping over block devices *only* when files are big and accessed randomly or multiple times [74].

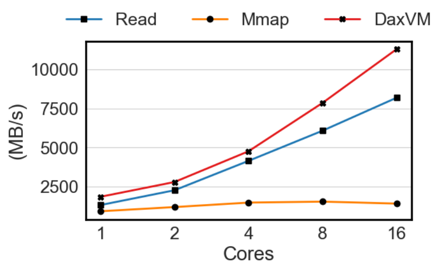
PMem and DAX mappings offer true zero-copy storage access for the first time. However, with the faster and direct access storage path, the performance bottleneck moves from the device latencies to the software stack, with the complex and prohibitively expensive virtual memory operations being the primary source of overhead. In this paper we seek to study how this affects the decades-old trade-off between memory-mapped and system call file access.

III. Memory as File Interface

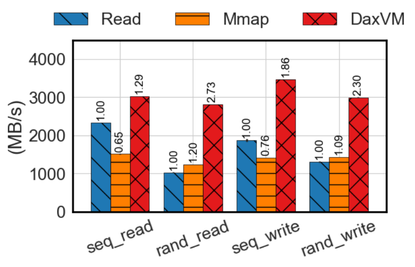
In this section we study how DAX zero-copy memory-mapped file access performs compared to system call access. We aim to understand the inherent overheads of file mapping. We run experiments on a system equipped with 384GB Intel Optane DCPMM (PMem) in AppDirect mode with an aged ext4-DAX file system using micro-benchmarks (Section V provides details regarding our methodology).



(a)



(b)



(c)

Figure 1: DAX interfaces: (a) the latency of reading a file once via MM is worse than read system calls, especially for small file sizes (lower is better), (b) MM read-once access does not scale to many cores (higher is better), (c) MM repetitive access on a large file can perform worse than read/write (higher is better). All results are from an aged ext4 image. DaxVM significantly reduces latency and improves scalability for MM, regardless of the file system fragmentation.

One time access. First, we examine the latency of accessing multiple files once: opening them, reading their content and closing them. This is a common access pattern in server workloads (e.g., web servers, mail servers). For memory-mapped access we measure the latency of mapping a file (mmap), access all its data in-place at 8-byte granularity (sum them) and then unmap it (munmap). For system call access, we read the file data into a private buffer (read) with one call and consume them similarly.

Figure 1a shows the latency results for a single thread as a function of the file size. We plot the average of reading up to 50K files or as can fit on 100GB of storage. For small files (shaded), memory-mapped access is significantly slower than read (up to $\sim$30%) despite the avoided copy. We refer to this as the *small files problem*. For larger files, the performance of memory-mapped access depends heavily on the number of huge persistent pages that back the mapped file. For files close to 2MB, memory-mapped access performs significantly better than system call access as files are 100% huge page covered. However, as the file size increases the performance drops non-deterministically, depending on the mix of small and huge pages that back the file, due to fragmentation on the aged ext4-DAX system [46]. For example, memory-mapped access performs $\sim$10% worse than read for 1GB file on this run. We did not consider 1GB huge pages in our experiments.

Next, we focus on throughput, and perform the same access pattern but over 32KB files using multiple threads. Figure 1b shows the operational throughput. As thread count increases, memory-mapped access does not scale to many cores, corroborating a known problem [18], [25], [26].

Repetitive access. Figure 1c shows the operational throughput of another common file access pattern, that of repetitive operations over the same large file (e.g., databases). We issue sequential and random 4KB reads and writes over a 100GB file. For memory-mapped access, we initially map the entire file and use memcpy() [46], [47], [80] with AVX512 instructions [56], [86] and non-temporal stores [77] to perform reads and writes [46], [47], [56], [80]. We observe that memory-map performs equally (for random access) or even worse than system calls (for sequential access) [32].

We now discuss inherent memory-mapped access overheads and compare them to system call behavior. We examine how PMem attributes affect some of the current OS design assumptions. We discuss prior work and seek ways to minimize the overheads; we target a design to stress the limits of memory as a file interface.

A. *Virtual Memory Overheads*

Unlike system call file access, memory mapping inherently involves virtual memory operations that are costly.

1) Paging: Memory-mapped file access requires a page table entry (PTE) for each mapped page of the file. Linux populates virtual mappings lazily, adding the cost of a synchronous page fault to create PTEs at page access time.

DAX impact: With block devices, file content is page cache buffered so faulting is important for fine-grain volatile memory management. With PMem, moving data between storage and DRAM is unnecessary as files are already in a byte-addressable medium. Also, the entire set of physical translations is known upfront and changes slowly as it reflects storage locations.

Prior works: Multiple works [39], [56], [68], [72], [79], [83] leverage the concept of page tables maintained by the file system to translate file offsets to PMem physical locations. O(1) memory [72] suggests the high-level idea of sharing them among processes to eliminate paging, following an older proposal for flash SSDs [39], but the design is only discussed at a conceptual level. MERR [83] emulates O(1) operations for fast address space randomization and requires special hardware to support per-process permissions. SIMFS [68] implements some key concepts but fails to support file sharing with different permissions and requires a preset global maximum file size. All the above also emulate persistent memory so the performance of persistent page tables remains unclear. Finally, ctFS [56] integrates file tables to a FS that maps the entire DAX device to user-space, trading secure (meta)data operations for fast appends. Table I summarizes the limitations of prior works and the key aspects of the DaxVM approach.

	FlashMap [39]	SIMFS [68]	O(1) [72]	Merr [83]	ctFS [56]	**DaxVM**
PMem storage		✓	✓	✓	✓	✓
Real OS implementation	✓	✓			✓	✓
Commodity hardware	✓	✓			✓	✓
O(1) mmap	✓	✓	✓	✓		✓
PMem/DRAM page table management						✓
Scalable mmap						✓
Fast unmap						✓
Per-process permissions	✓			✓	✓	✓
Dirty-page tracking avoidance						✓
Asynchronous block pre-zeroing						✓

Table I: Comparison of DaxVM with prior works that focus on memory mapping storage.

DaxVM approach: We provide flexible O(1) memory mappings via *pre-populated file tables* integrated on a well-tested kernel file system, without restrictions and using only existing hardware on a real system. We manage the potential overheads (e.g., TLB miss costs) of shared page tables residing on persistent memory.

2) Virtual address space management: File mapping requires (de)allocating an area in the process address space to (un)map the file. Operating systems serialize address space operations (e.g., virtual address allocation), limiting manycore scalability of virtual memory. For example, Linux protects the entire virtual address space of a process with a semaphore (mm→mmap_sem) [31].

Linux also records all allocated virtual memory areas (VMA) in a centralized data-structure (the VMA red-black tree). This fine-grain recording enables the support of a variety of POSIX memory operations (partial munmap and mprotect, etc.). However, it induces significant lock contention when VMAs live briefly (are quickly unmapped). Applications that access many small files once before closing (e.g., web servers, mail servers) issue frequent (un)map requests but rarely any other memory operations. On the other hand, applications that repetitively access files (e.g., databases) commonly use complex operations (mremap, etc.).
Prior works: Clements et. al [24], [25] proposed using concurrent data structures to enable simultaneous operations on different address ranges. However, a relevant Linux implementation [31] showed that the transition is not trivial because the contention may be transferred to range locks as memory operations commonly affect multiple ranges [52].

DaxVM approach: We focus on file mappings only, and differentiate the address space (de)allocation scheme based on the expected mapping lifetime and the required operation support. We provide lightweight file mappings that trade complex memory operations support (e.g., partial mprotect/mremap) for scalable virtual address space (de)allocation operations.

3) Synchronous resource release: File unmapping releases virtual addresses and requires maintaining virtual memory coherence. POSIX dictates that the release occurs synchronously, i.e., before the operation returns. This requirement for synchrony requires clearing PTEs and invalidating corresponding TLB entries in local and remote cores (shootdowns). If stale translations remain in the TLB, an application could access reclaimed physical memory, raising correctness and security issues. Shootdowns are inherently non-scalable operations, requiring synchronous inter-processor interrupts (IPIs) [17] that cost up to thousands of cycles [14].
DAX impact: With block devices, unmap operations also potentially release physical resources (page cache) under memory pressure. PMem mappings no longer occupy volatile memory; thus, unmap operations release only virtual addresses. The mapped persistent memory is independently and exclusively reclaimed by specific file system operations (e.g., when files truncate) whose frequency is relatively low [15].
Prior works: To avoid scalability overheads, state-of-the-art PMem user-space filesystems, never unmap the files [47], [56]. For example, SplitFS [47] maps files under the hood and keeps them mapped to user-space until the process dies or the files get truncated. This extreme strategy raises safety concerns [83]. LATR [53] on the other hand, uses a message-passing mechanism in place of IPIs to invalidate TLBs locally and asynchronously [26], but in very short time intervals as it targets volatile memory. This general-purpose mechanism is complex, error prone [14], and still suffers from scalability issues due to its own locking (Section V).

DaxVM approach: We focus on file mappings only, and take advantage of the fact that the mapped physical resources (storage) are reclaimed slowly and independently. We opt for a simple design of lazy unmapping; we still unmap files but asynchronously and use existing robust mechanisms to achieve that.

4) Dirty page tracking for file syncing: With block devices, both system call and memory-mapped write access is buffered in volatile memory and the corresponding pages are tagged as dirty in the OS page cache metadata tree, to be flushed to disk during sync (e.g., fsync).
DAX impact: Persisting data requires just writing-back dirty CPU cache lines. Thus, DAX write system calls commonly

bypass the CPU caches using non-temporal store instructions to copy data to the device (e.g., ext4-DAX, NOVA), omitting the need of any dirty tracking.

With DAX memory-mapped access, however, the kernel still has to record the physical regions that user-space dirtied, to be able to flush the corresponding CPU cache lines on sync operations. Thus, the OS still uses the page cache tree to tag dirty pages when DAX-mmap is used. While x86 hardware page walkers set the PTE dirty bit when a page is first written, Linux also tracks mapped dirty pages in software. It initially marks pages as read-only and relies on permission faults to detect writes and update the page cache tree [4], [65], [89]. Sync operations write-protect file pages again for all mapping processes after flushing, to restart the mechanism. We measure that performing one msync call every 10 write operations (random access, 1KB each) on a memory-mapped 10G file causes ~2.8x more faults compared to no sync.

Prior works: DAX mapping allows applications to manage data durability from user-space, omitting sync system calls. Prior works actually recommend this approach [47], [56], [80]. The application can use non-temporal stores or cache line flush instructions (e.g., clwb and sfence) over DAX mappings to persist file data at the granularity of bytes. However, the OS in those works still tracks each initial dirty page access and remains oblivious to user-space managed endurance. In this way, the system remains compatible with sync operations but pays all the overheads of dirty-page tracking.

DaxVM approach: We drop all kernel-space dirty page tracking activity for applications that manage durability from user-space, to achieve maximum performance.

B. *Double writing for secure appends*

Append operations are write operations that involve block allocations by the file system.

DAX impact: As write system calls directly update block content via non-temporal stores, the newly allocated storage blocks do not necessarily require zeroing for security. Ext4-DAX zeroes out blocks on write system calls, but NOVA and PMFS do not as they are optimized for PMem.

To append a file via memory mapping requires first allocating new blocks (e.g., via fallocate or ftruncate) and then mapping them to user-space for write access. In this case the new blocks must be zeroed; otherwise user-space will get access to stale data from deleted files previously using the same blocks, causing a security leak. Therefore, block zeroing is necessary for DAX memory-mapped appends, which doubles the writes per operation and penalizes performance. We measure that $\sim 30-40\%$ of append operation latency is spent in block zeroing, irrespective to the append size.

Prior works: The exact same security concerns exist for volatile memory, and multiple works [59], [64] propose asynchronous page pre-zeroing to control allocation overheads. For storage, asynchronous zeroing is not that simple as it can consume the available bandwidth and stall concurrent requests to the device. It is only considered for SSDs (garbage collectors) due to their erase-before-write NAND nature and multiple works attempt to mitigate GC overheads [49].

DaxVM approach: Inspired by volatile memory strategies and harnessing the high PMem bandwidth we adopt asynchronous pre-zeroing in PMem file systems.

C. *Micro-architectural performance*

Memory-mapped and system call file access behave differently at the micro-architectural level. These are fundamental observations about processor and OS operations that we cannot work around.

TLB performance. Linux maps the entire PMem physical space with huge pages. Thus, the internal copy of a read/write call benefits from reduced TLB misses, even when files are <2M or fragmented. But, small file memory-mapped access always pays small page TLB miss costs and large file access performance depends heavily on file system fragmentation and the ability to use huge pages.

Cache performance. System calls copy data, which prefetches persistent data into higher layers of the cache hierarchy. User-space code runs faster hitting in the caches. With memory-mapped access the user-space code will pay the cost of fetching data from persistent memory.

Vectorization. User-space memory-mapped access can use Advanced Vector Extensions [40], to perform SIMD operations over file data (e.g., memcpy). This can significantly improve performance [38], [56], [86]. Copies inside kernel system calls cannot benefit from AVX instructions as supporting them would introduce register save and restore overheads when crossing the user-kernel space boundary.

IV. DaxVM

Based on the observed opportunities we design and implement DaxVM, a fast and scalable MM interface aiming to come close to the limit of what hardware can provide. DaxVM extends the OS memory-management and file system layers and consists of five key components.

1. Fast paging operations through pre-populated file tables. With DaxVM, the file system maintains pre-populated page table fragments that translate file offsets into storage physical addresses. These *file tables* are attached/detached to processes page tables during m(un)map operations, eliminating the paging setup and teardown costs of DAX mappings.

2. Scalable address space management for ephemeral mappings. DaxVM maintains a dedicated heap to serve fast (de)allocation requests for *ephemeral mappings*. These mappings are expected to live for short periods and support no other operations (e.g., protection change through mprotect).

3. Asynchronous resource release; batching unmap requests. If application correctness does not rely on synchronous unmapping, DaxVM's virtual memory layer can

optionally defer munmap operations. It tracks the *zombie* mappings and releases them asynchronously in batches. This eliminates frequent fine-grain TLB shootdowns.
4. Low durability cost. DaxVM provides a mode that drops all kernel-space dirty tracking for applications that manage durability in user-space and opt for maximum performance.
5. Asynchronous storage block pre-zeroing. DaxVM extends PMem file systems to asynchronously zero out storage blocks when they are freed (e.g., unlink, truncate).
DaxVM comes as a *new interface*, with stripped-down POSIX features and relaxed restrictions targeting performance (Section IV-F). It adds two new system calls: daxvm_mmap and daxvm_munmap along with new optional flags.

DaxVM speeds applications that perform frequent m(un)map operations, e.g., briefly process small files, applications sensitive to paging (e.g., databases), and allocation-intensive workloads especially on fragmented FS images.

We implement and evaluate DaxVM in Linux 5.1 and the ext4-DAX [78] and NOVA [81] file systems. We target the x86-64 architecture. DaxVM primarily targets DAX-aware file systems that relax data operation atomicity for performance (e.g., NOVA relaxed [81], xfs-DAX). They allow in-place updates on DAX mappings. Atomic copy-on-write updates, e.g., shadow paging, negate DaxVM benefits with frequent page table updates.

A. O(1) mmap

DaxVM maintains pre-populated page tables per file, and (de)attaches them to process address spaces during m(un)map operations. This eliminates paging costs and provides instant access to files irrespective to their size (O(1) operation). For the rest of this paper we refer to the them as *file tables*.

1) Pre-populated File Tables: They are fragments of an x86-64 page table (radix tree) and translate file offsets to PMem addresses. For example, if a 1MB file is stored in pages P1-P256 of PMem, the file table stores the addresses of P1-P256 starting from index 0.
Maintenance. Upon storage allocation (e.g., append), DaxVM populates the file's tables with the physical addresses of the newly allocated PMem pages. Upon storage deallocation (e.g., ftruncate), DaxVM clears the file table entries and/or frees the corresponding file tables.
Fragments instead of entire trees. Files usually shrink/grow sequentially and densely at the end of the file. Unlike the sparse population of virtual address spaces, this characteristic makes it possible to build file tables in a bottom-up fashion. For small files, we use a single 4KB page of PTEs, and expand in 4KB increments. In Figure 2 only the PTE level of the radix tree is needed to hold the translations of inode 2. This bottom-up maintenance controls file tables storage tax.
Huge Pages. For aligned huge page blocks in larger files, DaxVM supports huge PMD entry formats. To simplify the description, we consider the general 4KB block condition.

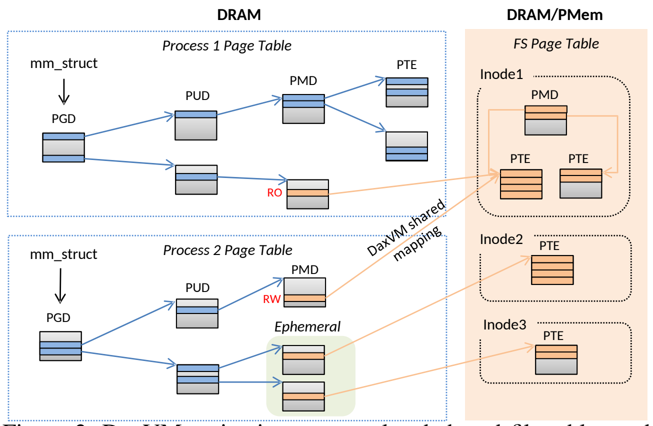


Figure 2: DaxVM maintains pre-populated shared file tables and attaches them to processes address spaces for O(1) mappings.

PTE status bits. Page table entries also maintain status bits to record per-page process access. Surprisingly, we find that most of these bits track metadata mostly relevant to volatile memory management: the access and dirty bit are mainly used for page cache evictions and volatile memory reclamation. DaxVM drops their maintenance in the file table entries, as reclamation happens explicitly during file delete for DAX mappings. DaxVM sets the PTE permission bits to maximum and supports per-process access at 2MB or coarser granularities. Similarly, it manages durability at coarser granularities. We discuss both in the next section.
Dynamic File Table Management. File tables can be maintained both in volatile and persistent memory.

Volatile file tables are re-constructed each time the system accesses an inode for the first time, loading it to the VFS inode cache (cold open). The table's root pointer is stored as metadata in VFS inodes. As long as the inode is cached, the tables are updated. When the inode is evicted from the VFS cache, they are destroyed.

Persistent file tables are stored in PMem pages and survive across power cycles/failures. Their root pointer is stored as metadata in the file's permanent inode struct. During table updates, the table entries must be flushed from the CPU caches synchronously to guarantee persistence. To control this overhead, DaxVM leverages that multiple PTEs are usually updated sequentially within a single operation (e.g., append). When possible, it batches their flushes at cache-line granularity (8 64-bit PTEs in x86_64).

Persistent tables occupy storage resources but provide good cold-start performance and save DRAM as they substitute parts of multiple processes' page tables. Apart from the storage cost, they introduce higher TLB miss costs, as page table walkers have to access slower memory [11]. Table II shows the average page walk latency measured with *perf* when we perform sequential and random reads on a file mapped using volatile and persistent file tables. We observe that with random access and persistent page tables, TLB

Benchmark	DRAM file tables	PMem file tables
seq_read	28	103
rand_read	111	821

Table II: Average page walk cycles measured for sequential and random 4K access on a 10G memory-mapped file.

Performance Monitor	
AvgPageWalk	Total Page Walk Cycles / Number of TLB misses
MMU overhead	Total Page Walk Cycles / Execution Time Cycles
Rule	if (AvgPageWalk $>$ 200 c) and (MMU overhead $>$ 5%) migrate

Table III: DaxVM monitors the average TLB miss costs and MMU overheads to migrates file tables to DRAM if necessary.

misses can cost up to 800 cycles. On the other hand, keeping all file tables always in DRAM can lead to waste of resources, while aggressively reclaiming them can penalize performance as they will have to be re-constructed frequently.

To keep the best of both worlds, DaxVM maintains volatile tables for files smaller than a threshold (32KB) and persists them for larger. This policy controls the storage tax which is high for small files (e.g., for every 4KB file a 4KB PTE is allocated). DaxVM also monitors the MMU performance of applications via performance counters [59]; it tracks the average page walk latency along with the average time spent in page walks (MMU overhead) (Table III). If the latency is above 200 cycles and 5% of the execution time is spent in walks, DaxVM (i) builds asynchronously volatile tables (copying the persistent ones) and (ii) walks the process tables to detach the persistent fragments and attach the new volatile. After DaxVM migrates file tables to DRAM both volatile and persistent tables are maintained.

File Tables and Crash Consistency. For journaling systems, e.g., ext4, file tables are updated within a journal transaction. When the transaction is committed, the tables are guaranteed to be consistent and persistent. Similarly, for a logging FS like NOVA, file table updates happen before the (meta)data updates are committed (log entry appended). File table PTEs are flushed on write and re-use the fence from the FS log/journal commit. Incomplete PTEs are recovered on reboot when replaying open transactions. The overhead of persisting file tables is included in all our experimental results.

2) Fast table (de)attachment: DaxVM uses file tables to minimize the cost of creating a mapping. When an application maps a file, DaxVM populates all translations for the requested target file offset (i.e., mmap-populate). It attaches parts of the pre-populated file tables to the process's private page table. Figure 2 depicts how DaxVM builds a page table in DRAM (blue) up to the PMD level. Then it attaches the pre-populated file PTE (orange) on the PMD. Attachments enable O(1) mmap operations; the latency is near constant with respect to file size.

Mapping size. Attaching a file table's fragment updates interior pointers at some level of the process's private page table radix tree. Therefore, the attachment can happen only at certain granularities/levels, i.e., at PMD, PUD, etc. In addition, the mapping's virtual address and the corresponding file offset must be properly aligned, i.e., to 2MB for PMD.

To enable O(1) mmap, DaxVM silently rounds the size and file offsets attributes of the daxvm_mmap system call to the granularity of the next level of the process's page table tree. Up to 1GB, files are mapped using PMD entries at 2MB granularity, and files above 1GB are mapped using PUD entries at 1GB granularity. This leads to the anomaly that mapping files >1GB can be faster than smaller files.

DaxVM returns to the user the virtual address that maps to the requested file offset. A larger portion of the file may be silently mapped (before/after the requested boundaries).

Permission rights per process. With DaxVM, the pre-populated file page table fragments are *shared among the processes that map the same file*. The pre-populated PTEs have the maximum access rights pre-set. To enable different access permissions per process, DaxVM manages the permission bits at the attachment level (e.g., the PMD) rather than at the PTEs, as the former belongs to the private part of each process's page tables. Figure 2 shows how two different processes have read-only and read/write access rights over the same 2MB file region while still using the shared file tables. The x86 translation hardware (page walker and TLB) applies the minimum access rights found at all the page table levels for an address, enabling this strategy [45].

ASLR. With DaxVM, the address space layout randomization works seamlessly at 2MB granularity. File tables are attached to randomly allocated virtual addresses aligned to 2MB. The file data will always be located at the same offset within the random 2MB region (alignment restriction).

FS extensions. To support O(1) mmap a file system must be extended to (de)construct and update file tables during storage block (de)allocations and to attach them during mmap.

B. *Ephemeral mappings*

As discussed in Section III, the centralized locking of a process's virtual address space prohibits issuing parallel frequent m(un)map operations as they cannot scale to many cores [19]. This almost excludes MM as an interface for a common file access pattern: open a file, quickly process its data and close it. An old study on distributed file systems shows that 75% of files are open for less than a quarter of a second [15]. We refer to this as *ephemeral file access*, and DaxVM provides a dedicated address space manager for *ephemeral mappings* of persistent memory files. It builds its strategy for better scalability on the idea that such mappings do not require support for complex virtual address space operations beyond unmap.

Ephemeral heap. DaxVM pre-allocates a virtual address range (*ephemeral heap*) in the process's address space and manages it independently to (de)allocate virtual address regions for ephemeral mappings. The allocator's objectives are similar to a user space heap allocator (e.g., malloc()): to

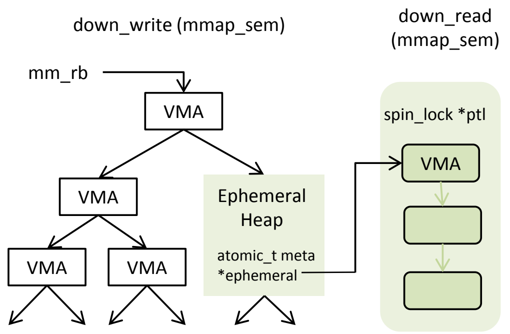


Figure 3: DaxVM ephemeral VMAs.

quickly allocate and free address ranges. It does not have to support splitting and merging of mappings.

Our current heap implementation leverages short mapping lifetimes to perform linear allocations. The heap is dynamically extended in virtual regions of 1GB, to avoid exhaustion. Each region's virtual addresses are reclaimed only when all the mappings populating it are destroyed; tracked by a counter. Thus, currently allocations from the ephemeral heap resemble a stack; but other allocation schemes can be applied.

Ephemeral mapping visibility and tracking. Only munmap operations are allowed for ephemeral mappings; any other operation (mprotect, mremap, etc.) that falls inside the heap's range returns an error. As complex per-mapping support is omitted, ephemeral VMAs do not need to be recorded by the virtual memory's core data structures, i.e., the VMA red-black tree (`mm_rb`). It is sufficient that the manager records only the aggregate ephemeral heap region. This enables tracking ephemeral VMA's in a dedicated data structure, a list (or a tree) associated only with the heap (Figure 3).

The major advantage of this design is that lightweight locking can be used to protect this structure, avoiding contention over the global manager's locks. In Linux, the entire address space of a process is protected by the heavily contended mmap semaphore [31], which mainly protects the VMA tree. Table IV summarizes the main code paths that contend for the semaphore for insight. DAX mappings inherently are not

	Path	Target	Reader/ Writer	DAX Mappings	Ephemeral Mappings
A	Khugepaged Ksm Mlock Madvise Mempolicy	Volatile Memory Management	R/W	✗	✗
B	Page Fault	Populate Mapping	R	✓	✗
C	Mremap Mprotect Exec	Resize mapping Change Perm Set up binary	R/W	✓	✗
D	Mmap Munmap Fork Msync	Create Mapping Dissolve Mapping Duplicate mm Flush dirty pages	R/W	✓	✓

Table IV: Paths acquiring the mmap semaphore and their involvement in DAX and ephemeral mappings management.

involved in paths that target volatile memory management (set A). On top of that, DaxVM ephemeral mappings do not fault often (only for dirty page tracking) (set B) nor support memory operations (set C). This leaves mainly m(un)map to contend for the semaphore (set D); simplifying the design of a more scalable address space manager.

We use atomic operations to update heap's metadata and a spinlock to protect the ephemeral VMA list. Heap (de)allocations hold the mmap semaphore as readers. The idea of VMA locks instead of a global semaphore has been discussed [28], with the concern that they could lead to contention for one big VMA lock. In our design, ephemeral heap locking scales because the operations that take place under the lock are stripped down and fast.

Ephemeral VMAs are still visible to the file system. They are attached to the address space trees that track the VMAs that map each file (address_space→i_mmap). This enables their management (e.g., unmapping) by the file system.

C. *Optimized munmap*

Unmapping a virtual region involves three steps: (i) clearing/destroying the page tables, (ii) invalidating the local and remote TLB entries (shootdowns) that cache the region's PTEs, and (iii) releasing resources. DaxVM detaches file tables instead of destroying them.

Async unmap. TLB shootdowns are inherently non-scalable as they require IPIs. Linux batches the virtual addresses of a single munmap request to perform a cheaper range TLB invalidation (one IPI) instead of individual page shootdowns. After a certain threshold (33 pages for x86), it opts for a full TLB flush as the gains of the flush are estimated to outperform the penalty of the TLB misses introduced.

DaxVM builds on this strategy and gives the option to *not perform munmap operations synchronously* at all. It records the VMAs that the user requested to unmap, the now "zombie" VMAs, and defers their unmapping to batch TLB invalidations *across requests*. It tracks the total number of zombie pages and when a threshold is reached, it tears down their corresponding page table entries and performs a single full remote TLB flush on the cores that the application runs. It does so on the munmap request that exceeds the threshold. Apart from the key advantage of replacing frequent TLB invalidations with fewer, cheaper, entire TLB flushes, the virtual memory locks are also held for shorter periods.

File system races. While an unmapping is deferred, the size of the mapped file may change if the file gets truncated or even deleted. DaxVM maintains safety by synchronously forcing unmappings if storage blocks are reclaimed.

D. *Durability management*

DaxVM fully supports msync and fsync calls in the same way as default DAX through permission faults. DaxVM tracks dirty regions at 2MB or coarser granularities, as access permissions are held at the attachment level of the file tables.

For example, if a 4KB page is written, DaxVM will mark the entire 2MB region as dirty in the page cache. Note that the same happens if a huge page backs the file. This can potentially penalize fsync calls, but reduces dirty tracking overheads, as fewer permission faults take place (Section V).

DaxVM does not require userspace durability management to work properly. But to further stress performance limits, DaxVM has a *nosync mode* for applications that manage durability from user-space [80]. In this mode, it does not track dirty pages via permission faults and does not record them at all in the page cache metadata tree. In a nutshell, it drops sync operation support (e.g., msync), which becomes a no-op, and data durability becomes entirely *a userspace responsibility*. This creates a race condition if a file is mapped via DaxVM and POSIX simultaneously: data modifications of the DaxVM mapping might not be captured by the msync() operations of the POSIX mapping. To manage this, DaxVM pushes the cost to the POSIX process, which flushes the entire file during its msync().

E. Asynchronous block pre-zeroing

DAX memory-mapped append operations – unlike system calls – inherently require the zero-out of the newly allocated blocks for security reasons, doubling the write activity and penalizing performance by $\sim$30-40% irrespective to the append size. DaxVM extends the file system to pre-zero blocks asynchronously to avoid this cost.

DaxVM does not interfere with the file system block allocator, to avoid inducing involuntarily external fragmentation to the system. With storage, external fragmentation matters in the granularity of extents – rather than pages – which can grow up to multiple MB. Instead, DaxVM hooks the file system's free operations. Upon a file truncate operation, the blocks to be freed are kept on per-core lists instead of being immediately released to the FS block allocator. A rate-limited kernel-thread periodically scans the lists and zeros-out blocks using non-temporal store instructions for persistence and to minimize bandwidth consumption [86]. Once a whole set of blocks-to-be-freed is zeroed, they are released to the allocator. Per-core lists preserve the scalability of free operations.

With PMem, more so than volatile memory, pre-zeroing consumes precious bandwidth and can potentially penalize other operations. To avoid BW saturation we throttle bandwidth to a configurable amount on an idle core.

F. DaxVM forms a new relaxed interface

Many of DaxVM's mechanisms derive performance by relaxing some POSIX strict requirements and abandoning some POSIX functionalities, e.g., advanced memory operations support for all file mappings. The impact of the interface on scalability and performance is a formally studied topic [26].

DaxVM interface consists of two new system calls (daxvm _mmap and daxvm_munmap). Daxvm_mmap implements O(1) file tables attachment and currently supports shared mappings. From the rest of the POSIX flags, DaxVM currently supports MAP_SYNC and adds three new flags.

MAP_EPHEMERAL: the mapping is expected to be brief and does not need any memory operation support. This flag activates the ephemeral address space allocator.

MAP_UNMAP_ASYNC: the program does not require access faults right after unmap. Activates asynchronous unmapping.

MAP_NO_MSYNC: this flag is combined with MAP_SYNC and means that the program will not rely on msync functionality at all. This flag activates the *no sync* mode, where all dirty page tracking is dropped and msync becomes a no-op.

We now discuss how DaxVM affects other operations.

Memory protection. Partial mprotect system calls over DaxVM mappings fail. DaxVM only allows changing the permissions for an entire mapping. Moreover, when the MAP_EPHEMERAL flag is set, any mprotect call fails.

Mremap. Similar to mprotect, DaxVM allows only mremap calls on the entire mapping (e.g., to resize) and fails if MAP_ EPHEMERAL is used.

Madvise. madvise is used for volatile memory management (e.g., page cache), thus DaxVM does not support it.

Msync. DaxVM supports msync as is, unless MAP_ NO_MSYNC is used when msync becomes a no op.

POSIX comparison. POSIX maps files in multiples of pages and references beyond the mapping's last page results in a segmentation fault. DaxVM guarantees that at least the portion of the user requested is mapped, but a portion before and after may also be silently mapped to the process address space (for proper alignment that enables O(1) mmap). If the file is extended inside this virtual portion, the new pages are automatically mapped to the address space. Moreover, POSIX promises synchronous unmappings. DaxVM relaxes that requirement, but guarantees that mappings are removed before physical and virtual resources are reassigned.

G. Discussion and summary

Security and correctness. Daxvm_mmap may map more of the file than requested. If entire file's content must not be visible to the calling process, DaxVM must not be used. Also, with MAP_UNMAP_ASYNC, user accesses to unmapped regions may not trigger an exception for a time window after a daxvm_munmap call. Correctness is not violated as DaxVM guarantees that the virtual regions will not be re-used before the page table and TLB entries are invalidated. However, if an application depends on traps triggered by accesses to unmapped regions (e.g., userfaultfd() or guard pages), MAP_UNMAP_ASYNC should not be used. With respect to security, if an application expects attacks, e.g., untrusted code injection/execution, DaxVM increases the time that data are vulnerable, keeping them mapped for longer than expected. Note that some DAX user-space file systems (e.g., SplitFS [47]) map files under the hood indefinitely. Applications can limit this behavior omitting the MAP_UNMAP_ASYNC flag.

Huge pages. Currently Linux and various file systems try to control DAX paging overheads by backing files with huge (2MB or 1GB) pages. DaxVM supports large pages when present, harnessing their TLB performance advantages. However, huge pages are very sensitive to FS fragmentation [46], due to alignment restrictions, and cannot be used for files smaller than 2MB. For both cases, DaxVM eliminates paging and sustains high performance (more in Section V).

Programmability. Applications must change to use DaxVM interface, replacing either a read system call or a POSIX mmap. Simple uses of mmap can be replaced directly, while reads should be replaced with daxvm_mmap and direct access to the data. MAP_EPHEMERAL is meaningful for files accessed briefly (i.e., once) and closed, but functionality does not break if used with mappings of longer lifetime.

Applicability. In a nutshell, DaxVM enables performant and scalable concurrent m(un)map requests and minimizes paging costs, unleashing DAX benefits also for applications that perform short-lived accesses to smaller files (a usage previously favoring read/write). DaxVM is still beneficial to applications that access files mapped for long periods (e.g., databases), especially on fragmented FS images, acting complementary to huge pages.

V. Evaluation

A. Experimental Setup

Our experimental platform is equipped with an Intel Xeon Gold 5812T Cascade Lake CPU with 2×16 physical cores, with frequency fixed at 2.7 GHz and SMT disabled. Each socket is equipped with 94GB DRAM and 384GB Intel Optane DCPMM (PMem) in AppDirect mode (3 DCPMM DIMMs). We limit our experimentation in a single socket. To study interfaces performance under realistic file system conditions, we use the Geriatrix [48] tool to age the file system image. We use the suggested [46] Agrawal profile [12] and apply 100TB of write activity to PMem (70% utilization).

We implement DaxVM in Linux kernel v5.1.0, incorporating it with ext4-DAX [78], NOVA [81], and the core virtual memory manager. We use a set of micro-benchmarks and real-world workloads to evaluate DaxVM in relation to (i) system call file access (read and write) and (ii) the default DAX-mmap interface. Due to space limitations, our evaluation focuses on the commonly used ext4-DAX FS. We discuss where results differentiate significantly with NOVA. We use the *nosync* mode when applications enforce durability from user-space. For Linux mmap we consider both lazy page faulting and pre-faulting (MAP_POPULATE flag – populate). We also provide some comparison with an asynchronous unmapping technique (LATR [53]). We run experiments three times and plot the average.

B. Micro-benchmarks

Because there is no standardized benchmark to profile file memory mapping and compare with read/write file access [57], [73], we construct our own set of micro-benchmarks. We revisit the same experiments as in Figure 1; we consider: (a) accessing files once – *ephemeral access* (e.g., webservers) and (b) accessing files repetitively (e.g., databases). We use AVX-512 instructions and non-temporal stores for user-space write access [86].

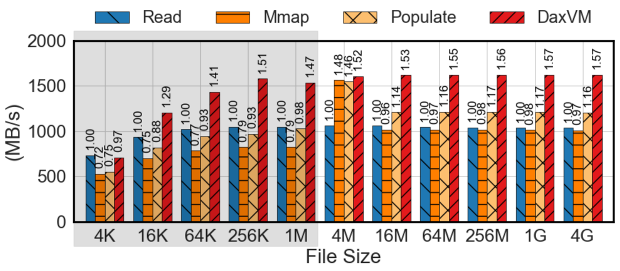


Figure 4: Read-once (ephemeral) file access.

Ephemeral access. We open 50K files (or 100GB/filesize for >2MB files), briefly process their content, and close them. For memory-mapped access, we map each file, access its data in-place at 8-byte granularity, sum them and then unmap the file. For read, we read the entire file with one system call into a private buffer and then process its data similarly.

Figure 4 reports throughput (MB processed/second) relative to read for a single thread and as a function of the file size (Figure 1 shows latency). For small file sizes (shaded), mmap performs $\sim 20\%$ worse than read despite avoiding data copies due to paging. Pre-faulting (Populate) improves performance, as the file size increases, but does not entirely solve the problem; it still pays the cost of (de)constructing page tables. DaxVM improves throughput by up to 50% over read for small files eliminating paging with O(1) mmap operations.

For larger files, baseline memory-mapped access is heavily affected by huge page coverage; reporting better performance for file sizes close to 2MB (e.g., 4MB). As the file size increases though, performance drops further and becomes non-deterministic due to the increasing number of small pages involved in the file's mapping from a fragmented FS. DaxVM's file tables provide an almost robust 55% benefit over read and $\sim 30-50\%$ over mmap, independent of the FS fragmentation.

Repetitive access over large files. We consider the case of memory mapping a 100GB file and use memcpy to perform 1KB and 4KB reads and overwrites in sequential and random order. This microbenchmark [46] mimics database operations [80]; a use-case favoring memory-mapped access as it avoids the significant cost of crossing the user-kernel boundary frequently [16]. Figure 5 summarizes our results.

For 1KB access all mmap interfaces outperform read/write access. Notably though, default mmap performs only 11% better than read for sequential access, despite avoiding 100M system calls. This is attributed to paging overheads. Pre-faulting (Populate) improves performance for read access, but penalizes it for write. For the latter, it ends up paying

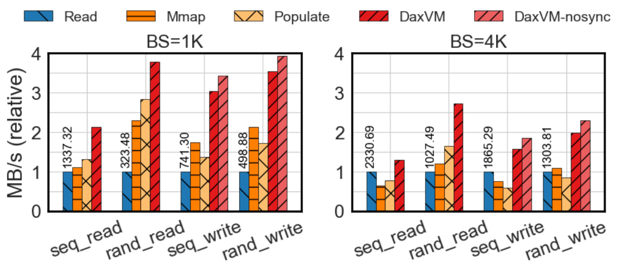


Figure 5: Repetitive file access.

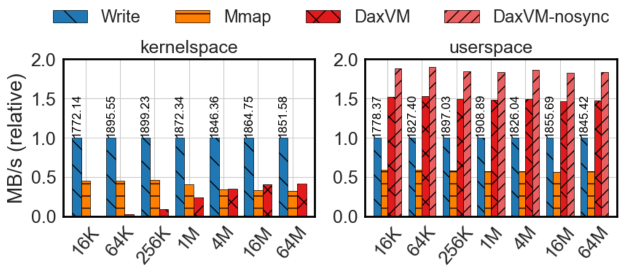


Figure 6: Kernel-space and user-space syncing operations.

the fault overhead twice for each page of the mapping: (i) pre-population and (ii) dirty page tracking faults (Section III). DaxVM eliminates all costs via O(1) file tables attachment during mmap and managing durability either (i) at 2MB granularity irrespective to fs fragmentation (faults) or (ii) entirely in user-space (nosync). It performs up to 3.9× better than system-call access and 1.9× than default mmap.

For 4KB access, default mmap, even with pre-faulting, performs worse than read/write sequential access. The avoided cost of the fewer system calls is not enough to amortize paging overheads. DaxVM outperforms read/write calls from 1.3× up to 2.72×, and mmap from 1.8× to 2.2×.

For the irregular access workloads, DaxVM's performance monitor detects the high TLB miss overheads (Section IV) and migrates file tables from PMem to DRAM. We measure that migrating the tables provides a 10% performance improvement, avoiding the costly page walks when table fragments are located in slow PMem.

Sync. With PMem, sync operations are needed to ensure modified file data is flushed from processor caches. We consider the same experiment as in Figure 5, but with a 10GB file and perform 1000 sync operations after a varying number of sequential write operations. For kernel-space syncing and MM, we use memcpy and perform periodically fsync. For user-space syncing we use non-temporal stores and omit the fsync calls. We turn huge pages off, to stress the comparison with DaxVM, that always performs flushes at 2MB granularities. Figure 6 summarizes our results for the variable syncing sizes. We omit pre-faulting results, since as discussed do not benefit write access.

Kernel-space syncing. Kernel syncing of a mapped file performs worse than DAX write syscalls (up to 68% slowdown). Writes use non-temporal stores and synchronously persist data, while fsync on a mapped file flushes CPU caches. A prior study [86] shows that non-temporal stores almost double the bandwidth of cacheline flushes. For smaller syncing (<2MB), DaxVM performs up to an order of magnitude worse than default MM because it always handles durability at 2MB granularity. However, in a non-fragmented FS image that uses 2MB pages, the default MM suffers from the same overheads due to huge pages (we measured this). Hence, DaxVM provides the same sync overhead performance trade-off with huge pages irrespective to FS fragmentation.

User-space syncing. Despite the kernel bypass, default MM performs worse than writes + kernel syncing (40%). DaxVM performs better and combined with the nosync optimization provides speedup up to 80%.

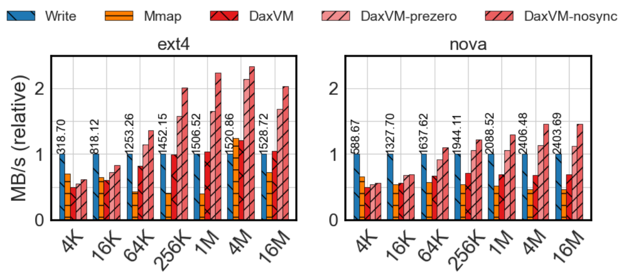
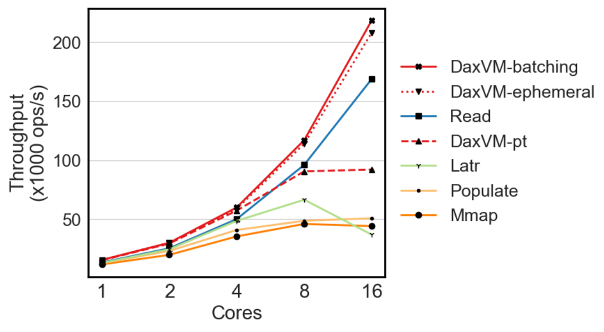


Figure 7: Append operations.

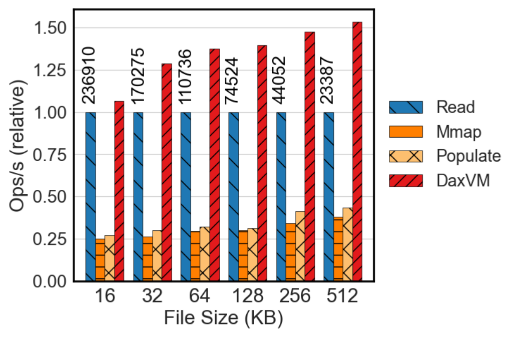
Appends. We now examine append performance via the different interfaces. As discussed in Section III, MM append operations require an fallocate() to allocate new blocks and then map them for user-space write access. For security reasons, the OS must zero all blocks before allowing user-space access. Figure 7 shows the relative throughput achieved appending variable sizes as a single operation (one system call) on an empty file from a single thread. We compare against DaxVM (i) without pre-zeroing and with kernel-level page tracking to support sync operations, (ii) with pre-zeroing, and (iii) with both pre-zeroing and nosync, where the application is responsible for data durability. Because the results for ext4-DAX and NOVA differ substantially, we present them separately.

Regarding ext4-DAX, the results show that pre-zeroing can improve MM performance up to 2× for larger file sizes (DaxVM). For ext4-DAX this reflects also as a benefit compared to system call appends, as this FS conservatively zeroes-out blocks also on the system call path unnecessarily. *Nosync* mode boosts further performance up to ∼50%, eliminating entirely durability management faults and their page cache metadata update operations. For 4MB, default MM performance improves significantly due to huge page coverage. For very small files (e.g., 4KB) DaxVM performs worse due to the overheads of page table construction.

On the other hand, NOVA is a PMem-aware file system that does not zero out blocks during write system calls, but it zeroes them out only during fallocate calls for secure user-space DAX access. Figure 7 shows how this inherent differentiation in DAX interfaces requirements leads to more



(a) Apache – scalability



(b) Apache – webpage size

Figure 8: DaxVM allows applications that issue many (un)map requests (e.g., web-servers) (b)) to scale to many cores and exposes the zero-copy advantage of MM over system call access on a setup that was previously considered prohibitive.

than 2× faster write call performance (compared to MM) even for large append sizes (>1MB). DaxVM's pre-zeroing narrows this gap, and combined with O(1) mmap (file tables) and nosync optimizations, DaxVM outperforms write syscalls by up to 45%. It eliminates paging costs and exposes the user-space benefit of AVX instructions that are unavailable to the kernel. These results underline the necessity of handling block zeroing asynchronously with PMem storage.

DaxVM storage overheads. DaxVM occupies at least 4KB for files >32KB and adds an overhead of 4KB per 2MB of data (0.2%). For file tables smaller than 32KB, DaxVM builds volatile files tables. For the 891MB Linux git tree consisting of 68K small files, DaxVM occupies 25MB of PMem, and ephemerally uses up to 216MB of DRAM if all inodes are cached in memory.

DaxVM latency overheads. DaxVM benefits come at the cost of (de)constructing page tables during FS operations that involve storage block (de)allocations (e.g., fallocate/append/unlink). We measure the latency of appends with and without DaxVM's file tables. We find that volatile table construction adds almost zero overheads. But, persistent table construction penalizes operations at worst by $\sim$ 10% for 32KB appends on an empty file, and thereafter the overhead declines and is entirely amortized for 256KB and beyond. Persistent tables are more expensive to (de)construct as cache lines are flushed for durability.

All DaxVM benefits discussed so far are attributed to O(1) mmap, durability management and asynchronous pre-zeroing. We study DaxVM 's scalability optimizations (*ephemeral allocator* and *async unmappings*) on real-world applications in the next section.

C. *Real-world Applications*

In this section we measure DaxVM performance with real-world applications operating over small and larger files. We change their source code to use `daxvm_m(un)map`.

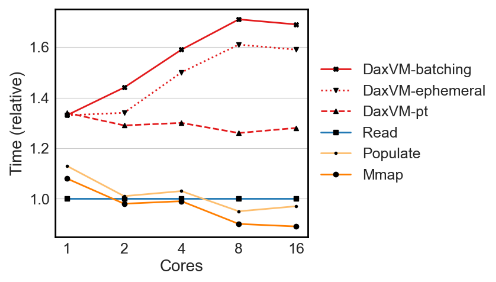
1) Small files and ephemeral access:

Apache [1] webserver uses the `mpm_event` module where threads serve requests via memory-mapped access. They map web pages, copy data into sockets, and unmap them. The scheme stresses virtual memory due to frequent m(un)map requests. We measure Apache's throughput (requests/second) while hosting static 32KB webpages stored on PMem. We use Wrk [5] to generate HTTP requests, configuring it to run with 16 threads and 16 open connections. We run Wrk and Apache on the same machine but on different sockets and scale Apache from 1 to 16 cores (socket limit). We run wrk/Apache with multiple webpages of the same size for each experiment [2] to avoid the unrealistic scenario of always hitting on the processor cache when serving a webpage.

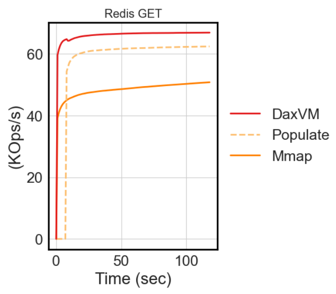
Figure 8a plots scalability results for Apache for 32KB webpages, and corroborate that its scaling is limited by virtual memory performance [14], [53]. Baseline MM access cannot scale beyond 4 cores, while read scales almost linearly up to 16. To study DaxVM performance we incrementally add each optimization, starting with pre-populated file tables.

We verify that paging significantly limits MM scalability; DaxVM's O(1) mmap via file tables enables scaling up to 8 cores and improves performance by 80% compared to pre-faulting (Populate). Address space management is the other severe bottleneck. DaxVM's ephemeral address space (de)allocation enables scaling to 16 cores and improves throughput by 100% over file tables alone. The ephemeral heap operations acquire the mmap semaphore only as readers and use independent spinlocks for ephemeral address space management (Section IV-B). This enables concurrent m(un)map requests, significantly improving scalability. Finally, for this workload batching unmap requests does not improve substantially performance over ephemeral mappings (5%). The latter is sufficient to release the stress from the mmap semaphore. Overall, DaxVM minimizes VM overheads and outperforms baseline MM by 4× and read by 30%.

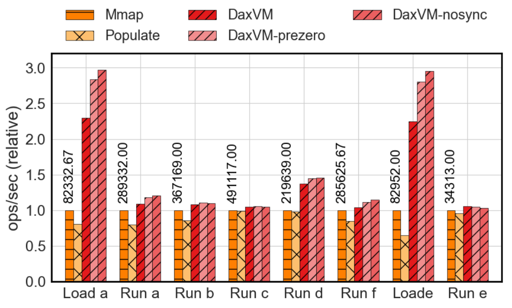
Finally, we run experiments with a kernel supporting LATR [53], a mechanism that uses message passing to replace TLB shootdowns with lazy local TLB invalidations on context switches. We run with MAP_POPULATE to control paging costs and find that LATR improves baseline MM performance by 10% at 8 cores and fails to scale



(a) Text search performance



(b) Fast boot times



(c) YCSB on RocksDB.

Figure 9: a) DaxVM improves scalability of applications that never move data out of PMem (like text search). b) Increases systems availability via fast boot time (e.g., Redis). c) DaxVM sustains high operational throughput for databases on a fragmented ext4 images.

beyond that because shootdowns are not the main problem. We find that DaxVM with only asynchronous unmapping (without O(1) mmap) outperforms LATR by 12% because: (i) DaxVM's batching can be more aggressive as it targets only PMem - it flushes the TLBs every 33 batched pages, (ii) DaxVM's batching is very simple, using existing IPIs, while LATR's status tracking mechanisms induce contention on its own locks. Note that DaxVM's asynchronous unmapping efficiency depends on the level of batching (number of pages allowed to be batched). When we increase batching level from 33 to 512, the performance increases by 20%. However, increasing the batching level increases also the DaxVM vulnerability window; the extra time that data remain mapped beyond what the user expected (Section IV-G).

Figure 8b shows how webpage size affects performance. It summarizes the relative throughput results (ops/sec with respect to read) when we run Apache at 16 cores and for increasing webpage sizes. With read system call access, Apache copies the webpage content from PMem to DRAM and then from DRAM to a socket, while with MM access it copies it directly from PMem to socket. As the webpage size increases the added cost of read's extra memory copy becomes more significant. DaxVM eliminates all paging overheads and minimizes VM scalability bottlenecks to expose the zero-copy advantage of MM access; it provides up to 50% benefit for larger webpage sizes.

Multi-threading vs multi-processing: Using multiple processes to serve requests trades system resource utilization (heavy processes vs. lightweight threads) for scalability to many cores, as there is less contention on the VM locks. Apache can run with a hybrid scheme, spawning a small set of processes with multiple threads each. However we find that even in the extreme case of using single-thread processes, baseline MM performs at best similar to read and only if pre-faulting is applied (populate). DaxVM provides maximum performance (50% benefit) both with lightweight threads and on a hybrid configuration, eliminating paging costs and scalability bottlenecks.

Overall, we note that combining DaxVM's optimizations under a single PMem dedicated interface is essential as the techniques operate synergistically. For example, combining asynchronous unmapping with O(1) mmap (applicable only to PMem) boosts the effect of the former, as shootdowns emerge as a contention bottleneck.

Text search. We now examine an application that operates directly over small mapped files (via load/store instructions). We use the ag [9] search engine to search the Linux codebase for a string. The folder contains the source tree (68K files) and a few large files used for git versioning. Using MM access, the search engine maps a file, searches for the requested string and unmaps it, while with read it copies the file into a private buffer. Figure 9a shows that DaxVM outperforms baseline mmap interfaces and read by $\sim$70% at 16 cores. The application does not spend time copying data and DaxVM eliminates contention on the data access interface. Unlike Apache, asynchronous unmapping further boosts performance by 10%.

2) Large files and long-lived mappings: Our evaluation so far shows that DaxVM is beneficial for applications that issue frequent m(un)map operations to access data. We now examine how DaxVM affects applications already benefiting from MM access, operating over large files and for longer periods. For this set of workloads we only compare against baseline MM access.

Increasing availability with fast startup times. DaxVM's mmap can significantly increase the availability of applications that serve requests from memory-mapped files, as it enables O(1) access to the file data after reboot.

P-Redis [80] is a PMem-aware version of the Redis [8] in-memory key-value store from NVSL [6], [80]. It consists of a key-value cache and an index hash table, both in PMem. When the server is spawned, it maps both structures and uses loads/stores for access. Loading data for P-Redis involves populating the mappings' page tables. With baseline MM access this happens lazily during a warmup period when client requests trigger faults. Figure 9b shows throughput for the first 2M random get operations on a 60GB cache that stores 16KB values. Baseline mmap performance increases slowly (warm-up faults) while mmap-populate penalizes server start-up time by 10sec to pre-fault the cache pages and then provides high throughput. DaxVM gets the best of both worlds, achieving instant maximum throughput at no cost.

YCSB on Pmem-RocksDB. Finally we examine how DaxVM affects the performance of a database optimized to use PMem programming. Pmem-RocksDB [42] is Intel's PMem-optimized version of RocksDB [37] that mmaps SSTables/write-ahead logs (WALs) (placed on PMem) and writes directly to PMem using non-temporal stores (e.g., nt-store), omitting kernel sync operations [41]. It also recycles SSTables and WAL files whenever possible to control paging and zeroing overheads. We run YCSB workloads on a 50G dataset [46] and perform ∼12M operations (4KB records).

Ext4-dax results. Figure 9c summarizes our results. As discussed earlier, pre-faulting (populate) hurts performance of write/append intensive workloads (such as Load a, Load e and Run a). For the rest it performs close to default mmap.

DaxVM significantly improves performance for applications that perform insert operations (e.g., Load a, e). DaxVM dirty page tracking faults happen always at 2MB granularities (Section IV), irrespective to the file system's fragmentation. This significantly decreases the number of page-faults (10x less) improving performance by ∼2.3×. When we pre-zero in advance of running the workload (shown), performance is further boosted to ∼2.8x. With concurrent pre-zeroing, a 64MB/sec throttle reduces this by 5-10%. Finally, this version of RocksDB enforces durability from user-space [41] so we apply the *nosync* mode that brings performance to ∼2.95× compared to default mmap, eliminating faults entirely.

The main reason why DaxVM is so effective is that default mmap on an aged ext4-dax suffers from synchronous faults imposed by the MAP_SYNC interface [30] necessary to safely handle durability from user-space. On the first write fault on each mapped page the dirty file metadata (if any) will be synchronously flushed to storage. This triggers journaling transaction commits on ext4 that severely penalize scalability. On an aged FS, 4KB pages are involved on the mapping of a file, and thus such faults are more frequent. With DaxVM this happens always at 2MB granularities (less frequently) irrespective of FS fragmentation, restoring scalability to many cores. Note that on a fresh file system (100% huge page coverage) default mmap performs similarly.

DaxVM improves also performance by 1.46× for workload d (that also performs insertions) and 1.05-1.21× for the rest. All benefits come from fault elimination, as DaxVM's ephemeral allocator and asynchronous unmapping do not affect the long-lived and big file mappings of the workload.

NOVA results. For PMem-aware file systems that update metadata synchronously and in-place (such as NOVA) the MAP_SYNC interface becomes a no-op with zero overheads. We run the same experiments on a NOVA FS image and DaxVM's benefits for Load a and e are ∼35% compared to default mmap. For the rest of the workloads are ∼10%.

Comparison to default RocksDB [37]. This (Intel) optimized version of the key-value store provides ∼1.1×-2.1× benefit compared to the default version, when we run on a fresh FS image. When run on an aged ext4 image this benefit is penalized (as discussed before). DaxVM sustains up to 2× benefit even on the fragmented FS.

VI. Discussion: DaxVM beyond persistent memory

According to Intel's 2022 Q2 earning release [44], the company is winding down its Optane Memory business, which is a significant step back for persistent memory research. We do not consider this as the end of PMem storage design potential and discuss how DaxVM is relevant and beneficial for other fast storage technologies (despite being designed on Optane).

A. O(1) mmap and file tables

Byte-addressable storage and CXL. DaxVM is directly applicable to any byte-addressable storage technology; a design advocated by the emerging Compute Express Link (CXL [33]), e.g., Samsung has already announced a memory-semantic SSD that is CXL-compatible [67]. Any such storage solution, even PCIe and byte-addressable Flash NVMe combinations [10], is very close to PMem's philosophy and can benefit from DaxVM.

Microsecond-scale PCIe SSDs and direct access. State-of-the-art flash memory technologies have reduced storage-access latency to tens of microseconds [16], [63]. Such performance has exposed system storage stack as an important bottleneck and has questioned DRAM buffering as a necessary layer leading to various proposals for user-space direct access to storage [20], [50], [87]. Such solutions require rethinking and speeding up file system indexing [58] and even accelerating it in hardware [55] for performance and security reasons. DaxVM's FS file tables fall into this scope.

Memory-mapped buffered access. DaxVM's O(1) mmap and pre-populated file tables can be integrated as a page cache extension, to speedup traditional buffered storage access.

B. Address Space Scalability

DaxVM's ephemeral mappings and asynchronous unmappings are relevant to any memory access with ephemeral characteristics. This could apply both to direct or buffered memory-mapped storage access or even heap mappings. Memory tiering and fast storage rapidly change the usage of memory as a now common interface to multiple mediums with varying latencies. This imposes new challenges to address space management, questioning the state-of-practice.

VII. Related Work

User-space file systems. Multiple works [21], [34], [47], [54], [56], [76], [88] exploit PMem direct access via new file system (FS) designs with user-space components. Performing (meta)data operations directly from user-space avoids syscall overheads, but comes with two inherent challenges: (i) (meta)data security and (ii) concurrent file sharing. Mapping parts [21], [47], [54] or the entire FS image [56], [76] to user-space for large time frames opens a window for intentional attacks or unintentional errors (stray writes) that can leak data or corrupt the FS image [34]. Such FS must employ a mechanism to control this that may lead to scalability issues [54], [76]. In addition, many user-level FS do not support memory mapping [54], [76] at all. Kernel-space FS can be less performant but support seamlessly sharing and secure (meta)data operations. In this paper we focus on such well-tested mature FS targeting to improve the kernel's MM interface performance rather than bypass it.

File system indexing. HashFS [58] uses hashing instead of the commonly employed extent trees to accelerate software overheads of file indexing on the read/write system call path. ctFS [56] is a user-space file system that maps the entire DAX device in user-space to (de)allocate files contiguously in the virtual address layers, similar to SCMFS [79]. It then uses page tables to index files quickly. Exposing the entire device to user-space raises significant security concerns acknowledged by the authors. DaxVM (de)attaches file tables directly to address spaces, primarily to eliminate the paging costs of MM access. By doing so, it entirely removes software file indexing from the MM path. We discuss other works that employ file system maintained page tables on Section III.

Address space scalability. Past studies [24], [25] of address space scalability target generic solutions (e.g., range locking or concurrent lock-free data structures in VM) that apply to all memory regions. However, the Linux community has been discussing such radical changes for many years [29] and relevant implementations [31], [66] show that the transition is not that easy in terms of performance [52] or complexity. A key insight of DaxVM is that one can exploit the special lifetime and access characteristics of PMem mappings to provide a much simpler dedicated address space (de)allocator that can scale to many cores (ephemeral mappings).

Fast unmap. LATR [53] proposes message passing – a generic radical re-design of the TLB invalidation mechanism to enable lazy invalidations. DaxVM exploits batched unmapping requests, adopting a dedicated design already present for the IOMMU and traditional storage DMA mappings [62]. The key insight is that opting for a dedicated design for targeted uses can enable higher performance at a much lower complexity. Numerous proposals for faster/simpler delivery of shootdowns in hardware [75], [84] and software [17], or for more accurate shootdowns [13], [14] would reduce the need for DaxVM's asynchronous unmapping. Boyd-Wickizer et al. [18] examine per-thread address private ranges to avoid synchronization and TLB shootdowns.

Pre-zeroing. Hawkey [59] and Trident [64] examine asynchronous pre-zeroing for huge volatile page allocation latency. DaxVM exposes its necessity for PMem file mappings and integrates it in a file system. Our key insight is that block zeroing is an inherently different requirement among DAX interfaces (MM access vs system calls) that if not managed can flip performance trends.

Faster paging: Previous works underline the cost of paging and particularly of faults for PMem direct access [47], [80], [83]. They propose huge page usage [47], [80], caching per-process file mappings [23], and O(1) memory [72] on a conceptual or emulated level (more in Section III) DaxVM expands on this work with a real implementation of O(1) mmap in Linux and on unmodified hardware, reduces DRAM consumption by placing file tables in PMem, and avoids dependence on huge pages. WineFS [46] is a new huge-page aware FS for high huge page coverage. DaxVM is complementary to huge pages, supporting them when available, but resilient to fragmentation in terms of paging. Prior work on sharing page tables focused on speeding fork [35] and removing duplicate TLB entries [69].

Song et al. [70], [71] focus on faster page reclamation and batching shootdowns under memory pressure. Papagiannis et al. [61] propose an mmap design that ignores DAX, targeting page cache optimizations. DaxVM focuses on DAX mappings that are neither subject to memory pressure nor use a cache.

PMem file systems: Many research projects focus on faster file systems for PMem, and mostly look at (i) avoiding the page cache like DAX [27], [36], (ii) providing faster metadata operations with fine-grained persistence [27], [51], [79], [81], [82], and (iii) moving kernel operations to user-space (discussed before).

VIII. Conclusions

Byte-addressable high performance storage and the DAX-mmap interface provide the shortest path to persistent data, mapping it into process address spaces. Yet, the high overheads of virtual memory operations involved in file mapping (e.g., paging, VM lock contention and TLB coherence) can be prohibitive to adopt the interface. We analyze these costs and propose DaxVM, an optimized POSIX-relaxed interface that provides fast and scalable storage access via (i) O(1) mmap, (ii) ephemeral mappings, (iii) asynchronous unmappings, (iv) asynchronous storage pre-zeroing, and (v) coarse-grain or zero kernel-space durability management.

Acknowledgements

We thank our anonymous reviewers for their insightful comments and feedback on the paper. We thank Sujay Yadalam and the SCAIL research group of UW Madison for their feedback. Similarly, we thank Nectarios Koziris, Dionisios Pnevmatikatos, Stratos Psomadakis, Christos Katsakioris and the systems research group of the CSLab at NTUA. Finally, we thank Dimitris Gizopoulos for his decisive comments. This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957407 (project DAPHNE). This work was also funded by NSF grants CNS 1815656 and CNS 1900758.

References

[1] "Apache HTTP Server project." https://httpd.apache.org/.

[2] "Benchmark multiple url paths with wrk." https://gist.github.com/xydinesh/28bd6ac7de1d45b61a9d896e3442248d.

[3] "Direct Access for files," https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[4] "ext4: Use page_mkwrite vma_operations to get mmap write notification." https://linux-ext4.vger.kernel.narkive.com/kplEwAhG/patch-ext4-use-page-mkwrite-vma-operations-to-get-mmap-write-notification.

[5] "Modern HTTP benchmarking tool." https://github.com/wg/wrk.

[6] "Non-volatile systems laboratory," https://nvsl.io/.

[7] "Persistent Memory File System." https://github.com/linux-pmfs/pmfs.

[8] "Redis: an in-memory data structure store," https://redis.io/.

[9] "The Silver Searcher – A Code Searching Tool for Programmers," https://www.tecmint.com/the-silver-searcher-a-code-searching-tool-for-linux/.

[10] A. Abulila, V. S. Mailthody, Z. Qureshi, J. Huang, N. S. Kim, J. Xiong, and W.-m. Hwu, "Flatflash: Exploiting the byte-accessibility of ssds within a unified memory-storage hierarchy," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19, 2019. [Online]. Available: https://doi.org/10.1145/3297858.3304061

[11] R. Achermann, A. Panwar, A. Bhattacharjee, T. Roscoe, and J. Gandhi, *Mitosis: Transparently Self-Replicating Page-Tables for Large-Memory Machines*. New York, NY, USA: Association for Computing Machinery, 2020, p. 283–300. [Online]. Available: https://doi.org/10.1145/3373376.3378468

[12] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch, "A Five-Year study of File-System metadata," in *5th USENIX Conference on File and Storage Technologies (FAST 07)*. San Jose, CA: USENIX Association, Feb. 2007. [Online]. Available: https://www.usenix.org/conference/fast-07/five-year-study-file-system-metadata

[13] N. Amit, "Optimizing the tlb shootdown algorithm with page access tracking," in *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '17. USA: USENIX Association, 2017, p. 27–39.

[14] N. Amit, A. Tai, and M. Wei, "Don't shoot down tlb shootdowns!" in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3342195.3387518

[15] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout, "Measurements of a distributed file system," *ACM SIGOPS Operating Systems Review*, vol. 25, apr 2000.

[16] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan, "Attack of the killer microseconds," *Commun. ACM*, vol. 60, no. 4, p. 48–54, mar 2017. [Online]. Available: https://doi.org/10.1145/3015146

[17] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: A new os architecture for scalable multicore systems," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09. Association for Computing Machinery, 2009, p. 29–44. [Online]. Available: https://doi.org/10.1145/1629575.1629579

[18] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang, "Corey: An operating system for many cores," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. USA: USENIX Association, 2008, p. 43–57.

[19] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich, "An analysis of linux scalability to many cores," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. USA: USENIX Association, 2010, p. 1–16.

[20] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson, "Providing safe, user space access to fast, solid state disks," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*, T. Harris and M. L. Scott, Eds. ACM, 2012, pp. 387–400. [Online]. Available: https://doi.org/10.1145/2150976.2151017

[21] Y. Chen, Y. Lu, B. Zhu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and J. Shu, "Scalable persistent memory file system with kernel-userspace collaboration," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, Feb. 2021, pp. 81–95. [Online]. Available: https://www.usenix.org/conference/fast21/presentation/chen-youmin

[22] D. Chinner, "xfs: DAX support." https://lwn.net/Articles/635514/, 2015.

[23] J. Choi, J. Kim, and H. Han, "Efficient memory mapped file i/o for in-memory file systems," in *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*. Santa Clara, CA: USENIX Association, Jul. 2017. [Online]. Available: https://www.usenix.org/conference/hotstorage17/program/presentation/choi

[24] A. T. Clements, M. F. Kaashoek, and N. Zeldovich, "Scalable address spaces using rcu balanced trees," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. Association for Computing Machinery, 2012, p. 199–210. [Online]. Available: https://doi.org/10.1145/2150976.2150998

[25] A. T. Clements, M. F. Kaashoek, and N. Zeldovich, "Radixvm: Scalable address spaces for multithreaded applications," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13. Association for

Computing Machinery, 2013, p. 211–224. [Online]. Available: https://doi.org/10.1145/2465351.2465373

[26] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler, "The scalable commutativity rule: Designing scalable software for multicore processors," *ACM Trans. Comput. Syst.*, vol. 32, no. 4, Jan. 2015. [Online]. Available: https://doi.org/10.1145/2699681

[27] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09. Association for Computing Machinery, 2009, p. 133–146. [Online]. Available: https://doi.org/10.1145/1629575.1629589

[28] J. Corbet, " Memory management locking," https://lwn.net/Articles/591978/, 2014.

[29] J. Corbet, "Memory-management scalability," https://lwn.net/Articles/636334/, 2015.

[30] J. Corbet, "Two more approaches to persistent-memory writes," https://lwn.net/Articles/731706/, 2017.

[31] J. Corbet, "How to get rid of mmap_sem," https://lwn.net/Articles/787629/, 2019.

[32] A. Crotty, V. Leis, and A. Pavlo, "Are you sure you want to use mmap in your database management system?" in *CIDR 2022, Conference on Innovative Data Systems Research*, 2022. [Online]. Available: https://db.cs.cmu.edu/papers/2022/p13-crotty.pdf

[33] CXL_Consortium, "Compute express link specification revision 2.0." https://www.computeexpresslink.org/download-the-specification.

[34] M. Dong, H. Bu, J. Yi, B. Dong, and H. Chen, "Performance and protection in the zofs user-space nvm file system," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 478–493. [Online]. Available: https://doi.org/10.1145/3341301.3359637

[35] X. Dong, S. Dwarkadas, and A. L. Cox, "Shared address translation revisited," in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16. Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2901318.2901327

[36] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys '14. Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2592798.2592814

[37] Facebook., "RocksDB." http://rocksdb.org., 2017.

[38] A. Fedorova, "Why mmap is faster than system calls," https://sasha-f.medium.com/why-mmap-is-faster-than-system-calls-24718e75ab37.

[39] J. Huang, A. Badam, M. K. Qureshi, and K. Schwan, "Unified address translation for memory-mapped ssds with flashmap," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. Association for Computing Machinery, 2015, p. 580–591. [Online]. Available: https://doi.org/10.1145/2749469.2750420

[40] Intel, "AVX-512 instructions." https://software.intel.com/content/www/us/en/develop/articles/intel-avx-512-instructions.html.

[41] Intel, "How Intel Optimized RocksDB Code for Persistent Memory with PMDK." https://www.intel.com/content/www/us/en/developer/articles/technical/how-intel-optimized-rocksdb-code-for-persistent-memory-with-pmdk.html.

[42] Intel, "Pmem-RocksDB." https://github.com/pmem/pmem-rocksdb.

[43] Intel, "Intel(R) Optane(TM) DC Persistent Memory." https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html, 2019.

[44] Intel, "Intel reports second-quarter 2022 financial results," https://www.intel.com/content/www/us/en/newsroom/news/intel-reports-q2-2022-financial-results.html, 2022.

[45] Intel Corporation, " Intel® 64 and IA-32 Architectures Software Developer Manuals," https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html.

[46] R. Kadedodi, S. Kadekodi, S. Ponnapalli, H. Shirwadkar, G. Ganger, A. Kolli, and V. Chidambaram, "WineFS: a hugepage-aware file system for persistent memory that ages gracefully," in *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP '21)*, October 2021.

[47] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram, "Splitfs: Reducing software overhead in file systems for persistent memory," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. Association for Computing Machinery, 2019, p. 494–508. [Online]. Available: https://doi.org/10.1145/3341301.3359631

[48] S. Kadekodi, V. Nagarajan, and G. R. Ganger, "Geriatrix: Aging what you see and what you don't see. a file system aging approach for modern storage systems," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018, pp. 691–704. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/kadekodi

[49] W. Kang, D. Shin, and S. Yoo, "Reinforcement learning-assisted garbage collection to mitigate long-tail latency in ssd," *ACM Transactions on Embedded Computing Systems*, vol. 16, no. 5s, sep 2017. [Online]. Available: https://doi.org/10.1145/3126537

[50] S. Kannan, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, Y. Wang, J. Xu, and G. Palani, "Designing a true Direct-Access file system with DevFS," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*. Oakland, CA: USENIX Association, Feb. 2018, pp. 241–256. [Online]. Available: https://www.usenix.org/conference/fast18/presentation/kannan

[51] J.-H. Kim, J. Kim, H. Kang, C.-G. Lee, S. Park, and Y. Kim, "Pnova: Optimizing shared file i/o operations of nvm file system on manycore servers," in *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, ser. APSys '19. Association for Computing Machinery, 2019, p. 1–7. [Online]. Available: https://doi.org/10.1145/3343737.3343748

[52] A. Kogan, D. Dice, and S. Issa, "Scalable range locks for scalable address spaces and beyond," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3342195.3387533

[53] M. K. Kumar, S. Maass, S. Kashyap, J. Veselý, Z. Yan, T. Kim, A. Bhattacharjee, and T. Krishna, "Latr: Lazy translation coherence," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. Association for Computing Machinery, 2018, p. 651–664. [Online]. Available: https://doi.org/10.1145/3173162.3173198

[54] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson, "Strata: A cross media file system," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. Association for Computing Machinery, 2017, p. 460–477. [Online]. Available: https://doi.org/10.1145/3132747.3132770

[55] G. Lee, W. Jin, W. Song, J. Gong, J. Bae, T. J. Ham, J. W. Lee, and J. Jeong, "A case for hardware-based demand paging," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 1103–1116.

[56] R. Li, X. Ren, X. Zhao, S. He, M. Stumm, and D. Yuan, "ctFS: Replacing file indexing with hardware memory translation through contiguous file allocation for persistent memory," in *20th USENIX Conference on File and Storage Technologies (FAST 22)*. Santa Clara, CA: USENIX Association, Feb. 2022, pp. 35–50. [Online]. Available: https://www.usenix.org/conference/fast22/presentation/li

[57] C. Min, S. Kashyap, S. Maass, and T. Kim, "Understanding manycore scalability of file systems," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, Jun. 2016, pp. 71–85. [Online]. Available: https://www.usenix.org/conference/atc16/technical-sessions/presentation/min

[58] I. Neal, G. Zuo, E. Shiple, T. A. Khan, Y. Kwon, S. Peter, and B. Kasikci, "Rethinking file mapping for persistent memory," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, Feb. 2021, pp. 97–111. [Online]. Available: https://www.usenix.org/conference/fast21/presentation/neal

[59] A. Panwar, S. Bansal, and K. Gopinath, "Hawkeye: Efficient fine-grained os support for huge pages," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3297858.3304064

[60] A. Papagiannis, M. Marazakis, and A. Bilas, "Memory-mapped i/o on steroids," in *Proceedings of the Sixteenth European Conference on Computer Systems*, ser. EuroSys '21. Association for Computing Machinery, 2021, p. 277–293. [Online]. Available: https://doi.org/10.1145/3447786.3456242

[61] A. Papagiannis, G. Xanthakis, G. Saloustros, M. Marazakis, and A. Bilas, "Optimizing memory-mapped i/o for fast storage devices," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 813–827. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/papagiannis

[62] O. Peleg, A. Morrison, B. Serebrin, and D. Tsafrir, "Utilizing the IOMMU scalably," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. Santa Clara, CA: USENIX Association, Jul. 2015, pp. 549–562. [Online]. Available: https://www.usenix.org/conference/atc15/technical-session/presentation/peleg

[63] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The operating system is the control plane," *ACM Trans. Comput. Syst.*, nov 2015.

[64] V. S. S. Ram, A. Panwar, and A. Basu, "Trident: Harnessing architectural resources for all page sizes in x86 processors," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1106–1120. [Online]. Available: https://doi.org/10.1145/3466752.3480062

[65] G. Rodrigues, " ile holes, races, and mmap()," https://lwn.net/Articles/357767/, 2009.

[66] M. Rybczyńska, "Introducing maple trees," https://lwn.net/Articles/845507/, 2021.

[67] Samsung, "Memory-semantic ssd," https://news.samsung.com/global/samsung-electronics-unveils-far-reaching-next-generation-memory-solutions-at-flash-memory-summit-2022, 2022.

[68] E. H.-M. Sha, X. Chen, Q. Zhuge, L. Shi, and W. Jiang, "A new design of in-memory file system based on file virtual address framework," *IEEE Transactions on Computers*, vol. 65, no. 10, pp. 2959–2972, 2016.

[69] D. Skarlatos, U. Darbaz, B. Gopireddy, N. S. Kim, and J. Torrellas, "Babelfish: Fusing address translations for containers," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 501–514.

[70] N. Y. Song, Y. J. Yu, W. Shin, H. Eom, and H. Y. Yeom, "Low-latency memory-mapped i/o for data-intensive applications on fast storage devices," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, 2012, pp. 766–770.

[71] N. Y. Song, Y. Son, H. Han, and H. Y. Yeom, "Efficient memory-mapped i/o on fast storage device," *ACM Trans. Storage*, vol. 12, no. 4, May 2016. [Online]. Available: https://doi.org/10.1145/2846100

[72] M. M. Swift, "Towards o(1) memory," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, ser. HotOS '17. Association for Computing Machinery, 2017, p. 7–11. [Online]. Available: https://doi.org/10.1145/3102980.3102982

[73] V. Tarasov, E. Zadok, and S. Shepler, "Filebench: A flexible framework for file system benchmarking," *login Usenix Mag.*, vol. 41, no. 1, 2016. [Online]. Available: https://www.usenix.org/publications/login/spring2016/tarasov

[74] L. Torvalds, " Linux kernel mailing list: mmap/mlock performance versus read," https://marc.info/?l=linux-kernel&m=95496636207616&w=2, 2000.

[75] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal, "Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory," in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '11. USA: IEEE Computer Society, 2011, p. 340–349. [Online]. Available: https://doi.org/10.1109/PACT.2011.65

[76] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift, "Aerie: Flexible file-system interfaces to storage-class memory," in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys '14. Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2592798.2592810

[77] Z. Wang, X. Liu, J. Yang, T. Michailidis, S. Swanson, and J. Zhao, "Characterizing and modeling non-volatile memory systems," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 496–508.

[78] M. Wilcox, "Add support for NV-DIMMs to ext4." https://lwn.net/Articles/613384/, 2014.

[79] X. Wu, S. Qiu, and A. L. Narasimha Reddy, "Scmfs: A file system for storage class memory and its extensions," *ACM Trans. Storage*, vol. 9, no. 3, Aug. 2013. [Online]. Available: https://doi.org/10.1145/2501620.2501621

[80] J. Xu, J. Kim, A. Memaripour, and S. Swanson, "Finding and fixing performance pathologies in persistent memory software stacks," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. Association for Computing Machinery, 2019, p. 427–439. [Online]. Available: https://doi.org/10.1145/3297858.3304077

[81] J. Xu and S. Swanson, "Nova: A log-structured file system for hybrid volatile/non-volatile main memories," in *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, ser. FAST'16. USA: USENIX Association, 2016, p. 323–338.

[82] J. Xu, L. Zhang, A. Memaripour, A. Gangadharaiah, A. Borase, T. B. Da Silva, S. Swanson, and A. Rudoff, "Nova-fortis: A fault-tolerant non-volatile main memory file system," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. Association for Computing Machinery, 2017, p. 478–496. [Online]. Available: https://doi.org/10.1145/3132747.3132761

[83] Y. Xu, Y. Solihin, and X. Shen, "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*, ser. ASPLOS '20. Association for Computing Machinery, 2020, p. 987–1000. [Online]. Available: https://doi.org/10.1145/3373376.3378492

[84] Z. Yan, J. Veselý, G. Cox, and A. Bhattacharjee, "Hardware translation coherence for virtualized systems," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. Association for Computing Machinery, 2017, p. 430–443. [Online]. Available: https://doi.org/10.1145/3079856.3080211

[85] J. Yang, J. Izraelevitz, and S. Swanson, "Orion: A distributed file system for non-volatile main memory and rdma-capable networks," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 221–234. [Online]. Available: https://www.usenix.org/conference/fast19/presentation/yang

[86] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 169–182. [Online]. Available: https://www.usenix.org/conference/fast20/presentation/yang

[87] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul, "Spdk: A development kit to build high performance storage applications," in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2017, pp. 154–161.

[88] T. Yoshimura, T. Chiba, and H. Horii, "Evfs: User-level, event-driven file system for non-volatile memory," in *Proceedings of the 11th USENIX Conference on Hot Topics in Storage and File Systems*, ser. HotStorage'19. USA: USENIX Association, 2019, p. 16.

[89] P. Zijlstra, "Tracking shared dirty pages." https://lwn.net/Articles/185463/, 2006.