Rethinking File Mapping for Persistent Memory

Ian Neal, Gefei Zuo, Eric Shiple, and Tanvir Ahmed Khan, *University of Michigan;* Youngjin Kwon, *School of Computing, KAIST;* Simon Peter, *University of Texas at Austin;* Baris Kasikci, *University of Michigan*

https://www.usenix.org/conference/fast21/presentation/neal

This paper is included in the Proceedings of the
19th USENIX Conference on File and Storage Technologies.

February 23–25, 2021

978-1-939133-20-5

Open access to the Proceedings
of the 19th USENIX Conference on
File and Storage Technologies
is sponsored by USENIX.

Rethinking File Mapping for Persistent Memory

Ian Neal
University of Michigan

Gefei Zuo
University of Michigan

Eric Shiple
University of Michigan

Tanvir Ahmed Khan
University of Michigan

Youngjin Kwon
School of Computing, KAIST

Simon Peter
University of Texas at Austin

Baris Kasikci
University of Michigan

Abstract

Persistent main memory (PM) dramatically improves IO performance. We find that this results in file systems on PM spending as much as 70% of the IO path performing *file mapping* (mapping file offsets to physical locations on storage media) on real workloads. However, even PM-optimized file systems perform file mapping based on decades-old assumptions. It is now critical to revisit file mapping for PM.

We explore the design space for PM file mapping by building and evaluating several file-mapping designs, including different data structure, caching, as well as meta-data and block allocation approaches, within the context of a PM-optimized file system. Based on our findings, we design HashFS, a hash-based file mapping approach. HashFS uses a single hash operation for all mapping and allocation operations, bypassing the file system cache, instead prefetching mappings via SIMD parallelism and caching translations explicitly. HashFS's resulting low latency provides superior performance compared to alternatives. HashFS increases the throughput of YCSB on LevelDB by up to 45% over page-cached extent trees in the state-of-the-art Strata PM-optimized file system.

1 Introduction

Persistent main memory (PM, also known as non-volatile main memory or NVM) is a new storage technology that bridges the gap between traditionally-slow storage devices (SSD, HDD) and fast, volatile random-access memories (DRAM). Intel Optane DC persistent memory modules [20], along with other PM variants [1,2,4,30,35,50], are anticipated to become commonplace in DIMM slots alongside traditional DRAM. PM offers byte-addressable persistence with only 2–3× higher latency than DRAM, which is highly appealing to file-system designers. Indeed, prior work has re-designed many file-system components specifically for PM, reducing overhead in the IO path (e.g. by eliminating memory copies and bypassing the kernel) [11, 14, 24, 29, 48, 52, 56].

However, not all aspects of the IO path have been examined in detail. Surprisingly, *file mapping* has received little attention in PM-optimized file systems to date. *File mapping*—the translation from logical file offsets to physical locations on the underlying device—comprises up to 70% of the IO path of real workloads in a PM-optimized file system (as we show in §4.8). Even for memory-mapped files, file mapping is still involved in file appends. Yet, existing PM-optimized file systems either simply reuse mapping approaches [11, 14, 24] originally designed for slower block devices [29], or devise new approaches without rigorous analysis [15, 57].

PM presents a number of challenges to consider for file mapping, such as dealing with fragmentation and concurrency problems. Notably, the near-DRAM latency of PM requires reconsidering many aspects of file mapping, such as mapping structure layout, the role of the page cache in maintaining volatile copies of file mapping structures, and the overhead of physical space allocation. For example, some PM-optimized file systems maintain copies of their block mapping structures in a page cache in DRAM, as is traditional for file systems on slower storage devices [11, 24, 29], but others do not [14, 15, 48] and instead rely on CPU caching. It is currently unclear if maintaining a volatile copy of mapping structures in DRAM is beneficial versus relying on CPU caches to cache mapping structures, or designing specialized file map caches.

In this work, we build and rigorously evaluate a range of file mapping approaches within the context of PM-optimized file systems. PM's random access byte addressability makes designing fully random access mapping structures (i.e., hash tables) possible, which expands the design space of possible file mapping structures. To this end, we evaluate and PM-optimize two classic *per-file* mapping approaches (i.e., where each file has its own mapping structure) that use extent trees and radix trees. We propose and evaluate two further, *global* mapping approaches (i.e., where a single structure maps all files in the entire file system). The first uses cuckoo hashing, the second is *HashFS*, a combined global hash table and block allocation approach that uses linear probing. We use these approaches to evaluate a range of design points, from an emphasis on sequential access performance to an emphasis on minimizing memory references, cached and non-cached, as well as the role of physical space allocation.

We evaluate these file-mapping approaches on a series of benchmarks that test file system IO latency and throughput under different workloads, access and storage patterns, IO sizes, and structure sizes. We show how the usage of

PM-optimized file mapping in a PM-optimized file system, Strata [29], leads to a significant reduction in the latency of file IO operations and large gains in application throughput. An evaluation of Strata on YCSB [10] workloads running on LevelDB [17] shows up to 45% improvement in throughput with PM-optimized file mapping (§4). Our analysis indicates that the performance of file-mapping is file system independent, allowing our designs to be applied to other PM file systems.

Prior work investigates the design of PM-optimized *storage structures* [31, 36, 54, 59] and indexing structures for PM-optimized key-value stores [28, 51, 58]. These structures operate on top of memory mapped files and rely on file mapping to abstract from physical memory concerns. Further, PM-optimized storage and indexing structures are designed with per-structure consistency (e.g., shadow paging) [31, 54, 59]. File systems already provide consistency across several metadata structures, making per-structure methods redundant and expensive. In our evaluation, we show that storage structures perform poorly for file mapping (§4.9).

In summary, we make the following contributions:

- We present the design and implementation of four file mapping approaches and explore various PM optimizations to them (§3). We implement these approaches in the Strata file system [29], which provides state-of-the-art PM performance, particularly for small, random IO operations, where file mapping matters most (§4.4). We perform extensive experimental analyses on how the page cache, mapping structure size, IO size, storage device utilization, fragmentation, isolation mechanisms, and file access patterns affect the performance of these file mapping approaches (§4).
- We show that our PM-optimized HashFS is the best performing mapping approach, due to its low memory overhead and low latency (§4.1) during random reads and insertions. We demonstrate that this approach can increase throughput by up to 45% over Strata's page-cached extent trees in YCSB workloads (§4.8).
- We show that using a traditional page cache provides no benefit for PM-optimized file mapping (§4.7). We also demonstrate that PM-optimized storage structures are ill-suited for use as file mapping structures and cannot be simply dropped into PM-optimized file systems (§4.9).

2 File Mapping Background

What is file mapping? *File mapping* is the operation of mapping a logical offset in a file to a physical location on the underlying device. File mapping is made possible by one or more metadata structures (*file mapping structures*) maintained by the file system. These file mapping structures map logical locations (a file and offset) to physical locations (a device offset) at a fixed granularity. File mapping structures have three operations: *lookups*, caused by file reads and writes of existing locations; *insertions*, caused by file appends or IO to unallocated areas of a sparse file; and *deletions*, caused by file truncation (including file deletion) or region de-allocation in a sparse file. Inserts and deletions require the (de-)allocation of physical locations, typically provided by a *block allocator* maintaining a separate free block structure. PM file systems generally map files at block granularity of at least 4KB in order to constrain the amount of metadata required to track file system space [15, 29, 52]. Block sizes often correspond to memory page sizes, allowing block-based PM file systems to support memory-mapped IO more efficiently.

2.1 File Mapping Challenges

The following properties make designing efficient file mapping challenging. Careful consideration must be taken to find points in the trade-off space presented by these properties that maximize performance for a given workload and storage device. PM intensifies the impact of many of these properties.

Fragmentation. Fragmentation occurs when a file's data is spread across non-contiguous *physical locations* on a storage device. This happens when a file cannot be allocated in a single contiguous region of the device, or when the file grows and all adjacent blocks have been allocated to other files. Fragmentation is inevitable as file systems age [42] and can occur rapidly [9, 23].

Fragmentation can magnify the overhead of certain file mapping designs. Many traditional file systems use compact file mapping structures, like extent trees (§3.1), which can use a single *extent entry* to map a large range of contiguous file locations. Fragmentation causes locations to become non-contiguous and the mapping structure to become larger, which can increase search and insert times (§4). This is referred to as *logical fragmentation* and is a major source of overhead in file systems [19], especially on PM (as file mapping constitutes a larger fraction of the IO path on PM file systems). Another consequence of fragmentation is the reduction in sequential accesses. Fragmentation results in the separation of data structure locations across the device, which causes additional, random IO on the device. This degrades IO performance, particularly on PM devices [22].

Fragmentation must be considered as an inevitable occurrence during file mapping, since defragmentation is not always feasible or desirable. Defragmentation is an expensive operation—it can incur many file writes, which can lower device lifespan by more than 10% in the case of SSDs [19]. Similar concerns exist for PM, which may make defragmentation undesirable for PM-optimized file systems as well [18, 59].

Locality of reference. Locality of reference when accessing a file can be used to reduce file mapping overhead. Accesses with locality are typically accelerated by caching prior accesses and prefetching adjacent ones. OS page caches and CPU caches can achieve this transparently and we discuss the role of the OS page cache for PM file mapping in §2.2. However, approaches specific to file mapping can yield fur-

ther benefits. For example, we can hide part of the file mapping structure traversal overhead for accesses with locality, by remembering the meta-data location of a prior lookup and prefetching the location of the next lookup. Further, with an efficient cache in place, the file mapping structure itself should be optimized for random (i.e., non-local) lookups, as the structure is referenced primarily for uncached mappings.

Mapping structure size. The aggregate size of the file mapping structures as a fraction of available file system space is an important metric of file mapping efficiency. Ideally, a file mapping structure consumes a small fraction of available space, leaving room for actual file data storage. Furthermore, the size of the mapping structure impacts the amount of data that can remain cache-resident.

Traditional file mapping structures are designed to be *elastic*—the size of the structure is proportional to the number of locations allocated in the file system. This means that as the number and size of files increase, the size of the file mapping structure grows as well. However, elastic mapping structures introduce overhead by requiring resizing, which incurs associated space (de-)allocation and management cost.

Concurrency. Providing isolation (ensuring that concurrent modifications do not affect consistency) can be an expensive operation with limited scalability. This is a well-known problem for database mapping structures (called *indexes*) that support operations on arbitrary ranges. For example, insertions and deletions of tree-based range index entries may require updates to inner tree nodes that contend with operations on unrelated keys [16, 49].

Isolation is simpler in *per-file* mapping structures, where the variety and distribution of operations that can occur concurrently is limited. With the exception of sparse files, updates only occur during a change in file size, i.e. when appending new blocks or truncating to remove blocks. File reads and in-place writes (writes to already allocated blocks) only incur file mapping structure reads. For this reason, it is often sufficient to protect per-file mapping structures with a coarse-grained reader-writer lock that covers the entire structure [29].

The most common scenario for per-file mapping structure concurrent access is presented by workloads with one writer (updating the file mapping structure via appends or truncates), with concurrent readers (file reads and writes to existing offsets). In this case, file mapping reads and writes can proceed without contention in the common case. Other mapping structure operations can impact concurrency, but they generally occur infrequently. For example, some file mapping structures require occasional resizing. For consistency, this operation occurs within a critical section but is required only when the structure grows beyond a threshold. Extent trees may split or merge extents and these operations require partial-tree locking (locking inner nodes) but occur only on updates.

For *global* file mapping structures, the possibility of contention is higher, as there can be concurrent writers (appending/truncating different files). Global file mapping structures have to be designed with concurrency in mind. In this case, contention does not favor the use of tree-based indices, but is tractable for hash table structures with per-bucket locks, as contention among writers only occurs upon hash collision.

2.2 File Mapping Non-Challenges

Crash consistency. An overarching concern in file systems is providing consistency—transitioning metadata structures from one valid state to another, even across system crashes. File systems have many metadata structures that often need to be updated atomically across crashes, i.e. a free-list and a file-mapping structure when a new location is allocated for a file. File systems generally employ some form of journaling to ensure these updates can be replayed and made atomic and consistent, even in the case of a crash. This makes crash consistency a non-challenge for file mapping. This is unlike PM persistent data structures, which are typically designed to provide crash consistency within the structure itself.

Page caching. Traditionally, file systems read data and metadata (including file-mapping structures) into a *page cache* in DRAM before serving it to the user. The file system batches updates using this page cache and writes back dirty pages. This is a necessary optimization to reduce the overhead of reading directly from a block device for each IO operation. However, page caches have overheads. A pool of DRAM pages must be managed and reallocated to new files as they are opened, and the pages in the cache must be read and written back to ensure updates are consistent.

For PM, an OS-managed page cache in DRAM may no longer be required for file mapping structures. One analysis [48] found that eschewing the page cache for inodes and directories results in better performance in PM-optimized file systems. Until now, there has been no such consensus on the optimal design point for file mapping structures: some PM-file systems maintain file mapping structures only in PM [15], others maintain file mapping structures only in DRAM [52, 56], still others manage a cache of PM-based file mapping structures in DRAM [11, 29]. However, as we show in §4.7, page-cache management turns out to be a non-challenge, as it is always more efficient to bypass page-caching for file-mapping structures on PM-optimized file systems.

3 PM File Mapping Design

Based on our discussion of the challenges (§2.1), we now describe the design of four PM-optimized file mapping approaches, which we analyze in §4. We first describe two traditional, per-file mapping approaches and their PM optimizations, followed by two global mapping approaches. We discuss the unique challenges faced by each approach, followed by a description of the approach's mapping structure.

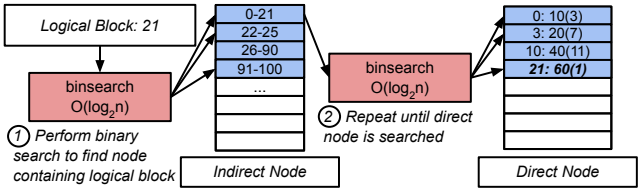

Figure 1: An extent tree lookup example. Each indirect node lists the range of logical blocks in the direct nodes it points to (shown as `<start block-end block>`). Each direct block contains *extents* that map ranges of logical blocks to physical blocks, represented as `<logical block start: physical block start (number of blocks)>`. Traversal of nodes is repeated until the direct mapping node is found.

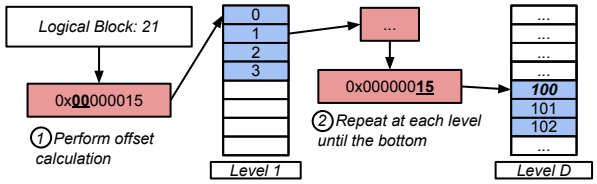

Figure 2: An example radix tree performing a lookup. Each level is indexed by using a portion of the logical block number as an offset. The last level offset contains a single physical block number.

3.1 Traditional, Per-File Mapping

File mapping is done *per-file* in most traditional file systems. Each file has its own elastic mapping structure which grows and shrinks with the number of mappings the structure needs to maintain. The mapping structure is found via a separate, fixed-size inode table, indexed by a file's *inode number*. Per-file mapping generally provides high spatial locality, as all mappings for a single file are grouped together in a single structure, which is traditionally cached in the OS page cache. Adjacent logical blocks are often represented as adjacent mappings in the mapping structure, leading to more efficient sequential file operations. Concurrent access is a minor problem for per-file mapping approaches due to the restricted per-file access pattern (§2.1). Physical block allocation is performed via a separate block allocator, which is implemented using a red-black tree in our testbed file system (§4).

Challenges. Per-file mapping structures must support resizing, which can be an expensive operation. Since they need to grow and shrink, per-file mapping structures have multiple levels of indirection, requiring more memory references for each mapping operation. Fragmentation destroys the compact layout of these structures and exacerbates overhead by increasing the amount of indirection and thus memory accesses.

We now discuss two common mapping structures used for per-file mapping.

3.1.1 Extent Trees

The extent tree is a classic per-file mapping structure, but is still used in modern PM-optimized file systems [11, 29]. Extent trees are B-trees that contain *extents*, which map file logical blocks to physical blocks, along with a field that indicates the number of blocks the mapping represents. Extents can also be indirect, pointing to further extent tree nodes rather than to file data blocks. In order to perform a lookup, a binary search is performed on each node to determine which entry holds the desired physical block. This search is performed on each level of the extent tree, as shown in Fig. 1.

We create a PM-optimized variant of extent trees, based on the implementation from the Linux kernel [33]. Traditionally, extent tree operations are performed on copies of tree blocks in the page cache. As using a page cache for PM mapping structures leads to unnecessary copy overhead (see §4.7), we instead design our extent tree to operate directly on PM. Consistency under concurrent extent access is guaranteed by a per-extent entry valid bit, which is only enabled once an extent is available to be read. Consistency for complex operations (resizing, splitting extents, etc.) is provided by an undo log. We also keep a cursor [5] in DRAM of the last accessed path, improving performance for accesses with locality.

Design considerations. Extent trees have the most compact representation of any of the mapping structures that we evaluate, since multiple mappings can be coalesced into a single extent, leading to small structure size. Extent trees, however, require many memory accesses, as they must perform a binary search at each level of the extent tree to find the final mapping. Cursors can simplify the search, but only for sequential scans through the structure and repeat block accesses.

3.1.2 Radix Trees

Radix trees are another popular per-file mapping structure [15], shown in Fig. 2. Each node in a radix tree occupies a physical block on the device (typically 4KB), with the exception of a few entries that can be stored within the inode directly. A lookup starts at the top level node resolving the top N bits of the logical block number. With 8 byte pointers and 4KB per node, we can resolve $N = log_2(\frac{4096}{8}) = 9$ bits per radix tree node. The second level node resolves the next N bits, and so on. Radix trees grow and shrink dynamically to use as many levels as required to contain the number of mappings (e.g., a file with $N < 9$ would only need a single-level tree). The last level node contains direct mappings to physical blocks. To accelerate sequential scans, a cursor in DRAM is typically used to cache the last place searched in the tree. As each pointer is simply a 64-bit integer (a physical block number), consistency can be guaranteed through atomic instructions of the CPU. Consistency for resizing is provided by first recording modifications in an undo log.

Design considerations. Radix trees are less compact than extent trees, and typically require more indirection. Thus, tree traversal will take longer on average, since radix trees grow faster than the compact extent trees. However, radix trees have the computationally simplest mapping operation—a series of

offset calculations, which only require one memory access per level of the tree. Hence, they often perform fewer memory accesses than extent trees, as extent trees make $O(log(N))$ memory accesses per extent tree node (a binary search) to find the next node to traverse.

3.2 PM-specific Global File Mapping

Based on our analysis of file mapping performance and the challenges posed by per-file mapping approaches, we design two PM-specific mapping approaches that, unlike the per-file mapping approaches used in existing PM-optimized file systems, are *global*. This means that, rather than having a mapping structure per file, these approaches map both a file, as represented by its inode number (inum), and a logical block to a physical block. Hence, they do not require a separate inode table.

Both of the global mapping structures we design are hash tables. The intuition behind these designs is to leverage the small number of memory accesses required by the hash table structure, but to avoid the complexity of resizing per-file mapping structures as files grow and shrink. We are able to statically create these global structures, as the size of the file system—the maximum number of physical blocks—is known at file-system creation time. Hash tables are not affected by fragmentation as they do not use a compact layout. Consistency under concurrent access can frequently be resolved on a per hash bucket basis due to the flat addressing scheme.

We employ a fixed-size translation cache in DRAM that caches logical to physical block translations to accelerate lookups with locality. It is indexed using the same hash value as the full hash table for simplicity. The goal of the fixed-size translation cache is to provide a mechanism for the hash table structures that is similar to the constant-size cursor that the extent and radix trees use to accelerate sequential reads [5]. This cache contains 8 entries and is embedded in an in-DRAM inode structure maintained by Strata, our testbed file system (§4). This fixed-size translation cache is different from a page cache, which caches the mapping structure, rather than translations. The translation cache reduces the number of PM reads required for sequential read operations, benefiting from the cache locality of the in-DRAM inode structure, accessed during file system operations (§4.1).

Global Structure Challenges. Since these global mapping structures are hash tables, they exhibit lower locality due to the random nature of the hashing scheme. These global hash tables potentially exhibit even lower locality than a per-file hash table might experience, since a per-file hash table would only contain mappings relevant to that file, where the mappings in a global hash table may be randomly distributed across a much larger region of memory. However, the translation cache ameliorates this challenge and accelerates mappings with locality.

We now discuss two global file mapping hash table designs.

3.2.1 Global Cuckoo Hash Table

We show a diagram for the first global hash table structure in Fig. 3. In this hash table, each entry maps `<inum>`, `<logical block>: <physical block>(<# of blocks>)`. This hash table uses cuckoo hashing [38], which means each entry is hashed twice using two different hash functions. For lookups, at most two locations have to be consulted. For insertions, if both potential slots are full, the insertion algorithm picks one of the existing entries and evicts it by moving it to its alternate location, continuing transitively until there are no more conflicts. We use cuckoo hashing for this design instead of linear probing to avoid having to traverse potentially long chains of conflicts in pathological cases, bounding the number of memory accesses required to find a single index to 2.

The hash table is set up as a contiguous array, statically allocated at file-system creation. Consistency is ensured by first persisting the mapping information (physical block number, size) before persisting the key (inum, logical block), effectively using the key as a valid indicator. Consistency for complex inserts (i.e., inserts which cause shuffling of previous entries) is maintained by first recording operations in the file-system undo log. As complex updates are too large for atomic compare-and-swap operations, we use Intel TSX [21] in place of a per-entry lock to provide isolation. This isolation is required as inserts may occur concurrently across files—this is not a challenge for per-file mapping structures, which can rely on per-file locking mechanisms for isolation.

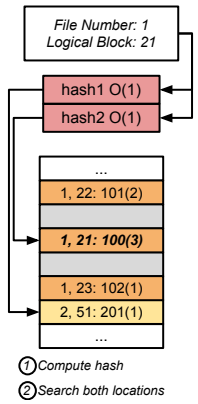

Figure 3: Global hash table with cuckoo hashing.

One issue with hash tables is that they generally present a one-to-one mapping, which is not conducive to representing ranges of contiguously allocated blocks like traditional mapping structures. To compensate, each entry in this hash table also contains a field which includes the number of file blocks that are contiguous with this entry. This mapping is maintained for every block in a series of contiguous blocks; for example, if logical blocks 21..23 are mapped contiguously to device blocks 100..102, the hash table will have entries for 100 with size 3 (shown in Fig. 3 as `100(3)`), 101 with size 2, and so on. Each entry also contains a reverse index field which describes how many blocks come before it in a contiguous range so that if a single block is removed from the end of a contiguous block range, all entries in the group can be updated accordingly. This hash table can also do parallel lookups for multiple blocks (e.g., by using SIMD instructions to compute hashes in parallel) to make lookup more efficient for ranged accesses (i.e. reading multiple blocks at a time) if fragmentation is high. Ranged nodes are crucial for the performance of large IO operations (we discuss this further in

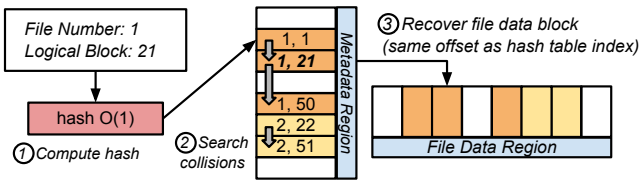

Figure 4: HashFS and an example lookup operation. Rather than storing a physical block number in the hash entry, the offset in the hash table is the physical block number.

§4.4). Ranged nodes are also cached in the translation cache to accelerate small, sequential IO operations.

Design considerations. There is a trade-off between cuckoo hashing and linear probing. Cuckoo hashing accesses at most 2 locations, but they are randomly located within the hash table, resulting in poor spatial locality. Linear probing may access more locations when there are conflicts, but they are consecutive and therefore have high spatial locality with respect to the first lookup. Therefore, a linear probing hash table with a low load factor may access only 1 or 2 cache lines on average, which would outperform a cuckoo hash table in practice. We explore this trade-off by comparing this global hash table scheme to a linear probing scheme (described in §3.2.2) to determine which scheme has better performance in practice—we measure the latency of lookups and modifications separately in our microbenchmarks (§4.1–§4.5) and end-to-end performance in our application workloads (§4.8).

3.2.2 HashFS

We also design a second global hash table, with the idea of specifically reducing the cost of the insert operation, which normally involves interacting with the block allocator. To this end, we build a single hash table that also provides block allocation, which we call *HashFS*. We present a diagram of this structure in Fig. 4.

HashFS is split into a metadata region and a file data region. Physical blocks are stored in the file data region, which starts at $fileDataStart$. A lookup resolves first to the metadata region. The corresponding physical block location is calculated from the offset of the entry in the metadata region. For example, if the hash of `<inum=1, lblk=21>` resolves to offset i in the metadata region, the location of the corresponding physical block is $(fileDataStart + i \times blockSize)$, with $blockSize = 4KB$.

Unlike the cuckoo hash table (§3.2.1), this table does not have any ranged nodes, instead providing a pure one-to-one mapping between logical blocks and physical blocks. This uses a constant space of 8 bytes per 4KB data block in the file system, for a total space overhead of $< 0.2\%$ of PM. Another advantage to this scheme is that the hash table entries are extremely simple—a combined inum and logical block, which fits into a 64-bit integer. Consistency can therefore be guaranteed simply with intrinsic atomic compare-and-swaps.

In order to efficiently implement large IO and sequential access, we perform vector hash operations using SIMD instructions. This is possible due to PM's load/store addressability and adequate bandwidth. If an IO operation does not use the maximum SIMD bandwidth, the remaining bandwidth is used to prefetch following entries, which are then cached in the translation cache. The global hash table uses linear probing, which inserts conflicting hash values into an offset slot in the same area [8]. This is in contrast to separate chaining, which allocates separate memory for a linked list to handle conflicts. An advantage of linear probing is that conflicting entries are stored in adjacent locations, reducing the overhead of searching conflicts by avoiding misses in the PM buffer [43]. Additionally, unlike cuckoo hashing, linear probing never relocates entries (i.e., rehashing), which preserves the correspondence between the index of the metadata entry and file data block. As discussed previously, we measure the trade-offs of linear probing and cuckoo hashing in our evaluation (§4.1–§4.5 and §4.8).

HashFS is not fundamentally limited to 8 byte entries. Rather than performing atomic compare-and-swap operations on an 8 byte entry, other atomic update techniques can ensure isolation (e.g., Intel TSX). The existing logging mechanisms provide crash consistency regardless of the size of the HashFS entries. The overall space overhead is low, even with larger entries (e.g., $< 0.4\%$ of the total file system capacity with 16 byte entries).

Design considerations. HashFS, like the global cuckoo hash table, also has very low spatial locality, but improves sequential access by prefetching via SIMD parallelism. HashFS's computational overhead is low on average, as HashFS only has to compute a single hash function per lookup. However, conflicts are expensive, as they are resolved by linearly following chains of entries. This means that HashFS may perform worse at high load factors (i.e. as the file system becomes more full).

How is file mapping affected by...	Design Question	§
Locality?	Optimize for specific workloads?	4.1
Fragmentation?	Make robust against file system aging?	4.2
File size?	Specialize for different file sizes?	4.3
IO size?	Optimize for sequential access?	4.4
Space utilization?	Make file mapping structure elastic?	4.5
Concurrency?	Is ensuring isolation important?	4.6
Page caching?	Is page caching necessary?	4.7
Real workloads?	Are mapping optimizations impactful?	4.8
Storage structures?	Can we reuse PM storage structures?	4.9

Table 1: The questions we answer in our evaluation.

4 Evaluation

We perform a detailed evaluation of the performance of our PM file-mapping approaches over a series of microbenchmarks and application workloads and discuss the performance characteristics of each mapping structure. We then demonstrate that the non-challenges we discussed in §2.2 are indeed

non-challenges in PM-optimized file systems. We summarize the questions we answer in our analysis in Table 1.

Experimental setup. We run our experiments on an Intel Cascade Lake-SP server with four 128GB Intel Optane DC PM modules [20]. To perform our analysis, we integrate our mapping approaches into the Strata file system [29]. We choose Strata for our analysis as it is one of the best performing open-source PM-optimized file systems, which is actively used by state-of-the-art research [39]. We configure Strata to only use PM (i.e., without SSD or HDD layers). Our file-mapping structures are integrated into both the user-space component of Strata (LibFS, which only reads mapping structures) and the kernel-space component of Strata (KernFS, which reads and updates mapping structures based on user update logs "digested" [29] from LibFS). Each experiment starts with cold caches and, unless noted, mapping structures are not page cached.

Generating fragmentation. We modify the PM block allocator in Strata to accept a *layout score* parameter at file-system initialization that controls the level of fragmentation encountered during our experiments. Layout score is a measure of file-system fragmentation which represents the ratio of file blocks that are contiguous to non-contiguous file blocks [42]. A layout score of 1.0 means all blocks are contiguous and a layout score of 0.0 means no blocks are contiguous. We allocate blocks in fragmented chunks such that the resulting files have an average layout score that is specified by the initialization parameter, which fragments both allocated file data and free space. These fragmented chunks are randomly distributed throughout the device to simulate the lack of locality experienced with real-world fragmentation. By making this modification, we can effectively simulate fragmentation without using high-overhead aging methods [25, 42]. Unless otherwise stated, we use a layout score of 0.85 for our experiments, as this was determined to be an average layout score for file systems in past studies [42].

Experimental results. Unless otherwise stated, we report the average latency of the file mapping operation over 10 repeated measurements, including all overhead associated with the mapping structure, such as hash computation and undo logging. Error bars report 95% confidence intervals. For insertion/truncation operations, the latency of the file mapping operation includes the overhead of the block allocator. Note that HashFS uses its own block allocation mechanisms (§3.2.2). All other evaluated file mapping structures use the block allocator already present in Strata.

4.1 Locality

We analyze how particular access patterns impact the performance of file mapping. We perform an experiment using a 1GB file where we perform single-block reads and appends in either a "cold cache" scenario (i.e., only perform one operation and then reset the experiment) or a "hot cache" scenario (i.e., perform 100,000 operations). We perform reads and appends as they exercise file mapping structure reads and file mapping structure writes, respectively. We measure the latency of the file mapping operation and report the average (in CPU cycles) in Fig. 5.

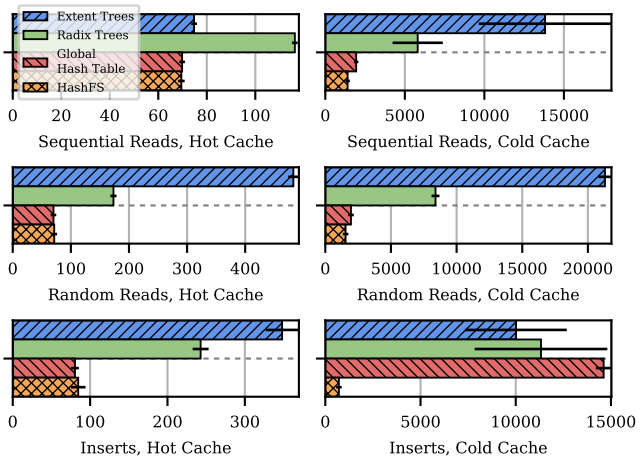

Figure 5: Locality test, measuring the latency of the file mapping operation in cycles (using `rdtscp`).

On average, hot cache operations result in low latency for all file mapping approaches, especially for sequential reads, for which the per-file cursor optimizations and global SIMD prefetching were designed. The greatest difference is in the cold cache case, where global file mapping is up to 15× faster than per-file mapping. This is due to the cost of tree operations when operating on a cold cache. Tree operations must traverse multiple levels of indirection, making multiple memory accesses per level of indirection; the global file-mapping structures, on the other hand, make fewer than two accesses to PM on average. The exception to the performance benefits of global file-mapping structures is the performance of cold global cuckoo hash table inserts, which perform on par with radix trees (within confidence intervals) and extent trees. The main factor causing the high variability in cold cache insert performance for these structures is the overhead of the block allocator, which also has persistent and volatile metadata structures which incur last level cache misses (e.g., a persistent bitmap, a volatile free-list maintained as a red-black tree, and a mutex for providing exclusion between kernel threads [29]). In contrast, HashFS, which does not use Strata's block allocator, is between 14–20× faster than all other file-mapping structures in this case. Overall, this initial experiment shows us that a global hash table structure can dramatically outperform per-file tree structures, particularly for access patterns without locality.

4.2 Fragmentation

We now measure the effect that file-system fragmentation has on file mapping. We perform this experiment on a single 1GB

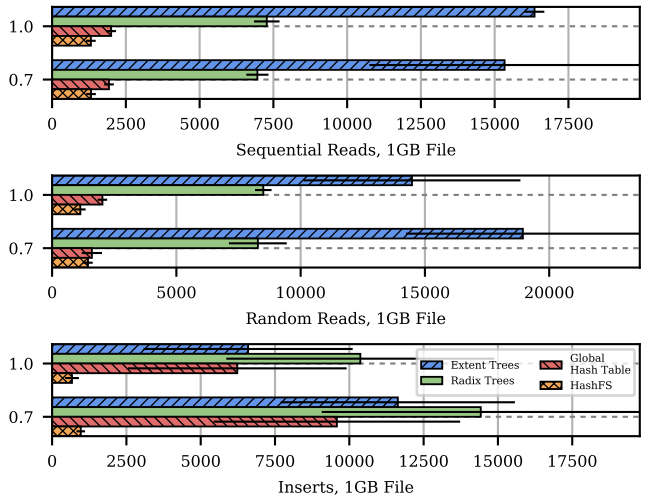

Figure 6: Fragmentation test, measuring the impact of high (0.7) and low (1.0) fragmentation on file mapping latency.

file, and vary the fragmentation of the file from no fragmentation (layout score 1.0) to heavily fragmented (layout score 0.7 represents a heavily-fragmented file system according to prior work [42]). We measure the average "cold cache" file mapping latency (i.e., one operation before reset, as described in the previous experiment) over 10 trials of each operation and present the results of our experiment in Fig. 6.

Between the different levels of fragmentation, the only major difference is the performance of the extent trees. Intuitively, this difference arises from the way that extent trees represent extents—if the file is not fragmented, the extent tree can be much smaller and therefore much faster to traverse. The other evaluated mapping approaches are unaffected by fragmentation for reads. For insertions, the block allocator takes longer on average to find free blocks, giving HashFS an advantage. We conclude that per-file mapping performs poorly on fragmented file systems, while HashFS is unaffected by fragmentation.

4.3 File Size

The per-file tree structures have a depth that is dependent on the size of the file, whereas the global hash table structures are flat. We therefore measure the latency of file mapping across three file sizes: 4KB, 4MB, and 4GB, representing small, medium, and large files. We report the average latency (over 10 trials) for each file mapping operation in Fig. 7.

As expected, the per-file structures grow as the size of the file increases, which results in more indirect traversal operations per file mapping, resulting in longer latency. The range of this increase is naturally smaller for sequential reads (less than 10% across extent trees and radix trees for sequential reads, but 34% for random reads) due to the inherent locality of the data structure). The mapping overhead for the hash tables, however, is static, showing that the hash table structures do not suffer performance degradation across file sizes.

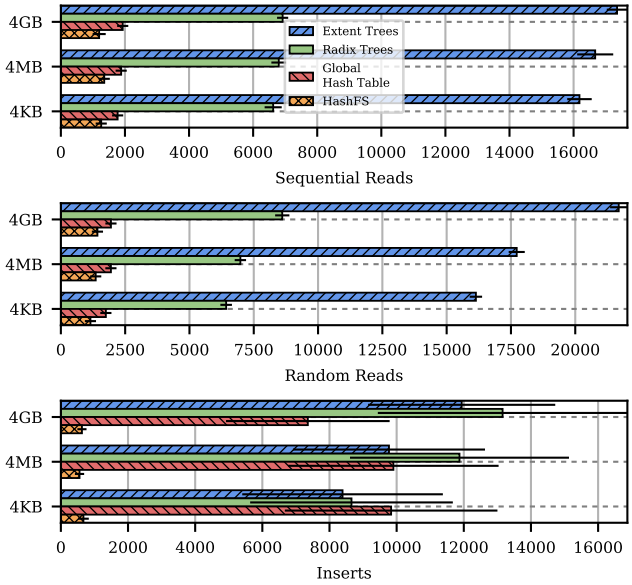

Figure 7: Impact of file size on file mapping latency.

4.4 IO Size

Our previous experiments perform single block operations (i.e., reading or writing to a single 4KB block). However, many applications batch their IO to include multiple blocks. We therefore measure the impact on file mapping as an application operates on 4KB (a single block), 64KB (16 blocks), and 1MB (256 blocks). We then measure the proportion of the IO path that comprises file mapping and report the average (over 10 trials) in Fig. 8. Since IO latency increases with IO size regardless of the mapping structure, we present the results of this experiment as a ratio of IO latency to normalize this increase in latency and to specifically show the impact of the file mapping operation on the overall latency.

As the number of blocks read in a group increases, the gap between the tree structures and the hash table structures closes. Radix and extent trees locate ranges of blocks together in the same leaf node and thus accelerate larger IO operations. At the same time, as the number of blocks increases, the proportion of the file system time spent on file mapping drops dramatically. For example, for an IO size of 1MB, radix tree mapping takes only 3% of the IO path for reads. At this IO size, radix trees perform best for both sequential and random reads. However, the impact of this performance increase is not seen at the application level (§4.8). We also note that without the ranged node optimization introduced for the cuckoo hash table (§3.2.1), the mapping latency ratio would remain constant for the global cuckoo hash table, rather than decreasing as the IO size increases.

We also see the advantage HashFS has over the global cuckoo hash table. Not only does HashFS have lower file mapping latency for all operations compared to cuckoo hashing, it also has the lowest insertion time, since it is able to bypass the traditional block allocation overhead.

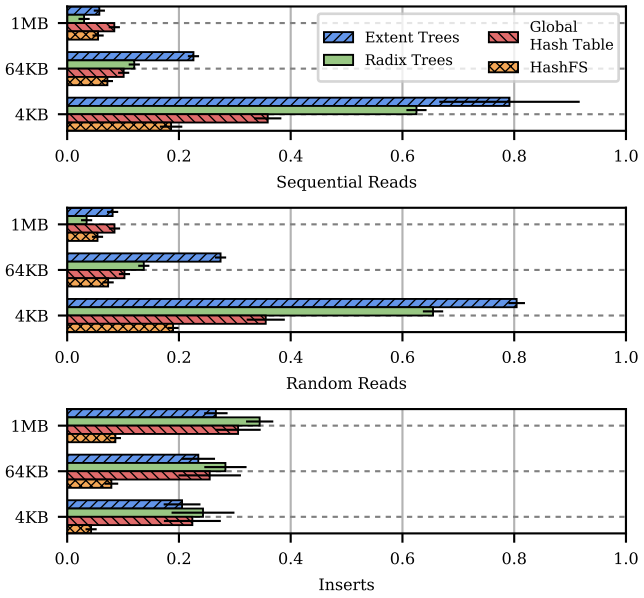

Figure 8: IO size test. We report the average proportion of the IO path spent performing file mapping.

4.5 Space Utilization

A potential disadvantage of using a global hash table structure is that collisions grow with the percentage of the structure used, meaning that as file system utilization grows, file mapping latency may increase as the number of collisions increase. We measure this effect by creating a file system with a total capacity of 128GB. We then measure the proportion of the IO path that comprises file mapping when the overall space utilization ranges from moderately full (80%) to extremely full (95%). Our prior experiments demonstrate the performance on a mostly empty file system. We perform this microbenchmark using "cold cache" (i.e., one operation per file), single-block file mapping operations as per previous experiments. We report the average file mapping latency ratio (over 10 trials) in Fig. 9. Since IO latency increases with overall file system utilization, we present the results of this experiment as a fraction of IO latency to account for the increase in overall latency.

As expected, the latency of the mapping operation for per-file tree structures is unchanged when comparing low utilization to high utilization. The main difference is the latency of the two global hash table structures. At low utilization, as seen in previous experiments (§4.4), HashFS outperforms global cuckoo hashing. At 80% utilization, the performance of random and sequential reads are very similar for HashFS and the global cuckoo hash table (within ±3%, or within error). HashFS still outperforms the global cuckoo hashing table for insertions. At 95% utilization, however, HashFS incurs between 12–14% more latency for reads than the global cuckoo hash table, and has equivalent performance for insertions.

We conclude that while higher utilization causes HashFS to degrade in performance relative to global cuckoo hashing, both file mapping structures still outperform the per-file mapping structures even at high utilization.

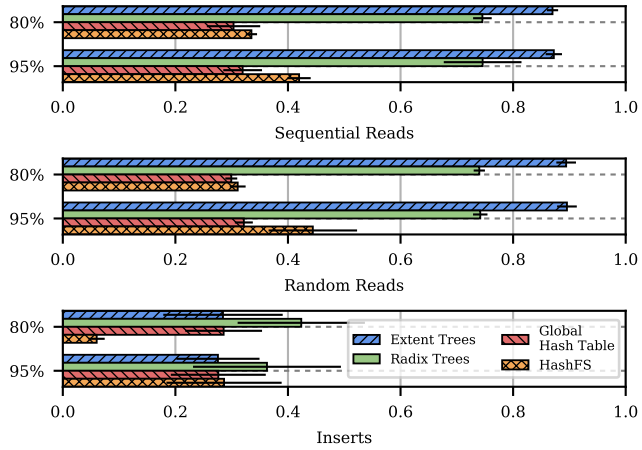

Figure 9: Space utilization test, measuring the proportion of the IO path spent in file mapping versus the overall utilization of the file system (in percent).

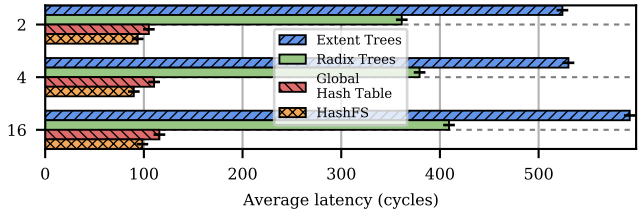

Figure 10: Average file mapping latency, varying # of threads.

4.6 Concurrency

We now conduct an experiment to quantify how well our file-mapping approaches perform under concurrent access. To simulate a high-contention scenario, we conduct an experiment with multiple threads reading and writing the same file (every thread reads and then appends to the same file). This causes high read-write contention between Strata's KernFS component which asynchronously updates the file-mapping structures and the user-space LibFS which reads the file-mapping structures, causing contention for both the per-file and global file-mapping structures. Each file-mapping structure manages its own synchronization—Strata manages the synchronization of other file metadata via a per-file reader-writer lock.

We show the result of the experiment in Fig. 10. We see that, as the number of threads increases, the latency of file mapping increases slightly (up to 13% for extent trees for an 8× increase in concurrency) across all file mapping approaches. Furthermore, the ranking among approaches does not change across any number of threads. This shows that common concurrent file access patterns allow per-file mapping approaches to use coarse-grained consistency mechanisms without impact on scalability, while the transactional memory and hash bucket layout optimizations for our two global file mapping

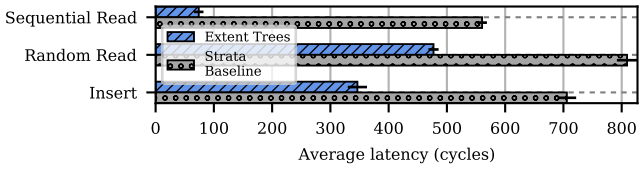

Figure 11: Page-cache experiment, measuring the average file-mapping latency of PM-optimized extent trees and page-cached extent trees (Strata Baseline).

Workload	Average file size	# of files	IO size (R/W)	R/W ratio
fileserver	128KB	1,000	1MB/16KB	1:2
webproxy	16KB	1,000	16KB/16KB	5:1

Table 2: Filebench workload configurations.

approaches can effectively hide synchronization overheads.

In summary, we find that the isolation mechanisms used in our PM-optimized file-mapping structures are not bottlenecks. All structures are scalable for common file access patterns.

4.7 Page Caching

We now discuss why traditional page-caching should not be employed for PM file-mapping structures. To demonstrate why, we provide a microbenchmark that compares Strata's default mapping structure (page-cached extent trees) to our implementation of extent trees (which is based on Strata's implementation, but bypasses the page cache and operates directly on PM). In this experiment, we open a 1GB file and perform 1,000,000 operations on it (single block reads or inserts) and report the average file mapping latency in cycles.

We show the results in Fig. 11. We can see that, even after many iterations, the dynamic allocation overhead of the page cache is not amortized in the read case. The insertion case is also slower due to the page cache overhead—the PM-optimized extent trees write updates directly back to PM.

Based on the results of this experiment, it is clearly more beneficial to perform file mapping directly on PM, as it reduces the number of bytes read and written to the device, reduces DRAM overhead, and decreases the overall overhead of the IO path. Therefore, we advocate for the use of lower overhead caching methods, such as cursors, rather than relying on the page cache.

4.8 Application Workloads

We provide two application benchmarks to measure the overall benefit our PM-optimized file-mapping structures have on application throughput. We compare our file-mapping structures to the file-mapping structure present in Strata, which is a per-file, page-cached extent tree. We use this file-mapping structure as our baseline, as Strata is a state-of-the-art PM-optimized file system [29, 39].

Filebench. We test our file mapping structures using

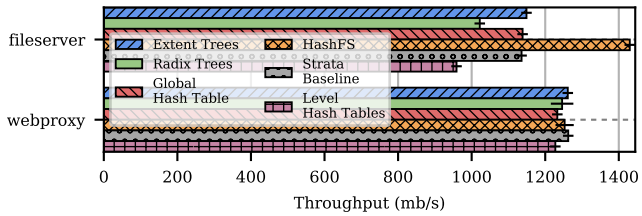

Figure 12: Filebench results for one write-heavy workload (fileserver) and one read-heavy workload (webproxy).

Workload	Characterization	Example
A	50% reads, 50% updates	Session activity store
B	95% reads, 5% updates	Photo tagging
C	100% reads	User profile cache
D	95% reads, 5% inserts	User status updates (read latest)
E	95% scans, 5% inserts	Threaded conversations
F	50% reads, 50% read-modify-write	User database

Table 3: Description of YCSB workload configurations.

filebench [45], a popular file-system-testing framework which has been used to evaluate many PM-optimized file systems [29, 52, 56]. We select the fileserver (write-heavy) and webproxy (read-heavy) workloads. These workloads test file mapping structure reads, updates, and deletions, and emulate the performance of mapping structures as they age under repeated modifications. We describe the characteristics of these workloads in Table 2.

We show the results of the filebench experiments in Fig. 12. In the fileserver workload, HashFS outperforms the baseline by 26%, while the other mapping structures perform similarly to the baseline. This result is explained by the IO size microbenchmark (§4.4). In this microbenchmark, HashFS performs the best for insertions, which is the predominant operation in this workload. In this workload, the radix trees perform the worst, experiencing a 10% throughput drop versus the baseline. Insert performance of radix trees is the worst among our file mapping approaches (Fig. 8). Extent trees and the global cuckoo hash table both perform similarly for large IO reads and smaller block insertions, so they perform similarly here and are not an improvement over the baseline.

The webproxy workload does not show any major difference in throughput across the file mapping approaches (all within ±2%, or within error). This is because this is a read-heavy workload on relatively small files with hot caches, and we show in §4.1 that the performance across file mapping structures is very similar in this case.

YCSB. We also evaluate the end-to-end performance on key-value workloads using YCSB [10] on LevelDB [17], a common benchmark to measure the performance of PM-optimized file systems [29]. We measure the throughput for all standard YCSB workloads (A–F). YCSB uses a Zipfian distribution to select keys for operations. We report the characteristics of these workloads in Table 3. We configure our YCSB tests to

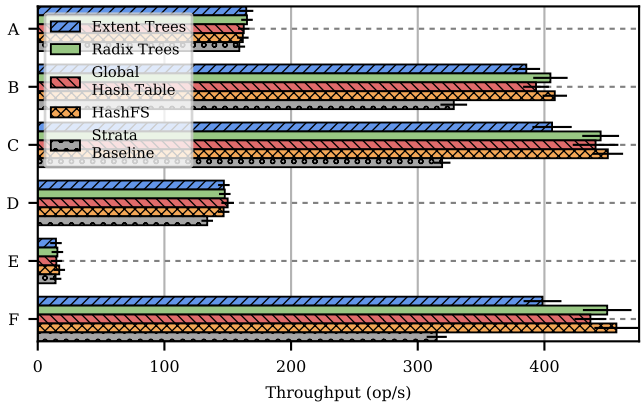

Figure 13: YCSB running on LevelDB. We report the average overall throughput (operations/second) on all workloads (A–F).

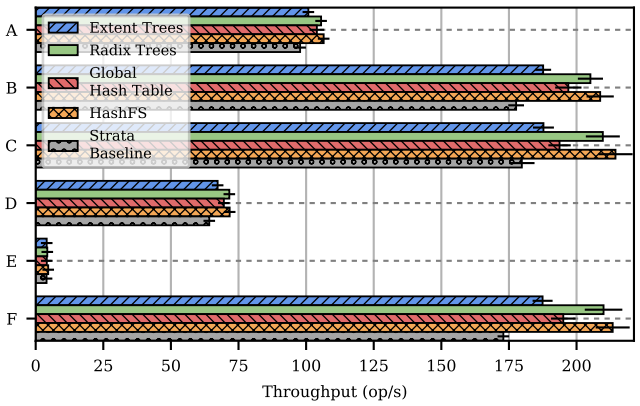

Figure 14: YCSB experiment using a single thread.

use a key-value database containing 1.6 million 1 KB key-value pairs (a 1.5 GB database), using a maximum file size of 128 MB for the LevelDB backend running on 4 threads. Our experiments perform 100,000 operations on the database, as dictated by the YCSB workload.

We show the results of our YCSB tests in Fig. 13. Workloads A and E are primarily bounded by the in-memory operations of LevelDB (e.g. performing read and scan operations), and not bounded by the file system, thus we see similar throughput across all mapping approaches (within 3% of the baseline). However, for the other workloads (B, C, D, F), HashFS provides the best performance, providing between 10–45% increase in throughput on workload F versus the other file mapping approaches. Radix trees and the global cuckoo hash tables are both slower by 2–4% on average than HashFS, but the PM-optimized extent trees perform worse than HashFS by 13% in workloads C and F. This is due to the generally poor performance of the extent tree structure for random reads, which dominate these workloads. In these workloads, the default file mapping structure in Strata spends 70% of the file IO path in file mapping—this provides ample opportunity for improvement, which is why HashFS is able to increase the overall throughput by up to 45% in these cases. We further discuss the performance of the baseline in our discussion on concurrency (§4.6).

We also show how concurrency impacts file mapping in real workloads by rerunning our YCSB benchmark using a single thread (Fig. 14).This experiment differs from the multi-threaded version (Fig. 13) other than the overall magnitude of the throughput. Additionally, the increase in throughput over the baseline is much higher (45%) in the multi-threaded experiment than the single threaded experiment (23%). Upon further investigation, we find that this is because the Strata *page cache* is not scalable. Strata's page cache maintains a global list of pages, with a single shared lock for consistency.

Summary. We draw two conclusions from these experiments: (1) HashFS results in the best overall throughput among our PM-optimized mapping structures; and (2) HashFS always matches or exceeds the performance of Strata's default mapping structure (page-cached extent trees). We therefore conclude that HashFS file mapping provides the best overall performance in real application workloads.

4.9 File Mapping via Level Hashing?

We examine whether PM storage structures can be efficiently used for file mapping, thus enabling the reuse of prior work. We select level hashing [7, 59], a state-of-the-art hash-table storage structure for PM, as a case study. Level hashing outperforms RECIPE-converted structures [32] and, as a hash table, is a good contender against our best-performing file mapping structures, which are also hash tables. The goal of this case study is to see if general-purpose PM data structures proposed in prior work can be used as file mapping structures, as-is. Hence, we do not apply any file-system specific optimizations to level hashing.

We evaluate how level hashing performs relative to our PM file mapping structure in our filebench workloads, shown in Fig. 12. In all cases, HashFS outperforms level hashing. In particular, for the fileserver workload (the most write heavy workload), level hashing underperforms even Strata's baseline. In particular, even though level hashing provides an efficient resizing operation, neither of our global hash table structures require resizing, as their total size is known at file system creation time. This experiment shows the importance of file-system specific optimizations for PM file-mapping structures and this suggests that PM storage structures should not be directly used for file mapping.

5 Discussion

Generalizability of results. In our microbenchmarks, we report the performance of file mapping operations in isolation from the rest of the file system; these results are applicable to other PM file systems that use persistent mapping structures (e.g., ext4-DAX, SplitFS [24], NVFS [40], and ZoFS [13]).

Strata batches file mapping updates via the application log

in LibFS (SplitFS has a similar batching system), which is measured in our macrobenchmark results. Batching amortizes the update overhead of mapping structures. For this reason, we predict that HashFS would outperform other mapping structures by a larger margin on PM file systems that do not batch updates (e.g., ext4-DAX, NFVS, and ZoFS).

Resilience. Resilience to crashes and data corruption is not a challenge exclusive to file mapping structures and reliability concerns are usually handled at a file-system level, rather than specifically for file-mapping structures. As we use a log for non-idempotent file-mapping structure operations (§3.1.1) and Strata logs other file mapping operations, all of our file mapping structures are equivalently crash-consistent. For resilience to data corruption and other device failures, our mapping structures can use existing approaches (e.g., TickTock replication from NOVA-Fortis [53]).

6 Related Work

There is little prior work that specifically analyzes the performance of file mapping structures. BetrFS [23] finds that write-optimized, global directory and file mapping structures are effective at optimizing write-heavy workloads. However, this analysis is performed on SSDs.

File mapping in PM file systems. PMFS [15] uses B-trees, allocating file data blocks in sizes of 4KB, 2MB, and 1GB memory pages. The PMFS allocator is therefore similar to an OS virtual memory allocator, albeit with different consistency and durability requirements. PMFS contrasts itself with systems that use extents for file mapping, but provides no justification for its scheme other than the fact that it transparently supports large pages [21]. We therefore do not know if its file mapping scheme is adequate for PM file systems. This problem extends to DevFS, which re-uses the metadata structures present in PMFS [27]. Strata and ext4-DAX both use extent trees for file mapping, with Strata using extent trees at all levels of its storage device hierarchy [11, 29]. Both of these systems use extent trees based on the legacy of ext4, providing no analysis if extent trees are optimal for PM.

PM-optimized storage structures. Much work has proposed PM optimized storage structures, both generic [6, 12, 31, 32, 37, 46, 47, 54, 58, 59] and within the context of database applications, such as key-value stores [26, 28, 51]. These provide in-place atomic updates whenever possible to avoid having to keep a separate log. However, common file system operations typically require atomic update of *multiple* file-system structures—e.g., when allocating blocks, the block bitmap must also be modified. Enforcing consistency and atomicity for a single data structure alone is therefore insufficient—we need to analyze file mapping structures within PM file systems to achieve efficient metadata consistency and durability.

Memory mapping. Mapping virtual to physical memory locations is similar to file mapping. A large body of research has improved virtual memory for decades [3, 44, 55] and has devised similar structures; page tables are radix trees on many platforms and recent work proposes cuckoo hashing as a more scalable alternative [41]. The key differences are in caching and consistency. File mapping caches are optimized for sequential access via cursors and SIMD prefetching; they are shared across all threads, simplifying frequent concurrent updates. MMUs optimize for random read access via translation lookaside buffers (TLBs) that are not shared across CPU cores, requiring expensive TLB shootdowns for concurrent updates. Additionally, since file-mapping structures are maintained in software rather than hardware, they allow for a wider variety of designs which may be difficult to efficiently implement in hardware (i.e., extent trees or HashFS's linear probing).

7 Conclusion

File mapping is now a significant part of the IO path overhead on PM file systems that can no longer be mitigated by a page cache. We designed four different PM-optimized mapping structures to explore the different challenges associated with file mapping on PM. Our analysis of these mapping structures shows that our PM-optimized hash table structure, HashFS, performs the best on average, providing up to 45% improvement on real application workloads.

Acknowledgements

We thank the anonymous reviewers and our shepherd, Rob Johnson, for their valuable feedback. This work is supported by Applications Driving Architectures (ADA) Research Center (a JUMP Center co-sponsored by SRC and DARPA), the National Science Foundation under grants CNS-1900457 and DGE-1256260, the Texas Systems Research Consortium, the Institute for Information and Communications Technology Planning and Evaluation (IITP) under a grant funded by the Korea government (MSIT) (No. 2019-0-00118) and Samsung Electronics. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] H. Akinaga and H. Shima. Resistive Random Access Memory (ReRAM) Based on Metal Oxides. *Proceedings of the IEEE*, 98(12):2237–2251, Dec 2010.

[2] Dmytro Apalkov, Alexey Khvalkovskiy, Steven Watts, Vladimir Nikitin, Xueti Tang, Daniel Lottis, Kiseok Moon, Xiao Luo, Eugene Chen, Adrian Ong, Alexander Driskill-Smith, and Mohamad Krounbi. Spin-transfer Torque Magnetic Random Access Memory

(STT-MRAM). *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 9(2):13:1–13:35, May 2013.

[3] Thomas W Barr, Alan L Cox, and Scott Rixner. Translation caching: skip, don't walk (the page table). *ACM SIGARCH Computer Architecture News*, 38(3):48–59, 2010.

[4] Jalil Boukhobza, Stéphane Rubini, Renhai Chen, and Zili Shao. Emerging NVM: A survey on architectural integration and research challenges. *ACM Trans. Design Autom. Electr. Syst.*, 23(2):14:1–14:32, 2018.

[5] Mingming Cao, Suparna Bhattacharya, and Ted Ts'o. Ext4: The next generation of ext2/3 filesystem. In *LSF*, 2007.

[6] Shimin Chen and Qin Jin. Persistent B+-trees in Non-volatile Main Memory. *Proc. VLDB Endow.*, 8(7):786–797, February 2015.

[7] Zhangyu Chen, Yu Huang, Bo Ding, and Pengfei Zuo. Lock-free concurrent level hashing for persistent memory. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 799–812. USENIX Association, July 2020.

[8] JG Clerry. Compact hash tables using bidirectional linear probing. *IEEE Transactions on Computers*, 100(9):828–834, 1984.

[9] Alex Conway, Ainesh Bakshi, Yizheng Jiao, William Jannen, Yang Zhan, Jun Yuan, Michael A. Bender, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, and Martin Farach-Colton. File Systems Fated for Senescence? Nonsense, Says Science! In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 45–58, Santa Clara, CA, 2017. USENIX Association.

[10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[11] Jonathan Corbet. Supporting filesystems in persistent memory, September 2014.

[12] Biplob Debnath, Alireza Haghdoost, Asim Kadav, Mohammed G. Khatib, and Cristian Ungureanu. Revisiting hash table design for phase change memory. *Operating Systems Review*, 49(2):18–26, 2015.

[13] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and protection in the zofs user-space nvm file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 478–493. ACM, 2019.

[14] Mingkai Dong, Qianqian Yu, Xiaozhou Zhou, Yang Hong, Haibo Chen, and Binyu Zang. Rethinking benchmarking for nvm-based file systems. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, page 20. ACM, 2016.

[15] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM.

[16] Jose M. Faleiro and Daniel J. Abadi. Latch-free synchronization in database systems: Silver bullet or fool's gold? In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, page 9, 2017.

[17] Sanjay Ghemawat and Jeff Dean. Leveldb. http://leveldb.org, 2011.

[18] Vaibhav Gogte, William Wang, Stephan Diestelhorst, Aasheesh Kolli, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. Software wear management for persistent memories. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 45–63, Boston, MA, 2019. USENIX Association.

[19] Sangwook Shane Hahn, Sungjin Lee, Cheng Ji, Li-Pin Chang, Inhyuk Yee, Liang Shi, Chun Jason Xue, and Jihong Kim. Improving file system performance of mobile storage systems using a decoupled defragmenter. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 759–771, 2017.

[20] Intel. Intel® Optane™ DC Persistent Memory. http://www.intel.com/optanedcpersistentmemory, 2019.

[21] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, 2019.

[22] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module, 2019.

[23] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Betrfs: A right-optimized write-optimized file system. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 301–315, Santa Clara, CA, 2015. USENIX Association.

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

[25] Saurabh Kadekodi, Vaishnavh Nagarajan, and Gregory 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 2018, Boston, MA, USA, July 11-13, 2018.*, pages 691–704, 2018.

[26] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young-ri Choi. SLM-DB: single-level key-value store with persistent memory. In Merchant and Weatherspoon [34], pages 191–205.

[27] Sudarsun Kannan, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Yuangang Wang, Jun Xu, and Gopinath Palani. Designing a true direct-access file system with devfs. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 241–256, 2018.

[28] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning lsms for nonvolatile memory with novelsm. In *2018 USENIX Annual Technical Conference (USENIX-ATC 18)*, pages 993–1005, 2018.

[29] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 460–477, New York, NY, USA, 2017. ACM.

[30] E. Lee, H. Bahn, S. Yoo, and S. H. Noh. Empirical study of nvm storage: An operating system's perspective and implications. In *2014 IEEE 22nd International Symposium on Modelling, Analysis Simulation of Computer and Telecommunication Systems*, pages 405–410, Sep. 2014.

[31] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 257–270, 2017.

[32] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. RECIPE: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*, Ontario, Canada, October 2019.

[33] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, volume 2, pages 21–33, 2007.

[34] Arif Merchant and Hakim Weatherspoon, editors. *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019*. USENIX Association, 2019.

[35] Micron. Battery-backed nvdimms, 2017.

[36] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H Noh, and Beomseok Nam. Write-optimized dynamic hashing for persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 31–44, 2019.

[37] Moohyeon Nam, Hokeun Cha, Youngri Choi, Sam H Noh, and Beomseok Nam. Write-Optimized and dynamic-hashing for Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 2019.

[38] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[39] Waleed Reda, Henry N. Schuh, Jongyul Kim, Youngjin Kwon, Marco Canini, Dejan Kostić, Simon Peter, Emmett Witchel, and Thomas Anderson. Assise: Performance and availability via NVM colocation in a distributed file system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, Banff, Alberta, November 2020. USENIX Association.

[40] RedHat. NFVS documentation. https://people.redhat.com/~mpatocka/nvfs/INTERNALS, 2020.

[41] Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. Elastic cuckoo page tables: Rethinking virtual memory translation for parallelism. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1093–1108, 2020.

[42] Keith A Smith and Margo I Seltzer. File system aging—increasing the relevance of file system benchmarks. In *ACM SIGMETRICS Performance Evaluation Review*, volume 25, pages 203–213. ACM, 1997.

[43] Steven Swanson. Early measurements of intel's 3dx-point persistent memory dimms, Apr 2019.

[44] M. Talluri, M. D. Hill, and Y. A. Khalidi. A new page table for 64-bit address spaces. In *Proceedings of the*

Fifteenth ACM Symposium on Operating Systems Principles, SOSP '95, page 184–200, New York, NY, USA, 1995. Association for Computing Machinery.

[45] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX; login*, 41, 2016.

[46] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, pages 5–5. USENIX Association, February 2011.

[47] Chundong Wang, Qingsong Wei, Lingkun Wu, Sibo Wang, Cheng Chen, Xiaokui Xiao, Jun Yang, Mingdi Xue, and Yechao Yang. Persisting rb-tree into NVM in a consistency perspective. *TOS*, 14(1):6:1–6:27, 2018.

[48] Ying Wang, Dejun Jiang, and Jin Xiong. Caching or not: Rethinking virtual file system for non-volatile main memory. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.

[49] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. Building a bw-tree takes more than just buzz words. In *Proceedings of the 2018 ACM International Conference on Management of Data*, SIGMOD '18, pages 473–488, 2018.

[50] H-S Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E Goodson. Phase Change Memory. *Proceedings of the IEEE*, 98(12):2201–2227, Dec 2010.

[51] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. HiKV: a hybrid index key-value store for DRAM-NVM memory systems. In *2017 USENIX Annual Technical Conference (USENIXATC 17)*, pages 349–362, 2017.

[52] Jian Xu and Steven 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*, FAST'16, pages 323–338, Berkeley, CA, USA, 2016. USENIX Association.

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

[54] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 167–181, 2015.

[55] Idan Yaniv and Dan Tsafrir. Hash, don't cache (the page table). In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, SIGMETRICS '16, page 337–350, New York, NY, USA, 2016. Association for Computing Machinery.

[56] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. Ziggurat: A tiered file system for non-volatile main memories and disks. In Merchant and Weatherspoon [34], pages 207–219.

[57] Shengan Zheng, Hao Liu, Linpeng Huang, Yanyan Shen, and Yanmin Zhu. HMVFS: A versioning file system on DRAM/NVM hybrid memory. *J. Parallel Distrib. Comput.*, 120:355–368, 2018.

[58] Jie Zhou, Yanyan Shen, Sumin Li, and Linpeng Huang. NVHT: An efficient key-value storage library for non-volatile memory. In *Proceedings of the 3rd IEEE/ACM International Conference on Big Data Computing, Applications and Technologies*, pages 227–236. ACM, 2016.

[59] Pengfei Zuo, Yu Hua, and Jie Wu. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 461–476, 2018.