AUGEFS: A Scalable Userspace Log-Structured File System for Modern SSDs

Wenqing Jia, Dejun Jiang, Jin Xiong
State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences;
Research Center for Advanced Computer Systems, Institute of Computing Technology, Chinese Academy of Sciences;
University of Chinese Academy of Sciences
{jiawenqing19z, jiangdejun, xiongjin}@ict.ac.cn

***Abstract*—We present AUGEFS, a scalable userspace log-structured file system for modern SSDs. AUGEFS re-architects the file system stack to address three critical challenges: inefficient control plane, limited metadata scalability, and underutilized device bandwidth. First, we propose a shared and protected address space within the userspace of accessing applications to run AUGEFS, which enables high-performance data plane and efficient control plane. Second, we design a scalable LSM-tree based key-value store called METADB to organize small-sized metadata in AUGEFS. To improve metadata scalability, METADB employs parallel request processing to reduce thread synchronization overhead and fine-grained parallel write-ahead log to eliminate false sharing in metadata persistence. Finally, AUGEFS distributes files into different domains. To reduce contention, we maintain space management metadata for each domain independently, which helps scale data performance and improve device utilization. Moreover, AUGEFS designs an asynchronous IO stack for *fsync* to reduce the latency of synchronous writes. The evaluation results show that AUGEFS significantly improves both metadata scalability and data scalability.**

***Index Terms*—SSD, Userspace File System, Scalability**

I. INTRODUCTION

Recent developments in non-volatile memory technologies bring modern solid-state drives (SSDs) with both ultra-low latency and high throughput. For example, Samsung Z-SSD [84], Intel Optane SSD [28], [30], and Toshiba XL-Flash [72] provide sub-ten microseconds of IO latency and up to 7.0 GB/s [30] of IO bandwidth. However, similar to the conventional SSDs [29], these advanced SSDs still exhibit higher performance under sequential accesses than that under random accesses [28], [72], [84], [92].

To fully leverage the features of modern SSDs, this paper explores the issue: how to build a scalable and high-performance userspace log-structured file system (LFS) based on modern SSDs. First, traditional kernel file systems [4], [10] entail crossing the user-kernel boundary for IO operations and traversing multiple layers of the kernel IO stack, resulting in substantial software overhead [97], [98]. In contrast, userspace file systems [7], [42], [59], [76] allow applications to directly access storage devices and reduce kernel involvement considerably. Second, the LFS organizes the address space exposed by storage devices as an append-only log, which is a promising practice to utilize the sequential access pattern favored by modern SSDs [39], [92]. Thus, exploring the design choices in a userspace LFS yields significance, which can benefit systems like databases, key-value stores, mail server, etc.

Intuitively, one can port mature kernel-based LFSs, such as F2FS [10], to the current userspace file system (FS) architectures [7], [42], [59], [76]. Unfortunately, we identify three inefficiencies existing in current userspace FS architectures and LFS designs when meeting modern SSDs. First, most userspace file systems [7], [42], [76], [87] bypass the kernel IO stack for high-performance data operations. However, they rely on a trusted process or the kernel for the control plane, including metadata operations and concurrency control. Thus, existing userspace FSs suffer from expensive inter-process communication (IPC) or costly kernel trapping, which imposes overhead for the control plane and limits the scalability of FS operations, especially for multi-process sharing [17].

Second, small-sized metadata in LFSs [10] mismatches with the block interface of SSDs, which brings IO amplification [40], [42] and causes suboptimal metadata performance. Although some works aim to improve metadata efficiency [60], [63], they still suffer from severe metadata write amplification due to the block-based metadata organization. A promising practice to handle small-sized metadata for local file systems is to use log-structured merge tree (LSM-tree) based key-value stores (LSM-KVS) as several FSs do [2], [34], [40], [85], [88]. However, we observe poor scalability in both metadata updates and metadata persistence when adopting LSM-KVS. Through profiling and analyzing, we find the root cause comes from a fundamental design in existing LSM-KVS: the group updating mechanism that aims to improve write performance for slow storage devices. On the one hand, group updating causes the false sharing problem in metadata updates. Metadata updates suffer from extra synchronization overhead among multiple threads even when they update their own private metadata, as in §II-C. On the other hand, group updating causes false sharing in metadata persistence. The group updating generates a shared write-ahead log (WAL) to batch multiple writes from small-sized metadata to generate one block-based log write. When a thread invokes *fsync* for only persisting a part of buffered metadata, it undergoes transaction entanglement [40] that requires flushing the entire shared WAL, which brings heavy write amplification and high latency.

Third, existing LFSs, such as F2FS [10], fail to fully exploit the device bandwidth of modern SSDs, especially

for synchronous writes that are commonly used in many applications [12], [16], [24], [64], [68]. Although Max [52] identifies that the locking of shared data structure limits data scalability and splits most of them, we find there still exist two potential issues causing suboptimal data scalability and underutilization of device bandwidth: the updating dependencies for the *fsync* operation and the locking contention overhead at the submitting stage of both data and node blocks (file index).

In this paper, we propose AUGEFS, a scAlable Userspace loG-structurEd File System for modern SSDs. To address the aforementioned inefficiencies, AUGEFS re-architects the file system stack with the following techniques. First, AUGEFS introduces a shared and protected address space architecture to run the file system, which aims to provide an efficient control plane. AUGEFS is built in a reserved range within the user address space of the accessing applications, which achieves library-level file system performance. Meanwhile, the address space of AUGEFS is shared by multiple processes to support cross-process sharing efficiently. To protect AUGEFS from stray writes, we adopt memory protection technology of existing processors (e.g., Intel Memory Protection Key, MPK) [15] to allow a process to access the address space of AUGEFS only when it executes the file system services.

Second, we propose METADB, an LSM-tree based key-value store, to manage small-sized metadata (e.g., inode). Specially, METADB designs two techniques to address the limitation of group updating. To scale up the processing of metadata updates, METADB introduces parallel request processing to allow threads to update metadata independently without sacrificing correctness, which greatly reduces the thread synchronization overhead. To overcome the limitation of false sharing on WAL, METADB designs fine-grained parallel WAL by exploiting persistent memory region (PMR) [70] in modern SSDs. Its WAL allows different threads to write and persist their log entries independently, improving the performance and scalability of metadata persistence significantly.

Finally, AUGEFS introduces domain-based file organization to increase the device bandwidth utilization. After delegating inode and directory operations to METADB, AUGEFS further groups files into different domains and maintains space management metadata for each domain individually. Thus, different domains can execute file *read*/*write*, garbage collection, and checkpoint in parallel with little locking contention. In addition, AUGEFS proposes an asynchronous IO stack for *fsync*, which parallels the writing of data and node blocks to eliminate their updating dependencies. This stack improves device utilization without compromising consistency.

We implement AUGEFS and evaluate it with file system benchmarks and real-world applications against six file systems: F2FS [10], Max [52], uFS [59], Strata [42], Ext4 [4], and TableFS [34]. Specially, AUGEFS increases the throughput by 34× on average for synchronous metadata updates and improves the throughput by up to 36% for LevelDB [22]. In summary, the contributions of this paper include:

- A detailed analysis of challenges when building userspace log-structured file systems on modern SSDs.
- A shared and protected address space in userspace to hold the file system with efficient control plane and data plane.
- A scalable KV store to organize metadata with parallel request processing and fine-grained parallel WAL to achieve high metadata scalability and performance.
- A domain-based file organization to enable in-parallel file access without cross-domain contention and an asynchronous IO stack for *fsync* to accelerate data persistence.
- An extensive evaluation of AUGEFS to show its efficiency over state-of-the-art file systems on modern SSDs.

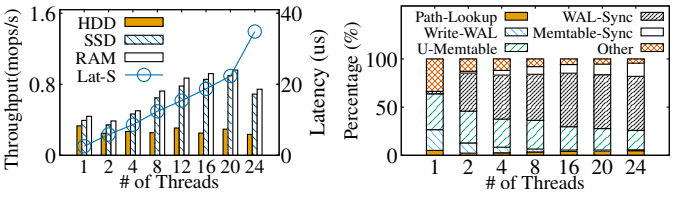
II. BACKGROUND AND MOTIVATION

A. Modern SSDs and Userspace LFS

The development of modern SSDs exhibits three attractive features, which inspire us to build a scalable and high-performance userspace LFS. First, modern SSDs provide sub-ten microseconds of IO latency, leading the kernel IO stack to become the performance bottleneck [44], [58]. Thus, building a userspace FS can reduce the heavy kernel-involvement overhead and improve performance [7], [58], [59], [76], [95]. Second, modern SSDs still exhibit higher sequential performance than random performance. For example, Samsung SZ1735a Z-SSD [83] offers a 4.1GB/s bandwidth for sequential writes while only 330K IOPS for random writes. Even the Intel Optane SSD obtains better sequential performance (10% on average) than random performance [92]. LFS [82] can convert random writes into sequential writes, thereby fully exploiting the sequential performance of modern SSDs. Third, a new feature named persistent memory region (PMR) [69] is proposed starting from NVMe specification 1.4 [70] for modern SSDs. PMR is a designated small-capacity persistent memory area within SSDs. It can be exposed as an internal memory area to applications by memory mapping (MMIO). Applications can access PMR directly through CPU load/store instructions [45], [53]. Thus, PMR raises an opportunity to accelerate small-sized (tens-to-hundreds-of-bytes) FS metadata access.

B. Inefficient Control Plane in Userspace FS

Userspace file systems are usually designed with the separation of control plane and data plane [7], [42], [58], [76], [87]. The control plane generally includes metadata operations and concurrency control, especially in cross-process cases. The data plane mainly involves reading/writing file data. Existing userspace FSs execute data plane operations in userspace to reduce the kernel overhead. However, most of them depend on the kernel [7], [76] or a trusted process [42], [87] to execute control plane operations. This results in an inefficient control plane as metadata operations require either trapping into the slow kernel or using expensive inter-process communication (IPC, e.g., socket-based IPC [42]) to interact with the trusted process. Unfortunately, write operations also slow down as they are intertwined with metadata updates (e.g., space allocation). Moreover, their control plane introduces inefficient cross-process sharing. For example, when multiple processes read/write the same file concurrently, multiple expensive IPCs are required [42] to acquire and release the



(a) Create Performance (b) Create Performance breakdown

Fig. 1: Performance of Create and Its Breakdown

inode lock (lease mechanism), which add high communication overhead and lead to lower performance [17], [80].

A few works [58], [59] adopt a semi-microkernel approach to place both the data and control planes in a trusted userspace process. However, they bring additional CPU overhead. uFS [59] is such a userspace FS that consolidates all FS functionalities into a trusted server (process). Application threads use the shared ring buffer to interact with the server. They write FS requests into the shared buffer and wait for completion by polling the buffer. The server threads poll the shared buffer to fetch requests and use the polling-based SPDK to access SSDs to serve the requests. This results in a *polling on polling* problem because both application and server threads spend too much time on polling. For example, uFS requires up to 80% additional CPU cores [59] for running the trusted file system process when running LevelDB [22].

C. Poor Metadata Scalability

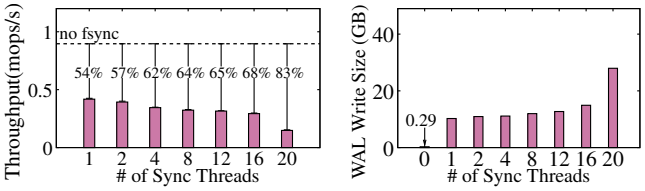
Metadata operations are critical to file system performance as they are reported to occupy nearly half of all FS operations [46], [81]. Metadata/fsync-intensive applications, such as mail server and rsync [40], demand highly scalable metadata performance [16]. Unfortunately, metadata operations often incur small and random writes, resulting in large amplification and suboptimal performance. To mitigate this, some works [60], [63] have introduced optimizations (e.g., inverted index) on traditional file systems. However, these approaches still involve many extra IOs for metadata persistence due to the mismatched granularity. Moreover, they may cause serious delays for guaranteeing crash consistency due to IO ordering [9], [13], [90] and journaling [16], [55].

There is another kind of file systems that utilize LSM-KVS (e.g., LevelDB [22] and RocksDB [20]) to organize metadata within local file systems [2], [34], [40], [85]. We argue that the LSM-KVS is well-suited to manage FS metadata. First, LSM-KVS groups small-sized metadata and writes them to the device sequentially in bulk, which is favored by modern SSDs. We observe that directly using RocksDB to organize FS metadata shows up to 5× performance improvement than traditional FSs (detail in §VII-A). Second, LSM-KVS can expose atomicity to support transactions considerably easier than the block interface [40], which can help support FS metadata consistency. However, the integration of LSM-KVS into FS metadata organization presents two challenges. First, LSM-KVS requires building atop a file system. For example, TableFS [34] is built atop Ext4 [4]. Fortunately, LSM-KVS does not require a complex FS. Instead, we can build a simple FS to manage its files (e.g., SSTables), greatly reducing the integration complexity. Second, we find a fundamental design: group updating adopted by existing LSM-KVS [6], [8], [11], [20], [22], [62], [79], [96] causes the false sharing problem, which severely limits metadata scalability. We choose the widely used and mature RocksDB as a representative LSM-KVS to illustrate this problem below.

False Sharing in Metadata Updates. When different threads update metadata in their private directories, there should be little dependence among them. However, we identify that the group updating mechanism requires synchronization among these threads. It causes high latency and constrains the scalability of metadata operations, leading to the false sharing problem. To display it, we first build a library FS and use the RocksDB to store its metadata. It adopts the metadata to KV mapping policy in §V and its KV size is about 192B. Then, we measure the throughput and average latency of its *create* operation with increasing thread count. Each thread creates files in its private directory. We run this experiment on HDD, Intel P4800X SSD [28], and RamDisk respectively with the machine in §VII, and show the results in Fig. 1(a). When running on HDD, the tested FS shows low throughput and little scalability as the slow device is the bottleneck. As for SSD and RamDisk, their throughputs scale initially, but their scalability is limited. When reaching 24 threads, the latency running on SSD (Lat-S) increases sharply, and the throughput even drops. Since RamDisk offers bandwidth that is ten times higher than that of SSD, it still encounters scalability limitation. This indicates that increasing storage device bandwidth does not improve scalability and throughput, instead, the performance bottleneck shifts from the storage device to the CPU.

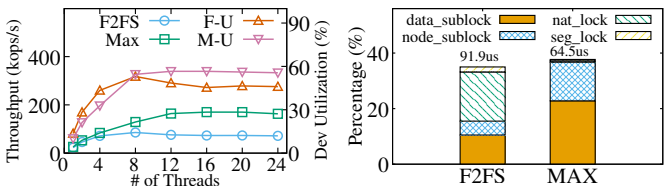
To verify the above claim, we describe the procedures of the *create* operation and illustrate its performance breakdown running on the SSD in Fig. 1(b). When multiple threads execute *create*, they first perform Path-Lookup, then issue a *put* request to RocksDB. The group updating mechanism organizes these threads into a group. One thread is elected as the leader that is responsible for aggregating and writing all log entries into the WAL file (Write-WAL). Other threads are followers and wait until WAL writes are finished (WAL-Sync). After that, these threads concurrently update the Memtable (U-Memtable). The thread group exits synchronously after all threads finish U-Memtable (Memtable-Sync). As in Fig. 1(b), when the thread count increases, Write-WAL and U-Memtable occupy less overhead (decreases from 65% to 15%). However, the group synchronization overhead (WAL-Sync and Memtable-Sync) increases a lot, contributing up to 78% latency. It is mainly from waiting among these threads and thread invoking (*yield* occupies 60% of the WAL-Sync time).

False Sharing in Metadata Persistence. When using LSM-KVS to store FS metadata, the updates are first written to the WAL file but may reside in the page cache. Thus, guaranteeing metadata persistence requires flushing the WAL file. For example, TableFS [34] provides the *fsync* operation that *put*s a KV pair with a sync write option to flush the



(a) Create Performance With Fsync (b) Device Write Size of WAL

Fig. 2: Performance of Create with Fsync



(a) Synchronous Write Performance (b) Lock Overhead in Fsync

Fig. 3: Performance of Write Operations

WAL. However, the metadata of a single directory may lie in different positions of the WAL file. Even persisting one directory's private metadata requires flushing the entire WAL, resulting in false sharing on WAL. To show its impact, we add the *fsync* operation atop the above *create* workload with 20 threads. Fig. 2(a) shows the results. The X-axis indicates the thread count executing *fsync* within all threads. For example, one sync thread means only one thread executes *create-fsync* while the others only execute *create*. We observe that only one sync thread can severely degrade the whole system performance, up to 54% degradation compared to that all threads executing *create* without *fsync*. With the increasing count of sync-threads, the throughput decreases by up to 83%. Thus, false sharing on WAL hurts performance heavily when file system metadata is persisted synchronously.

To analyze the aforementioned problem further, we measure the device write size caused by WAL (*wal_dev_write_size*) and show the results in Fig. 2(b). When there is no sync thread, *wal_dev_write_size* is only 290MB, which is far less than the size of workloads (about 2.5GB). The reason is that most WAL files are deleted before they are persisted to SSD as their corresponding Memtables have been flushed. However, when there is only one sync thread, *wal_dev_write_size* increases sharply to 10.2GB, and it can increase up to about 28GB with all 20 sync threads. Large *wal_dev_write_size* brings additional device IOs and increased latency. When one thread executes *fsync*, its updating group suffers from the long latency of device IOs, as it requires flushing the whole WAL. Meanwhile, other updating groups must wait for a long time until the previous group exits, which causes severe performance degradation by blocking the whole process.

D. Underutilized Device Bandwidth of Data Operations

Data operations can execute faster within the userspace FS architecture by directly accessing storage devices [7]. However, we find that the device bandwidth is underutilized with existing LFSs's data management, especially for synchronous writes that are commonly used for data durability and storage order [90], [91]. To illustrate this problem, we measure the write throughputs of the state-of-the-art LFSs (F2FS [10] and Max [50], [52]) on the above SSD with increasing thread count. Each thread executes 4KB append writes to its private file followed by *fsync* independently. We also measure the device bandwidth utilization (calculated as the average device used bandwidth ÷ device maximum bandwidth). Fig. 3(a) shows the results, where F-U and M-U denote the bandwidth utilizations of F2FS and Max respectively. We find the performance of Max only scales up to 12 threads, failing to fully saturate the device bandwidth (up to 60%). We analyze the reasons for the aforementioned results as below.

Intrinsic Dependency in Data Persistence. When persisting dirty data explicitly with *fsync*, both Max and F2FS require the modified data blocks to be durable before their associated node blocks to guarantee correctness and data consistency. However, this strict write order introduces a serial execution path and intrinsic dependencies, which lead to unnecessary waiting time and hurts the device's parallel performance.

Lock Contention under High Concurrency. The shared in-memory data structures, such as shared node address table (NAT) and segment information table (SIT) in F2FS, cause severe lock contention under high concurrency. Although existing works (e.g., Max) split the shared data structure to eliminate most contention, we still find the mutual submitting locks to guarantee write ordering brings underutilization of device bandwidth. For example, the data log in F2FS has a reader-writer lock, which serializes the data submitting stage. It also uses a global mutex lock in superblock to further serialize IO. Although Max [50], [52] splits the large log into minor logs, it only focuses and implements on the space allocation stage. When submitting data blocks, it still simply locks the whole stream. Moreover, all data logs in Max share a global submitting lock, which hinders its scalability. The submitting locks include the data log submitting lock (data_sublock) and the node log submitting lock (node_sublock). To verify it, we isolate the overhead proportion of locking in *fsync* under 24 threads. As shown in Fig. 3(b), the latency of locks in Max is lower than that in F2FS (from 91.9us to 64.5us). However, the entire lock overhead in Max still occupies 37.7% of the *fsync* operation, mainly from the submitting locks.

III. Design Overview

To address the above challenges, we propose AugeFS, a scalable and high-performance userspace LFS. Fig. 4 presents a high-level overview of AugeFS design, which mainly contains the following components:

Architecture. To provide an efficient control plane without extra CPU overhead, AugeFS places the file system in a shared and protected address space within the user address space for all application processes accessing it. Applications access AugeFS through the userspace library (LibFS), which only requires a fast userspace context switch. Additionally, it provides the hash-based inode cache, the hash-based dentry

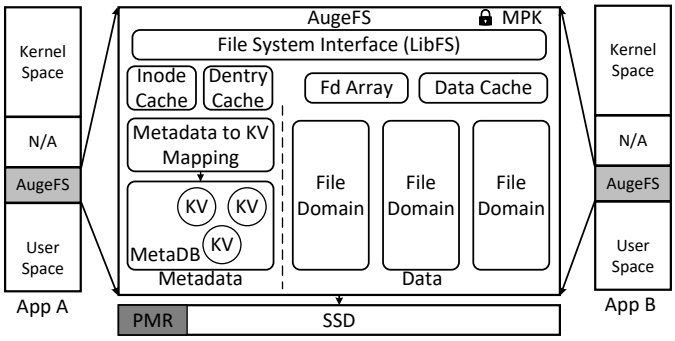


Fig. 4: The Overview of AUGEFS

cache, the radix-tree-based file data cache, and the run-time file management (Fd array) similar to the virtual file system. For protection, AUGEFS leverages memory protection technology (e.g., Intel MPK) to ensure its integrity (§IV-B).

METADB. METADB is a scalable LSM-based KV store that is responsible for storing the file system metadata, especially for inodes. Metadata operations such as *create* and *mkdir* are converted into KV operations (i.e., *put*, *get*, *delete*), which can handle small-sized metadata more efficiently. To further improve the scalability of metadata operations, METADB adopts parallel request processing (§V-A) to reduce the thread synchronization overhead and fine-grained parallel WAL (§V-B) to eliminate the false sharing on WAL.

File Domains. Thanks to the adoption of METADB, AUGEFS enables scalable inode and directory operations through METADB. Then, AUGEFS distributes files into different domains to achieve high data scalability (§VI-A). Each domain includes a collection of files and maintains its own space management metadata. AUGEFS also conducts garbage collection and checkpoint in a per-domain way. Moreover, AUGEFS provides an asynchronous IO stack for *fsync* to accelerate data persistence in each domain (§VI-B).

IV. AUGEFS ARCHITECTURE

A. Efficient Control Plane in AUGEFS

At first, AUGEFS is designed as a userspace library FS [95], which can avoid both the IPC and kernel trapping overhead for the control plane. Unfortunately, this design does not support cross-process sharing despite obtaining promising performance. To overcome this problem, we propose architecting AUGEFS as a shared and protected address space, a shared part within the user address space for AUGEFS specially. Fig. 5 illustrates the address space layout of an application using AUGEFS. Since the virtual address space provides up to 64PB space range [57], we reserve a fixed and enough range (i.e., AUGEFS in Fig. 5) from the user address space to hold AUGEFS. Similar to the kernel address space, AUGEFS address space has code segment (the library file system), data segment, stack, heap, background threads, etc.

Each application accessing AUGEFS must register with the kernel. During the registration, the application invokes a new system call *augefs_init*, then the kernel checks applications' permissions and maps AUGEFS into the applications' address space by adding the page table entries of AUGEFS into the applications' page table. Thus, any registered application can share AUGEFS akin to sharing the kernel address space. Then, multiple threads (both intra- and inter-process) can manage the same data structures in AUGEFS, and lock structures (e.g., inode lock) can be built to support concurrent accesses among them. Applications access AUGEFS via a library called LibFS, which offers file system interfaces like *open* and *close*. Since AUGEFS resides in the same address space within applications, application threads can utilize the non-privileged function calls (*call* and *ret* instructions) to execute AUGEFS functions entirely in userspace like the library FS. Meanwhile, AUGEFS does not require extra server threads to execute functions on behalf of application threads. Therefore, AUGEFS obtains the library-file-system-like high performance while supporting cross-process sharing, which provides both efficient control plane and data plane.

B. Protection for Shared User Address Space

Since AUGEFS is designed as a part of user address space, we must concern about the corruption resulting from stray writes [19]. Stray writes typically happen when the control flow is messed up due to bugs outside AUGEFS [17], where misused pointers can modify the contents in arbitrary memory addresses in AUGEFS to hurt the FS integrity. Some works [17], [49] use Intel memory protection keys (MPK) to prevent stray writes to persistent memory, but they cannot directly be applied to AUGEFS as they either protect specific abstraction (coffer in [17]) or lack protection for volatile data structures (e.g., inode cache, page cache) [17], [49]. Thus, we co-design AUGEFS architecture with MPK and introduce a protection technique called MPP (i.e., Memory Protection for address sPace). With the help of MPP, AUGEFS can expand its protection boundary to a trusted process group [99] as in the data center environment [49].

Intel MPK. MPK is used to restrict memory accesses in userspace. It leverages four unused bits in page table entries to store a protection key, giving 16 possible keys. The pages sharing the same key are considered as one memory region. To control access permission of each memory region, MPK introduces a new userspace-accessible CPU register named PKRU. This register contains 16 pairs of two-bit permissions for these keys, determining the corresponding regions' rights (read, write, neither, or both). The permissions can be changed by a user-mode instruction WRPKRU. Moreover, its permission control is at the per-thread granularity.

MPP. Its key idea is to guarantee the AUGEFS address space to be exclusively accessible only during the invocation of file system services. When executing the application's code outside AUGEFS, application threads are prohibited from accessing AUGEFS address space, thereby preventing stray writes. To achieve it, we reserve the last memory protection key (*pkey*) to control the access permission of AUGEFS, treating AUGEFS as one memory region. After registration, one application is not allowed to access AUGEFS by default. Then,

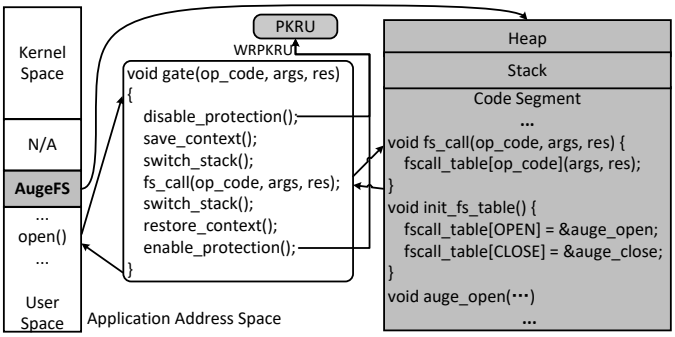


Fig. 5: The Example of AUGEFS Address Space

we design a *gate* function to wrap each file system call of AUGEFS to manage the permission. The *gate* function resides in a code page that is mapped into the applications' address space at registration. When an application thread invokes a file system call, the *gate* function first grants the access permission of AUGEFS by updating the PKRU (*disable_protection*). The instruction WRPKRU is executed fast (less than 20 cycles [73]) without trapping into the kernel. After the FS service finishes, the *gate* function disables the accessibility of AUGEFS again (*enable_protection*). Thus, AUGEFS is inaccessible when the application's code is executed. Since AUGEFS code (i.e., LibFS) is a trusted library, there are no stray writes corrupting AUGEFS. Note that MPP is not specific to Intel processors, as similar hardware features exist in other processors for memory protection, such as ARM Domain [3] and IBM Storage Protection [26].

Malicious Attacks. AUGEFS still faces malicious attacks [99] from untrusted processes (e.g., modifying *pkey*) as in previous works [17], [49]. Fortunately, there are several orthogonal works [25], [86] exploring low-overhead binary-writing techniques. They can be directly applied to AUGEFS to protect against malicious attacks while having little impact on the performance of both AUGEFS and applications [25], [86], which leaves as our immediate future work.

C. Put It Together

Here, we take the process of opening a file as an example, as shown in Fig. 5. First, an application thread calls the *open* function, which calls the *gate* function subsequently. Second, within the *gate* function, it calls *disable_protection* to gain access to AUGEFS address space. It then saves the application thread context and switches to the stack in AUGEFS. Third, it calls *fs_call* to execute the actual FS function (*auge_open*) according to the *opcode* (OPEN). The *fs_call* function is placed in a fixed address in AUGEFS, and the *gate* can call it using the function pointer. Note that AUGEFS pre-initializes the *fscall_table* in *init_fscall_table*. Finally, it switches to the application thread stack, restores the application thread context, and calls *enable_protection* to revoke the access permission to AUGEFS.

V. METADB

In this section, we present METADB, a scalable LSM-KVS for packing metadata of AUGEFS. To improve its scalability, we design two techniques: parallel request processing (§V-A) and fine-grained parallel WAL (§V-B). We first describe how to apply METADB to AUGEFS.

Metadata to KV Mapping. AUGEFS stores FS-level metadata (superblock) and file-level metadata (e.g., inode) into METADB. The key of superblock is *m:superblock*. We adopt similar naming rules for the key of file-level metadata as existing works [34], [40]. Specifically, AUGEFS assigns the key of file-level metadata by combining the prefix 'm', the inode number of the parent directory, a delimiter (:), and the file or directory name. For example, if the parent directory of *foo.txt* is */home/dir* whose inode number is 100, the key for *foo.txt*'s metadata is set as *m:100:foo.txt*, with its value being the inode of *foo.txt*. AUGEFS also adopts an in-memory inode cache to accelerate metadata accessing [34]. METADB provides several APIs to serve metadata operations. For example, *create* is converted to *put* operation, *unlink* is converted to *delete* operation, and *rename* is converted to *delete* and *put* operations. AUGEFS has no dentries in METADB and it uses the *iterate* function to organize the hierarchical namespace, which can list all files and directories in a parent directory [34], [40].

A. Parallel Request Processing

To eliminate thread synchronization of the group updating mechanism, METADB introduces parallel request processing to enable each thread to update Memtable and WAL independently without considering the status of other threads. Thus, different threads in AUGEFS can update file system metadata in METADB independently instead of being in a co-dependent group. To support parallel request processing, METADB adopts the skiplist-based Memtable to allow concurrent updating [20], which supports KV insertions from multiple threads. Meanwhile, METADB creates a separate WAL file for each CPU core to eliminate the dependence on WAL writes from different threads. When executing *put*, METADB divides the process into four stages, as shown in Fig. 6. ①Preparing: a thread checks whether the current Memtable is full. If full, it switches both the Memtable and the WAL files. Then, the thread obtains a sequence number of the current write batch. ②Writing Log: the thread gets the running CPU ID and updates the corresponding WAL file. ③Updating Memtable. ④Return. The stages ② to ④ can be executed independently by different threads. Instead, the stage ① is protected by a global mutex. Fortunately, the Memtable and WAL switching occurs infrequently, thus stage ① cannot become a bottleneck for concurrent updates. By doing so, metadata updates in AUGEFS avoid thread synchronization overhead, improving the metadata scalability significantly, as in §VII.

Correctness Guarantee. Compared with existing LSM-KVS, METADB compromises the strict sequential updating order requirement, which brings two potential issues. First, during the preparing stage, when the active Memtable is checked as fully filled and switched to be immutable, there may exist in-flight KV pairs that have acquired the position of the active Memtable but not finished the insertion due to the non-strict sequential order. These KV pairs risk being lost when this

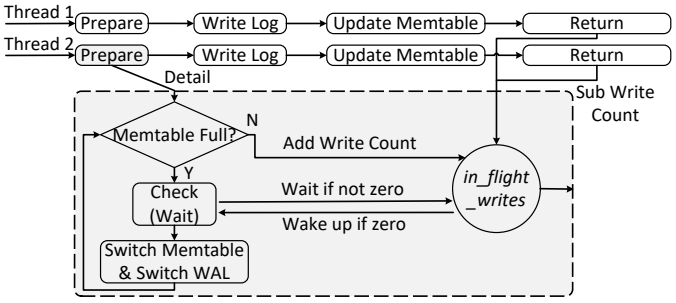


Fig. 6: Parallel Request Processing

immutable Memtable is flushed by background threads. To address it, METADB introduces a synchronization barrier between two adjacent Memtables. Its key idea is before an active Memtable is switched to be immutable, it should wait until all in-flight updates belonging to it are finished. METADB uses a semaphore primitive with an atomic number (*in_flight_writes*) for each Memtable to count in-flight KV pairs. In the preparing stage, the *in_flight_writes* is added to the count of KV pairs in the current write batch. After the KV pairs are inserted into the Memtable, the *in_flight_writes* counter subtracts the write count. When switching an active Memtable into an immutable one, METADB checks *in_flight_writes* and waits if it is not zero. When another thread finishes insertion and finds *in_flight_writes* becoming zero after subscribing, it wakes up the waiting threads. Other threads cannot proceed until the Memtable switching is finished. Since the synchronization barrier only sits between two adjacent Memtables, it has little impact on scalability as concurrent updates to the same Memtable can still execute without waiting for each other.

Second, the sequence number (seq_num) is used to identify different versions of KV pairs [18] in LSM-KVS. Seq-num is generated and incremented on each update in METADB. Thus, all KV pairs are logically arranged in a sorted order. Meanwhile, multiple KV pairs with the same key may co-exist and can be differentiated by the seq_nums. METADB violates this order as the Memtable inserting order may be inconsistent with seq_num. That means AUGEFS may not read the latest value with the latest seq_num from METADB when updating the same key simultaneously. Fortunately, this issue can be addressed by the specific feature of the file system. In the POSIX file system, the correctness of metadata concurrent updating is guaranteed by inode locks. For example, when two threads perform *mkdir* and *rmdir* to the same directory simultaneously, their order is protected by the inode lock of its parent directory, and they are executed in order. Thus, there are no simultaneous updates to the same key in METADB. In conclusion, parallel request processing does not sacrifice correctness and has the same consistency level (similar to the ordered mode in Ext4) with other KVS-based file systems (e.g., TableFS).

B. Fine-Grained Parallel WAL

Although METADB employs parallel request processing with per-core WAL, it still suffers false sharing in metadata persistence. The aforementioned false sharing on a single WAL file turns to false sharing on multiple WAL files. For example, when one directory is invoked *fsync*, its metadata may lie in multiple WAL files due to concurrent metadata updates to this directory from multiple threads. To guarantee persistence, METADB must flush all WAL files synchronously. This procedure hurts performance severely, similar to §II-C. To solve this, METADB utilizes the persistence memory region (PMR) [70] in SSDs to build a fine-grained parallel WAL.

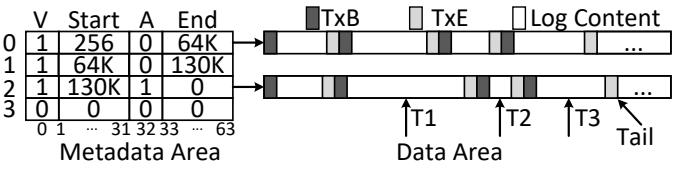


Fig. 7: Fine-Grained Parallel WAL

Enabling Parallel Logging. Different from the block interface, PMR allows byte-addressable and durable access. This allows the small-sized log of FS metadata to be persisted immediately in a fine-grained granularity. Thus, METADB introduces parallel logging to eliminate false sharing in metadata persistence. First, METADB maintains a log tail in DRAM, and concurrent threads use the tail as a synchronization point. Concurrent threads use the compare-and-swap (CAS) instruction to increase the tail atomically. Since the threads know the log entry size before logging, they can write log entries to their positions in parallel without dependence. Second, one log entry is 64B-aligned by adding paddings. The first 8B is its TxB (transaction begin) area, which includes 4B for a magic number and 4B for length. The last 8B is its TxE (transaction end) that is used to record the commit information. There is the log content between TxB and TxE. TxB, the log content, and TxE are written sequentially for recovery. Note that parallel logging does not compromise the correctness of conflicting metadata updates because they are still protected by FS locks (i.e., the locks of superblock and inode).

Log Data Management. Directly using PMR to store WAL complicates log data management without the help of a file system. Fortunately, with the byte-addressability and durability features of PMR, METADB can organize fine-grained parallel WAL with little overhead. METADB divides the PMR region into two parts: a large data area and a small metadata area. The data area stores WAL "files". For each mutable Memtable, METADB allocates a continuous data area to hold its logs. The Memtables are flushed following their born order. Thus, the data area for each Memtable is allocated and freed in a log-structured way without incurring space fragmentation. The metadata area resides at the beginning of PMR, which stores metadata of WAL "files" such as their beginning and ending offsets. Fig. 7 shows the overview of WAL on PMR for METADB. The metadata of each WAL "file" occupies an 8-Byte slot, which consists of 1 valid bit (V), 31 start address bits, 1 active bit (A), and 31 end address bits. The valid bit shows the availability of the slot, and the active bit shows whether the active Memtable uses the slot. The start and end address bits describe the address range on PMR of the WAL

"file". Fig. 7 shows the WAL "files" of three Memtables: one active (mutable) and two inactive (immutable). The end address of the active one is zero as it is being updated. When becoming an immutable one, its end address is updated. After a Memtable is flushed, its data area can be reclaimed, and the valid bit of its metadata slot is also reset. Note that each update to an 8-Byte slot can be atomic for PMR [1], [75].

PMR Size. The size of the metadata area is set to 4KB, and the size of the data area is related to the number of Memtables instead of the SSD's capacity. By default, the data area requires about 128MB to hold logs from two Memtables, each of which is 64MB (same as RocksDB). Other information (TxB, TxE, and paddings) requires extra but much smaller space. A large PMR (hundreds of MBs) can directly persist logs without flushing logs into SSDs to maximize performance and previous works [32], [45], [47] show this PMR size is feasible. However, a small PMR (e.g., 2MB used in prior work [53]) is still able to serve logs for MetaDB efficiently. To achieve this, we reserve a few hundreds of MBs (e.g., 256MB) on SSDs as the backed space of PMR. We equally divide the PMR into two parts. Initially, the first part receives the written logs. Once the first part is full, its logs are asynchronously migrated into the backed area on SSD. Meanwhile, the second part of PMR is used to receive log writes. As for a small PMR, the full part can be quickly migrated into the underlying fast SSD. Then, MetaDB repeats the above process to continue using the first part after its migration finishes. Thus, the divided parts are used by turns, which allows MetaDB to still highly benefit from our optimizations, as shown in §VII-F.

Log Recovery. First, MetaDB reads the metadata area to fetch the valid Memtables and their address ranges. Second, MetaDB recovers the logs of inactive Memtables by re-doing them as they are complete. Third, for the active Memtable, its log entries may be incomplete due to parallel logging. To detect them, during the WAL initialization or WAL region reclamation, MetaDB zeros out the corresponding space. Then, MetaDB scans the data area from the start address of the active Memtable at the 64B granularity until a TxB is found. The log entry length is fetched from TxB, and MetaDB uses it to check whether TxE exists. If it exists, this log entry is complete and can be recovered. Otherwise, it is dropped. MetaDB continues scanning and repeating the above procedures until reaching the address of an immutable Memtable or traversing the whole area. This procedure is fast due to the small size of the WAL region.

Metadata Consistency: MetaDB first writes logs of metadata to PMR synchronously using fine-grained persisted instructions (store + cflush [53]), which provides immediate persistence for logs. Then, MetaDB updates the volatile Memtable. The metadata can be recovered due to its durable logs even meeting crash down when updating the Memtable. Some directory operations, such as *rename*, are converted into more than one KV pair. MetaDB wraps the KV pairs into a write batch and uses the above WAL to guarantee its atomicity and durability because they are in one transaction unit. Thus, AugeFS can persist metadata atomically and immediately without any consistency problems.

Discussion. While current commercial SSDs do not support PMR, we believe it will be a key feature in future SSDs. First, several recent works [5], [45], [51], [53], [54], [93] show that PMR is able to accelerate system performance, which evidences its importance for SSD-based storage systems. Second, existing technologies like non-volatile memory technologies (e.g., 3D XPoint [27] or capacitor-backed DRAM [32], [36], provide medium-level support for realizing PMR in SSDs. Finally, PMR requires no DIMM slots and has a smaller capacity (hundreds of MBs or less) than persistent memory (e.g., Intel Optane Persistent Memory [31]), which makes it more practical in terms of cost and power saving.

VI. Data Management

A. Domain-Based File Organization

The scalability of data accessing in LFSs is usually limited by the contentions on manipulating FS-level shared data structures (as in §II-D). Dividing the whole FS into multiple partitions is an effective approach to reduce such contention. However, existing partitioning approaches [35], [52] either require extra effort to guarantee cross-partition metadata consistency [35], [52] or suffer from global checkpoint locking because different files may share the same segment [52]. Thanks to the centralized and scalable MetaDB, AugeFS avoids the cross-partition metadata consistency issue. Then, AugeFS designs a domain-based file organization to distribute files into different partitions (called *domain* here). Each domain can execute *read*/*write* operations, garbage collection, and checkpoint independently, which enables in-parallel file access with little cross-domain contention. Thus, AugeFS eliminates the locking contention completely for synchronous writes, including contention on the shared data structures and the submitting stage. AugeFS adopts a modular-based hash function to distribute files into domains evenly, i.e., $D_f = Hash(ino)\%N$, where D_f is the domain ID of file f, ino is the inode number of file f, and N is the total number of domains (the default number of domains is 16 as in §VII-F).

Space Management. AugeFS divides the device space into fixed-size segments (2MB), which are the space allocation granularity of domains. Each segment is free or owned by one domain exclusively. AugeFS reuses the three data structures of F2FS: segment information table (SIT), node address table (NAT), and segment summary area (SSA). To avoid cross-domain contention when manipulating the three data structures, AugeFS lets each domain maintain its own structures. Moreover, we apply some modifications to SIT and NAT, but leave SSA unchanged as each SSA (4KB block) belongs to a segment that is not shared among domains.

Since SIT contains per-segment information (e.g., the number of valid blocks and the validity bitmaps), we add a domain ID in each SIT entry (representing one segment) to indicate the ownership of the segment. To avoid updating sharing between SIT entries of different domains, we store the on-disk SIT entries in MetaDB. The key of a SIT item is the combination of a flag *sit*, the segment ID, and a checkpoint

version number (e.g., *sit:0:1* for segment 0 in version 1). The version number is used for roll-back recovery to the latest consistent checkpoint. The value of a SIT item is the segment entry. As for NAT, the NAT entries from different domains may co-locate in the same block as NAT originally adopts per-entry allocation granularity. Instead, AUGEFS allocates NAT entries in a block granularity. In such doing, a NAT block can only contain entries belonging to the same domain, which avoids block sharing when persisting NAT entries. Meanwhile, AUGEFS uses two in-memory radix trees for SIT and NAT separately in each domain to accelerate their lookup. Since the contents in SIT and NAT are partitioned into different domains, the per-domain radix trees bring little extra memory overhead as they only index a subset. Through this partition, different domains nearly share no data structures, which mitigates their contention significantly. AUGEFS still adopts multi-head logging to support log-structured writing, where each domain keeps data logs and node logs. Note that AUGEFS maintains one active data segment and one active node segment to accept new updates for each domain.

Garbage Collection. The garbage collection of AUGEFS first selects the domain with the highest garbage ratio. Then, AUGEFS performs garbage collection on the selected domain, including victim selection and block identification and migration. The garbage collection granularity is the segment whose SIT maintains a validity bitmap for detecting the valid blocks. After all valid blocks are migrated, the selected segment is registered as a free segment candidate. It finally becomes free after a checkpoint and can be allocated by domains again.

Crash Consistency and Recovery. AUGEFS uses checkpoint to provide a consistent recovery point after a crash. Each domain can do its own checkpoint as it manages its own SIT, NAT, SSA, and other metadata (e.g., current active segment information) in both in-memory and on-disk data structures, reducing contention on the global checkpoint lock. To maintain a consistent state of the whole file system across domains (e.g., the *sync* operation), AugeFS performs checkpoint on all domains that are protected by a global lock. Although the checkpoints are executed more frequently, the size of each checkpoint is smaller as each domain is only responsible for its own data checkpoint. For consistency between metadata and data, AUGEFS uses the ordered mode that persists data and node blocks first and then updates the inode. Different from the serial recovery in F2FS, AUGEFS runs the recovery procedure in parallel for each domain to accelerate its recovery. After a sudden power-off, AUGEFS rolls back to the latest consistent checkpoint and performs roll-forward recovery on the data and node logs.

Discussion. Since AUGEFS adopts a hash-based static partition approach, it inherits a general limitation from the partition approach: access skewness across domains, which can make AUGEFS degenerate to F2FS in the worst case. Several optimizations [59], [61], [74] can be used to mitigate this limitation. For example, we can adopt a dynamic partition with file ownership migration between domains to avoid access skewness, and we leave it as the future work. Note that there is no resource contention due to skewness in AUGEFS because each domain allocates resources (e.g., segments) on demand and AUGEFS does not reserve space for each domain. Additionally, AUGEFS makes it easier to eliminate dependence completely as it assigns one file only to one domain, and one file operation only involves one domain.

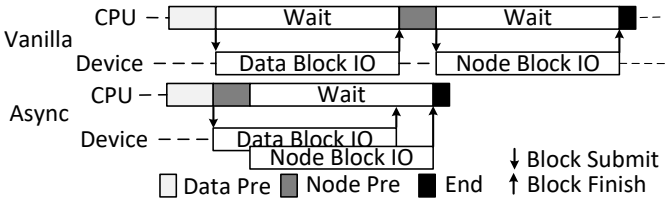


Fig. 8: Asynchronous Data Stack for Fsync

B. Asynchronous IO Stack for Fsync

Current LFSs such as F2FS and Max employ the roll-forward recovery that only writes data blocks and their direct node blocks to enhance *fsync* performance. However, ensuring crash consistency requires data blocks to be durable before node blocks, which leads to intrinsic dependencies. Consequently, this mechanism underutilizes the multi-head logging design and cannot fully exploit the device bandwidth. AUGEFS designs an asynchronous IO stack for *fsync* that updates data and node blocks in parallel. Fig. 8 shows the *fsync* paths of vanilla F2FS and AUGEFS. Vanilla F2FS first prepares the write of data blocks, including getting node blocks, allocating data blocks, refreshing SIT, etc. Then, it submits the data block IO and waits for completion. After that, F2FS prepares the write of node blocks, including finding dirty node blocks, setting the *fsync* flag, etc. Then, it submits the node block IO and waits for completion. AUGEFS overlaps the data block IO and node block IO to reduce latency by utilizing the waiting time. After the data blocks are submitted to the device, AUGEFS begins to prepare and submit the node blocks. Finally, it waits for their completion together.

This naive design may bring inconsistency. When meeting a sudden crash, the data blocks may be lost while the node blocks have been persisted, whose indexes may point to garbage data. To handle this, we add a durable write version number (*write_ver*) for per-CPU core. We reserve a small area at the end of PMR for *write_ver* and each CPU core occupies 64 Bytes (several KBs totally). Before submitting the node blocks, the thread fetches *write_ver* of its running CPU core. It then adds a special flag (*fsync*), *write_ver*, and its running CPU ID inside direct node blocks. After the data block IO is finished, *write_ver* is increased. Since PMR is byte-addressable and low-latency, the version number can be updated with little overhead. The concurrency of this procedure is protected by a per-CPU lock. In the roll-forward recovery, AUGEFS collects the direct node blocks having the special flag, fetches the *write_ver* and CPU ID, and compares them with the corresponding *write_ver* on PMR. If equal or greater, the data blocks may not have been persisted, and the node block can be dropped. If less, the node block can be used for

recovery. The asynchronous IO stack does not compromise the crash consistency and keeps the same consistency guarantee with the roll-forward recovery of F2FS. It only accelerates the data persistence path in *fsync* and improves device utilization.

VII. Evaluation

In this section, we present evaluation results for AugeFS. We first use a set of microbenchmarks to evaluate the basic performance of AugeFS. Then, we show the performance of AugeFS under the macrobenchmark Filebench [21] and the real-world application LevelDB [22]. Finally, we evaluate the overhead of the control plane of AugeFS under multi-process workloads, and conduct a performance breakdown as well as sensitivity analysis for AugeFS.

Experiment Setup. We conduct all experiments on a server with two 24-core Intel(R) Xeon(R) Platinum 8260 CPUs, which runs CentOS 7 with Linux kernel version 4.19.11. Hyper-threading is disabled. The machine is equipped with 256GB DRAM, 512GB Optane persistent memory (PM), a 375GB Intel P4800X SSD, and a 2TB Seagate HDD.

Implementation. AugeFS mainly consists code segment, heap, and stack. We implement AugeFS in the userspace as a dynamic linker library to be shared among multiple processes, which is mapped into a fixed address space by modifying the linker. For heap, AugeFS uses a shared heap on top of a memory-mapped file with using a simple and efficient allocator [41]. For stack, AugeFS creates a new stack in its address space for new threads using AugeFS [25]. We implement a manager process, which is responsible for registration. Application threads communicate with the manager process for getting permission to access AugeFS (map the shared heap and stack). We assume the address range of AugeFS is enough (e.g., 1TB) and rarely used, and we can further modify the kernel to reserve the address range. It intercepts POSIX IO calls [42] and requires little modifications to applications.

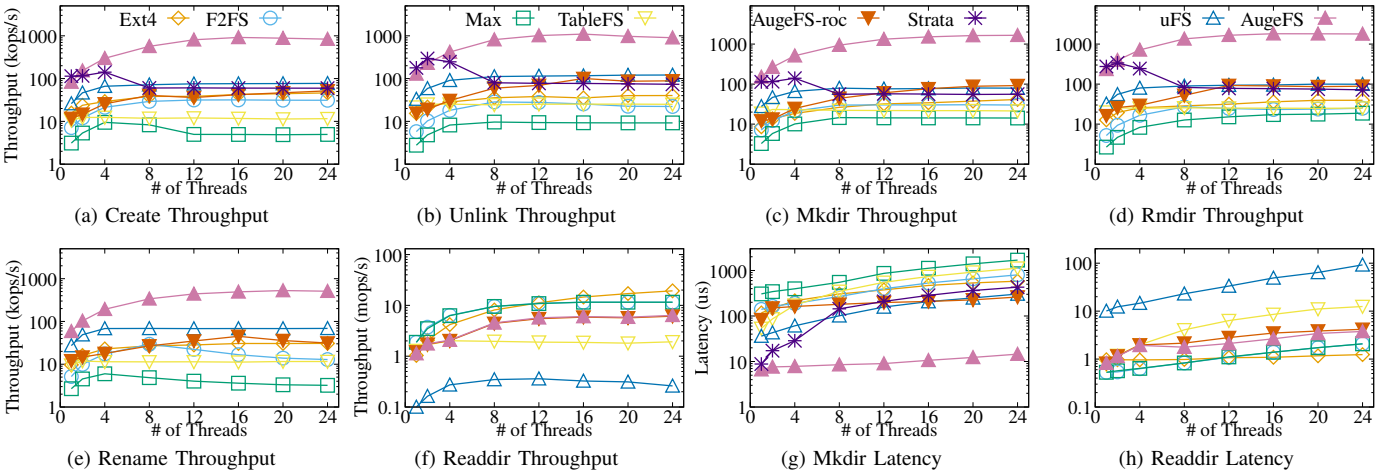
Since there is no available product with PMR, we reserve 256MB Optane PM to emulate it that can maximize the performance of AugeFS, and we also evaluate performance under limited PMR (§VII-F). Note that the 256MB size of PMR is reasonable as previous works [32], [45], [47]. To emulate PCIe latency, we add a 900ns [67], [80] software delay for per 64 Bytes PM accessing. We reserve a fixed SSD area (50GB by default) for MetaDB and build a simple file system for MetaDB to access it in userspace like TopFS in SpanDB [11]. A global space allocator between MetaDB and data management leaves as future work. Since SPDK works poorly for mutliple processes [11], AugeFS uses the performance-similar NVMeDirect [38] to access NVMe SSDs.

Compared Systems: We compare AugeFS with six FSs: Ext4 [4], F2FS [10], Max [52], TableFS [34], Strata [42], and uFS [59]. Ext4 is a popular and widely-used journaling FS. F2FS and Max are two state-of-the-art LFSs for SSDs. TableFS first adopts LSM-KVS to store metadata. Strata is a log-structured and cross-media FS involving PM and SSD. uFS is a recent high-performance userspace FS for SSDs. Except AugeFS, other FSs are not explicitly designed for PMR. We argue that only using PMR to build scalable metadata operations atop them is not direct: i) The size of PMR is limited, which is not sufficient to store all the metadata for an entire FS. ii) PMR tends to have high latency through PCIe, making it challenging to maintain complex data structures.

A. Microbenchmarks

Metadata Performance. We conduct six microbenchmarks to measure the throughput and latency under different thread counts. They create empty files (*create*) or directories (*mkdir*), delete empty files (*unlink*) or directories (*rmdir*), rename directories (*rename*), and iterate directories (*readdir*). Each thread performs metadata operations in private directories followed by *fsync* for persistence and executes 1M operations with a dir-width 10, where one directory has 10K files approximately. We add another compared system, AugeFS-roc, which is the same as AugeFS except using RocksDB to replace MetaDB. Fig. 9 shows the results.

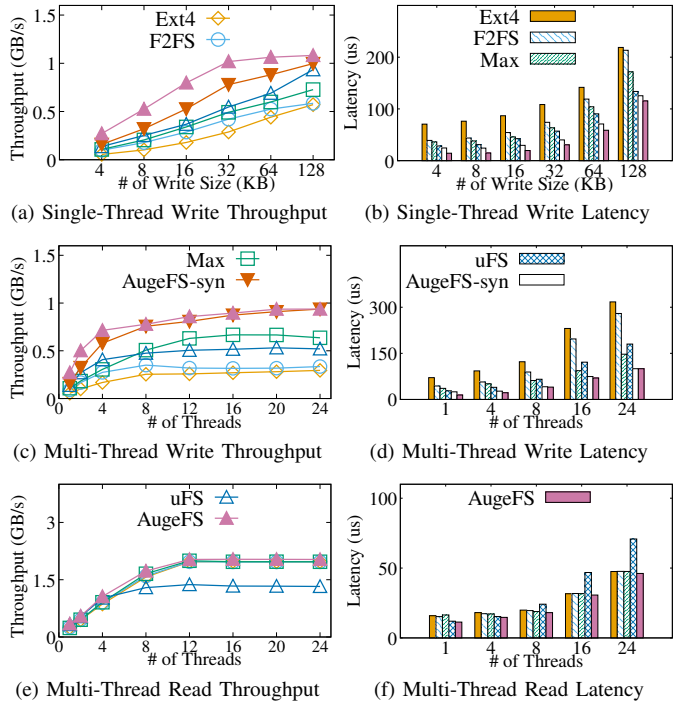
For metadata updates (Fig. 9 (a-e)), AugeFS performs best generally, outperforming other systems by 34× on average. It also scales well with increasing thread count, while others only scale to 8-12 threads. Simply using LSM-KVS to store metadata (AugeFS-roc) can also bring some improvement but it suffers from the false sharing problem as in §II-C. AugeFS adopts parallel request processing and fine-grained parallel WAL to handle this, which bring up to 35× improvement than AugeFS-roc. Note that the rmdir operation invokes the *delete* interface in MetaDB. The *delete* in MetaDB behaves similarly to *put* except its value is the delete flag. uFS performs better than others except AugeFS because it avoids coordination with logical journaling [59]. However, it meets the bottleneck in high thread count because it only uses one thread for directory operations. Max performs worse than F2FS because the submitting of its node or data blocks is serial even in different logs, but F2FS can batch metadata updates and submit them together. Ext4 performs poorly due to a centralized journal design. Strata has an outstanding and scalable performance in low thread count since it logs process-private updates in PM using the operation log pattern. However, it scales worse for two reasons. First, its digest operation brings double write amplification. Second, the socket-based communication between LibFS and KernFS of Strata is slow, which hurts performance. For *readdir* in Fig. 9(f), AugeFS, TableFS, and AugeFS-roc exhibit suboptimal performance because they require more block reads to retrieve values. Ext4 or F2FS stores the dentries of the same directory to contiguous blocks. However, the FSs based on LSM-KVS may place data in different SSTs and scattered blocks. This problem can be mitigated by re-inserting the KVS of the same directory [40] to place them in the same data blocks of one SST. uFS behaves worst because it has only one work thread to handle *readdir* and its communication overhead is heavy for read-intensive operations. We do not evaluate *readdir* and *rename* for Strata because it does not support *readdir* and can not run *rename* for multiple threads. For latency, we select *mkdir* and *readdir* and show their results in Fig. 9(g) and 9(h). AugeFS has the



(a) Create Throughput (b) Unlink Throughput (c) Mkdir Throughput (d) Rmdir Throughput

(e) Rename Throughput (f) Readdir Throughput (g) Mkdir Latency (h) Readdir Latency

Fig. 9: Performance of Metadata Operations



(a) Single-Thread Write Throughput (b) Single-Thread Write Latency

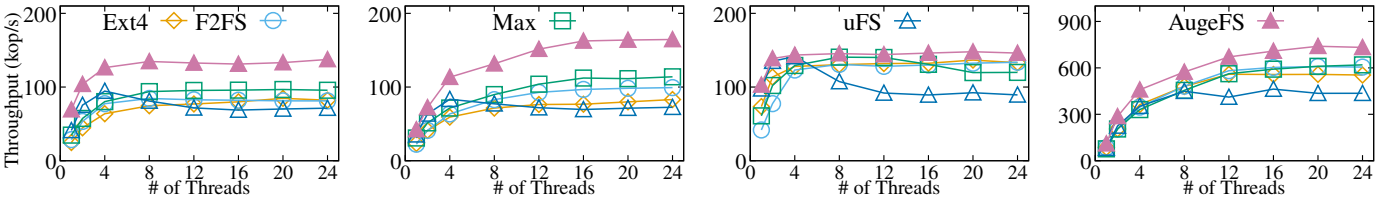
(c) Multi-Thread Write Throughput (d) Multi-Thread Write Latency

(e) Multi-Thread Read Throughput (f) Multi-Thread Read Latency

Fig. 10: Performance of Data Operations

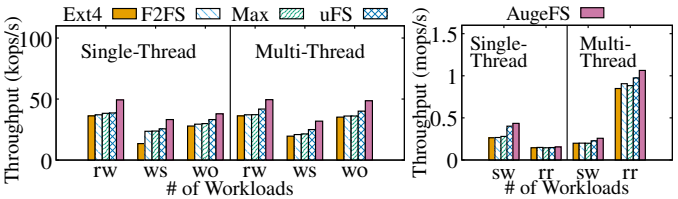
lowest latency for *mkdir*, which increases little even in high thread count. However, other systems have an increasing latency, up to 21×. For *readdir*, Ext4, F2FS, and Max have the lowest latency, and AUGEFS has a stable but higher latency.

Data Performance. We design three microbenchmarks to measure the read/write throughput and latency, and each thread performs on a private 4GB file. We add another compared system AUGEFS-syn, which is the same as AUGEFS but without the asynchronous IO stack. Strata is not compared due to fairness because it stores data in a large-capacity PM while AUGEFS only uses small-size PMR as the WAL log for metadata. First, single-thread write results are shown in Fig. 10(a) and 10(b). During the test, each thread issues append writes followed by *fsync* while varying the write size. AUGEFS exhibits 40% (than AUGEFS-syn)∼2.7× (than Ext4) higher throughput and its latency decreases by 27% (than AUGEFS-syn)∼69% (than Ext4) on average against other systems. Due to its userspace architecture, AUGEFS eliminates kernel software overhead and accesses SSDs in userspace. Additionally, AUGEFS adopts an asynchronous IO stack for *fsync* operations, which parallels the direct node blocks and data blocks writing and brings 40% higher throughput than AUGEFS-syn. As the write size increases, the benefits become less because more IOs are concentrated on data blocks. uFS performs better than kernel file systems because it adopts the userspace high-performance SPDK [94]. Second, multiple-thread write results are shown in Fig. 10(c) and 10(d), where multiple threads issue append write followed by *fsync* with 4KB block size. AUGEFS exhibits 20% (than AUGEFS-syn)∼2.8× (than Ext4) higher throughput and its latency decreases by 14% (than AUGEFS-syn)∼72% (than Ext4) on average against other systems on average. AUGEFS can quickly utilize the bandwidth of SSDs and the device IO becomes the major bottleneck when meeting high thread counts. Ext4 has the worst performance because it only uses a separate thread to dispatch the journal blocks for consistency. uFS has a scalable throughput in low thread count but meets the bottleneck after 12 threads due to its limited number of server threads. Max has a lower throughput than AUGEFS because it suffers from kernel software overhead, competitive submitting locks, and the serialization between the node blocks and data blocks. Third, multi-thread read results are shown in Fig. 10(e) and 10(f), where multiple threads read their private files randomly with 4KB block size. AUGEFS achieves the best performance since the data is served directly from userspace, while most of the time is spent on device IO. Through Ext4, F2FS and Max have a closer performance than AUGEFS because they can almost fully exploit the device bandwidth. uFS scales worse for the reasons below. First, its client cache has little improvement for non-skew random access patterns and we disable it for fairness. Second, the limited server threads and



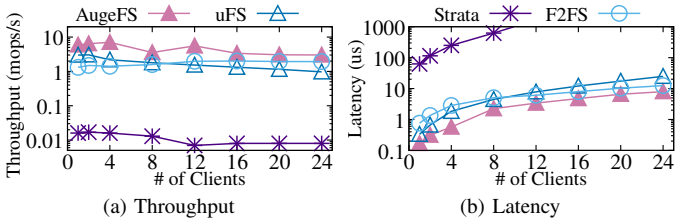
(a) Fileserver (b) Varmail (c) Webserver (d) Webproxy

Fig. 11: Performance of Filebench



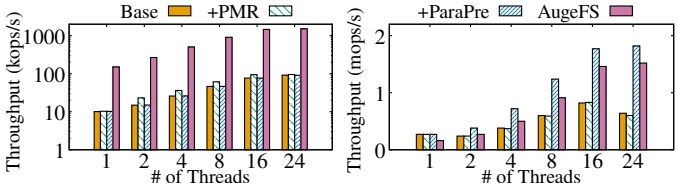
(a) Throughput (b) Throughput

Fig. 12: Throughput of Different Workloads on LevelDB



(a) Throughput (b) Latency

Fig. 13: Cross-Process Sharing Workloads



(a) Mkdir With Fsync (b) Mkdir Without Fsync

Fig. 14: AUGEFS Metadata Performance Breakdown

high communication overhead become the bottleneck.

B. Macrobenchmark: Filebench

We use four workloads (Fileserver, Varmail, Webserver, and Webproxy) from the Filebench to evaluate AUGEFS. Fig. 11 shows the results. Generally, AUGEFS performs the best under all four workloads. AUGEFS outperforms other file systems by 60% to 1× for Fileserver, 50% to 92% for Varmail, 15% to 37% for Webserver, and 24% to 46% for Webproxy. Specially, for write-intensive and fsync-intensive workloads (Fileserver and Varmail), AUGEFS has more evident performance improvement because of its optimizations for metadata and data scalability. For read-intensive workloads (Webserver and Webproxy), all scale well, except uFS which only performs better in low thread count. AUGEFS has a higher read performance due to all-userspace operations, which avoids system calls and kernel IO stack overhead.

C. Real-World Application: LevelDB

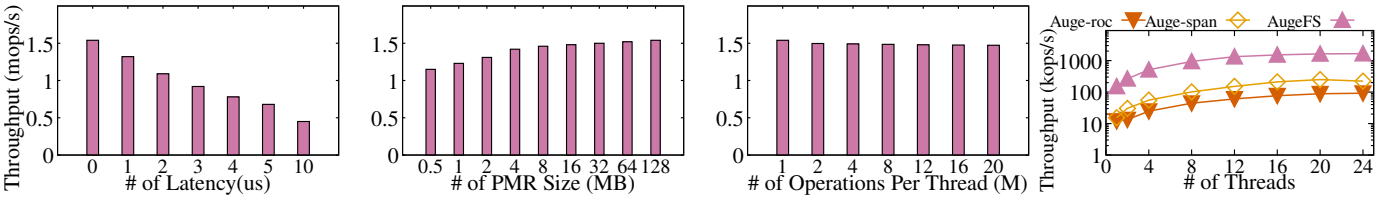
Here we evaluate the performance of AUGEFS using the real-world application LevelDB [22], which is widely used in cloud environments. We run LevelDB's db_bench on different file systems and issue 1 million operations with a value size of 1 KB for each thread. We choose five workloads: random write (rw), write sync (ws), overwrite (wo), sequential write (sw), and random read (rr). We run them using one thread (Single-Thread) and 16 threads (Multi-Thread), and the results are shown in Fig. 12. AUGEFS achieves the highest throughput across all file systems across all workloads. In particular, for random write, AUGEFS outperforms 36%, 33%, 30%, and 23% than Ext4, F2FS, Max, and uFS respectively.

D. Multiple-Process Workloads

We show the ability of AUGEFS to support multi-process sharing by letting multiple processes execute *setxattr* to the same file, which requires acquiring and releasing the inode lock. *setxattr* is a control plane operation by setting an extended attribute value for a file. We compare AUGEFS against three file systems with different architectures to evaluate their control plane overhead: Strata uses socket-based IPC for the control plane (*digest*), uFS uses the shared ring buffer, and F2FS uses *syscall* to execute control plane operations. Fig. 13(a) and 13(b) show the throughput and latency results. For Strata, its LibFS has to acquire and release the lock by communicating with KernFS. Its control plane performs worst as it requires two expensive IPCs and has to write twice for metadata. As for uFS, its server processes delegate all the operations without requiring lock, but it performs worse because only one worker thread can handle metadata operations, which becomes the bottleneck. Though F2FS can enter the shared kernel address space to acquire and release locks, it suffers from expensive kernel trapping overhead. AUGEFS has the highest throughput and lowest latency due to its shared address space in userspace.

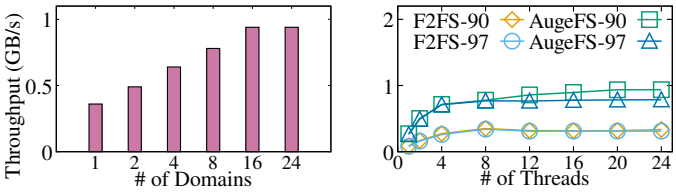
E. Metadata Performance Breakdown

To evaluate the source of AUGEFS performance gains in metadata performance, we show the throughput breakdown in Fig. 14. Base represents using origin RocksDB. +PMR represents only using PMR to store WAL files. +ParaPre represents only using parallel request processing while using multiple WAL files without PMR on SSDs. AUGEFS represents using both parallel request processing and fine-grained



(a) Mkdir Throughput (b) Mkdir Throughput (c) Mkdir Throughput (d) Mkdir Throughput

Fig. 15: AUGEFS Sensitivity Analysis of Metadata Management



(a) Write Throughput (b) Write Throughput

Fig. 16: AUGEFS Sensitivity Analysis of Data Management

parallel WAL. We select two workloads: *mkdir* with/without *fsync*, where *mkdir* with *fsync* occurs in a strong consistency scenario [56] by issuing *fsync* after each metadata operation. For *mkdir* with *fsync* (Fig. 14(a)), only using PMR to store WAL files brings little benefit as it suffers from the grouping overhead severely. Meanwhile, serial WAL write is also the bottleneck. Only using ParaPre also has little benefit as it requires flushing all per-core WAL files. AUGEFS performs best because it eliminates grouping overhead and can update WAL persistently in parallel. For *mkdir* without *fsync* (Fig. 14(b)), AUGEFS performs worse than +ParaPre as AUGEFS persists the WAL due to the persistence feature of PMR, while +ParaPre only writes WAL files in the page cache initially. Even without persistence, Base has a much poorer performance.

F. Sensitivity Analysis

Latency of PMR: Fig. 15(a) shows how the latency of PMR affects the metadata performance. Note that our previous experiments use Optane PM + latency (900ns) to emulate PMR. Here we add more extra latency (1-10us) for per-64Bytes and measure the *mkdir* with *fsync* throughput with 24 threads. As the added latency increases from 1us to 5us, the throughput decreases by 14%, 29%, 40%, 49%, and 54%. When the added latency is 10us, the throughput decreases by 70%. However, it still outperforms other file systems by at least 3.9× (AUGEFS-roc) and up to 31× (Max).

Size of PMR: Fig. 15(b) illustrates how the size of PMR affects the metadata performance with the same workload in §VII-A with 24 threads. We vary the PMR data area size from 512KB to 128MB. When only using 512KB PMR, the throughput decreases by about 25.3%, however, it still outperforms other file systems by at least 11×. As its size increases, more time for migrating WAL data to SSD is hidden, resulting in improved metadata throughput. When the size reaches 128MB, AUGEFS reaches the highest throughput (similar to 256MB case that stores all WAL on PMR) because it nearly does not require waiting for migrating WAL data to SSD.

Size of Metadata Workload: Fig. 15(c) shows how the size of the metadata workload (aging the LSM-KVS) affects the metadata performance. Here, we add more operations to each thread (from 1M to 20M) and expand the metadata area on SSD, and the dataset size is about from 4.6GB to 92GB. Even for the largest dataset we used, the metadata performance only drops less than 5%, where compactions occur frequently. We also increase the number of compaction threads (2 by default) in MetaDB and find it brings little performance improvement, which indicates compaction is not the bottleneck here.

Different Kinds of LSM-KVS: We compare METADB with one state-of-the-art LSM-KVS SpanDB [11], which uses SPDK to accelerate the WAL writes and data IOs. We measure the *mkdir* with *fsync* throughput with up to 24 threads. Fig. 15(d) shows the results, where Auge-roc and Auge-span only use RocksDB and SpanDB to replace METADB in AUGEFS. Though Auge-span shows higher throughput (up to 3×) than Auge-roc, it underperforms AUGEFS for below reasons. First, while SpanDB assigns WAL writes to the specialized logger threads, it still retains the group logging mechanism, suffering from thread synchronization overhead. Second, SpanDB offers asynchronous interfaces to increase the request depth that helps improve throughput. However, the file system metadata operations are executed synchronously, thus gaining limited benefit from the asynchronous request processing.

Number of Domains: Fig. 16(a) shows how the number of domains affects the write performance in AUGEFS with the same workload in §VII-A with 24 threads. As the number of domains increases, its throughput scales well until 16 because the locking contention is becoming less and less. The performance bottleneck has shifted from the FS to the device after 16, i.e., the device bandwidth is saturated fully. Note that AUGEFS currently focuses on building a FS on one storage device like common local file systems [4], [10], [52].

High Volume Utilization: Fig. 16(b) illustrates how the volume utilization affects the LFS performance. We populate the tested file systems (F2FS and AUGEFS) with two levels of volume utilization, 90% and 97%, and then execute a random-write workload on them to incur frequent garbage collections. Each thread performs 4KB random overwrites to its own file, followed by an independent fsync operation. At 90% utilization, AUGEFS outperforms F2FS, benefiting from its domain-based file organization that enhances the scalability of data operations. However, at 97% utilization, AUGEFS experiences

a performance drop due to insufficient free space to support all domains. In this scenario, AUGEFS can also outperform F2FS due to its better scalability.

G. Recovery Time

Different from F2FS, AUGEFS deploys per-domain recovery, which requires recovering for each domain. However, the sizes of each checkpoint for roll-back recovery and the roll-forward data are smaller as each domain is responsible for its own data. After the execution of the multiple-thread append workloads (each thread writes 1GB data with 24 threads), as in Fig. 10(c), we inject an immediate crash and measure the recovery time when remounting the file systems. The time of F2FS and AUGEFS are 2.6s and 393ms, respectively. The recovery procedure of F2FS is executed serially, requiring a longer time. AUGEFS could execute the recovery procedure in parallel because each domain is independent. Compared with F2FS, the recovery of AUGEFS has to compare the versions on node blocks and per-CPU core due to the asynchronous IO stack, however, it brings little overhead because the comparison only costs several CPU cycles.

VIII. RELATED WORK

Userspace File Systems. Many userspace file systems for NVMs (including SSD and persistent memory) are proposed, such as Moneta-D [7], Aerie [87], Arrakis [76], Strata [42], DevFS [37], ZoFS [17], SplitFS [33], EvFS [95], uFS [59], Simurgh [66], ArckFS [99]. EvFS [95] is a library FS lacking multi-process support. The trio architecture of ArckFS is specifically designed for persistent memory. It only maps the data pages of one file into the application's address space for isolation and relies on the page table to restrict PM accesses for protection, which is not applicable to SSDs. uFS, ZoFS, and Simurgh are the most related architecture to AUGEFS. uFS brings additional CPU overhead, and its shared-memory-based IPC is less efficient than AUGEFS. ZoFS and Simurgh map PM into the address space of each application and do not duplicate data or metadata by caching it in DRAM [66]. Simurgh provides high security for PM with the hardware modification and cannot be directly applied to SSDs. Both AUGEFS and ZoFS use MPK to prevent stray writes, however, their file system architectures are different. ZoFS only maps the PM and requires the leases or locks on the slow PM for concurrency between processes. AUGEFS is more likely to the kernel address space and multiple processes share AUGEFS like sharing kernel address space. Thus, we can implement the concurrency control, page cache, and background threads in AUGEFS directly.

SSD File System Metadata Optimization. ReconFS [63], Otter [60], and BlzFS [78] aim to improve the efficiency of metadata operations. Though ReconFS and BlzFS can mitigate the metadata write amplification by recording metadata in a temporary area, they still face the amplification when applying metadata to block devices (i.e., SSDs). TableFS [34] first proposes building a file system on the key-value store, and it simply uses LevelDB and does not consider the performance issues. KVFS [85] is based on a transactional variation of LSM-KVS, called VT-Tree, and it still uses the group updating mechanism. BεtrFS [89] is an in-kernel file system to take advantage of B^{ε}-tree. However, it has heavy software overhead because it is a stacked file system built on Ext4. KEVIN [40] improves file system performance by offloading indexing capability to the storage hardware based on an LSM-tree based KVSSD. However, the computer power in SSD is weak.

LSM-KVS WAL Optimization. SpanDB [11] enables multiple concurrent WAL writes. However, it still keeps the group updating mechanism. p^2KVS [65] partitions the global KV space into a set of independent subspaces. However, it suffers from grouping overhead in the single subspace and requires flushing all WALs for metadata persistence, which brings write amplification of WAL. SineKV [48] leverages the CMB feature of SSDs to store WAL for crash consistency, but it ignores the grouping overhead.

SSD File System Data Optimization. AIOS [44] proposes an asynchronous IO stack to accelerate read and write operations. SpanFS [35] distributes files and directories to partitions for scalability. However, they are for the journaling FS, and SpanFS has to solve the metadata consistency across partitions. Max [52] is the state-of-the-art kernel LFS but suffers from poor metadata performance and submitting lock contention. IPLFS [39] reduces the garbage collection of F2FS, and exF2FS [71] enhances transaction support of F2FS. They are orthogonal to AUGEFS and can be applied to AUGEFS. [14] gives an analysis for the performance degradation of *fsync* in F2FS. There are also works building LFS on ZNS SSDs [23], [43], [77], and our techniques can also be applied to ZNS SSDs. [43] leverages the write pointer of the ZNS SSD to design the order-preserving recovery mechanism.

IX. CONCLUSION

This paper presents AUGEFS, a high-performance and scalable userspace log-structured file system targeting modern SSDs. Building a userspace LFS on modern SSDs faces three challenges: inefficient control plane, limited metadata scalability, and underutilized device bandwidth. Correspondingly, AUGEFS proposes three techniques to address these challenges: protected and shared address space for file systems, a scalable LSM-tree based metadata store, and domain-based file organization with an asynchronous IO stack for *fsync* operation. We compare AUGEFS against six file systems. The evaluation results show the high efficiency of AUGEFS to achieve both metadata and data scalability.

ACKNOWLEDGMENT

We thank our shepherd, Yu Hua, and the anonymous reviewers for their constructive comments and insightful suggestions. This work is supported by the Strategic Priority Research Program of the Chinese Academy of Sciences under grant No. XDB44030200, the Major Research Plan of the National Natural Science Foundation of China (Grant No. 92270202).

REFERENCES

[1] Ahmed Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-mei Hwu. Flatflash: Exploiting the Byte-Accessibility of Ssds Within a Unified Memory-Storage Hierarchy. In *Proc. of ACM ASPLOS*, 2019.

[2] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R. Ganger, and George Amvrosiadis. File Systems Unfit as Distributed Storage Backends: Lessons from 10 Years of Ceph Evolution. In *Proc. of ACM SOSP*, 2019.

[3] ARM. Developer guide: Arm memory domains, 2001. http://infocenter.arm.com/help/.

[4] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The New Ext4 Filesystem: Current Status and Future Plans. In *Proc. of the Linux Symposium*, 2021.

[5] Duck-Ho Bae, Insoon Jo, Youra Adel Choi, Joo-Young Hwang, Sangyeun Cho, Dong-Gi Lee, and Jaeheon Jeong. 2B-SSD: The Case for Dual, Byte-and Block-Addressable Solid-State Drives. In *Proc. of ACM/IEEE ISCA*, 2018.

[6] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *Proc. of USENIX ATC*, 2019.

[7] Adrian M Caulfield, Todor I Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. Providing Safe, User Space Access to Fast, Solid State Disks. In *Proc. of ACM ASPLOS*, 2012.

[8] Chan, Helen HW and Li, Yongkun and Lee, Patrick PC and Xu, Yinlong. Hashkv: Enabling Efficient Updates in KV Storage via Hashing. In *Proc. of USENIX ATC*, 2018.

[9] Yun-Sheng Chang and Ren-Shuo Liu. OPTR : Order-Preserving Translation and Recovery Design for SSDs with a Standard Block Device Interface. In *Proc. of USENIX ATC*, 2019.

[10] Changman Lee, Dongho Sim, Joo Young Hwang, and Sangyeun Cho. F2FS: A New File System for Flash Storage. In *Proc. of USENIX FAST*, 2015.

[11] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage. In *Proc. of USENIX FAST*, 2021.

[12] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Optimistic Crash Consistency. In *Proc. of ACM SOSP*, 2013.

[13] Vijay Chidambaram, Tushar Sharma, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Consistency without ordering. In *Proc. of USENIX FAST*, 2012.

[14] Gyeongyeol Choi and Youjip Won. Analysis for the Performance Degradation of fsync () in F2FS. In *Proc. of ACM IC4E*, 2018.

[15] Jonathan Corbet. Memory protection keys, 2015. https://lwn.net/Articles/643797/.

[16] Daejun Park and Dongkun Shin. iJournaling: Fine-Grained Journaling for Improving the Latency of Fsync System Call. In *Proc. of USENIX ATC*, 2017.

[17] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and Protection in the ZoFS User-Space NVM File System. In *Proc. of ACM SOSP*, 2019.

[18] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience. In *Proc. of USENIX FAST*, 2021.

[19] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System Software for Persistent Memory. In *Proc. of ACM EuroSys*, 2014.

[20] Facebook. RocksDB. http://rocksdb.org/.

[21] Filebench. Filebench 1.4.9.1. https://github.com/filebench/filebench.

[22] Google. LevelDB. https://github.com/google/leveldb.

[23] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Jooyoung Hwang. ZNS+: Advanced Zoned Namespace Interface for Supporting In-Storage Zone Compaction. In *Proc. of USENIX OSDI*, 2021.

[24] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. In *Proc. of ACM SOSP*, 2011.

[25] Mohammad Hedayati and Spyridoula Gravani. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *Proc. of USENIX ATC*, 2019.

[26] IBM. Power isatm version 3.0 b, 2017.

[27] Intel. Intel and Micron Produce breakthrough Memory Technology. https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/.

[28] Intel. Intel Optane SSD P4800X Specification. https://www.intel.com/content/www/us/en/products/sku/129969/intel-optane-ssd-dc-d4800x-series-375gb-2-5in-pcie-2x2-3d-xpoint/specifications.html.

[29] Intel. Intel-SSD-DC-P4510-Review. https://www.storagereview.com/review/intel-ssd-dc-p4510-review.

[30] Intel. Intel® Optane™ SSD P5800X Series. https://www.intel.com/content/www/us/en/products/docs/memory-storage/solid-state-drives/data-center-ssds/optane-ssd-p5800x-p5801x-brief.html.

[31] 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.

[32] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. Improving SSD Lifetime with Byte-Addressable Metadata. In *Proc. of ACM MEMSYS*, 2017.

[33] 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 *Proc. of ACM SOSP*, 2019.

[34] Kai Ren, Garth Gibson. TABLEFS: Enhancing Metadata Efficiency in the Local File System. In *Proc. of USENIX ATC*, 2013.

[35] Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma, and Jinpeng Huai. SpanFS: A Scalable File System on Fast Storage Devices. In *Proc. of USENIX ATC*, 2015.

[36] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Yang-Suk Kee, and Moonwook Oh. Durable Write Cache in Flash Memory SSD for Relational and NoSQL Databases. In *Proc. of ACM SIGMOD*, 2014.

[37] 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 *Proc. of USENIX FAST*, 2018.

[38] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. NVMeDirect: A User-space I/O Framework for Application-specific Optimization on NVMe SSDs. In *Proc. of USENIX HotStorage*, 2016.

[39] Juwon Kim, Minsu Kim, Muhammad Danish Tehseen, Joontaek Oh, and YouJip Won. IPLFS:Log-Structured File System without Garbage Collection. In *Proc. of USENIX ATC*, 2022.

[40] Jinhyung Koo, Junsu Im, Jooyoung Song, Juhyung Park, Eunji Lee, Bryan S Kim, and Sungjin Lee. Modernizing File System through In-Storage Indexing. In *Proc. of USENIX OSDI*, 2021.

[41] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A Cross Media File System. https://github.com/ut-osa/strata.

[42] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A Cross Media File System. In *Proc. of ACM SOSP*, 2017.

[43] Euidong Lee, Ikjoon Son, and Jin-Soo Kim. An Efficient Order-Preserving Recovery for F2FS with ZNS SSD. In *Proc. of ACM HotStorage*, 2023.

[44] Gyusun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W Lee, and Jinkyu Jeong. Asynchronous I/O Stack: A Low-latency Kernel I/O Stack for Ultra-Low Latency SSDs. In *Proc. of USENIX ATC*, 2019.

[45] Sangjin Lee, Alberto Lerner, André Ryser, Kibin Park, Chanyoung Jeon, Jinsub Park, Yong Ho Song, and Philippe Cudré-Mauroux. X-SSD: A Storage System with Native Support for Database Logging and Replication. In *Proc. of ACM SIGMOD*, 2022.

[46] Andrew W Leung, Shankar Pasupathy, Garth Goodson, and Ethan L Miller. Measurement and Analysis of Large-Scale Network File System Workloads. In *Proc. of USENIX ATC*, 2008.

[47] Cheng Li, Philip Shilane, Fred Douglis, Hyong Shim, Stephen Smaldone, and Grant Wallace. Nitro: A Capacity-Optimized SSD Cache for Primary Storage. In *Proc. of USENIX ATC*, 2014.

[48] Fei Li, Youyou Lu, Zhe Yang, and Jiwu Shu. SineKV: Decoupled Secondary Indexing for LSM-based Key-Value Stores. In *Proc. of IEEE ICDCS*, 2020.

[49] Ruibin Li, Xiang Ren, Xu Zhao, Siwei He, Michael Stumm, and Ding Yuan. ctFS: Replacing File Indexing with Hardware Memory Translation through Contiguous File Allocation for Persistent Memory. In *Proc. of USENIX FAST*, 2022.

[50] Xiaojian Liao. Max: A Multicore-Accelerated File System for Flash Storage. https://github.com/thustorage/max.

[51] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. Write dependency disentanglement with HORAE. In *Proc. of USENIX OSDI*, 2020.

[52] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. Max: A Multicore-Accelerated File System for Flash Storage. In *Proc. of USENIX ATC*, 2021.

[53] Xiaojian Liao, Youyou Lu, Zhe Yang, and Jiwu Shu. Crash Consistent Non-Volatile Memory Express. In *Proc. of ACM SOSP*, 2021.

[54] Xiaojian Liao, Zhe Yang, and Jiwu Shu. RIO: Order-Preserving and CPU-Efficient Remote Storage Access. In *Proc. of ACM EuroSys*, 2023.

[55] Seung-Ho Lim, Hyun Jin Choi, and Kyu Ho Park. Journal Remap-Based FTL for Journaling File System with Flash Memory. In *Proc. of Springer HPCC*, 2007.

[56] Zhen Lin, Lingfeng Xiang, Jia Rao, and Hui Lu. P2CACHE: Exploring Tiered Memory for In-Kernel File Systems Caching. In *Proc. of USENIX ATC*, 2023.

[57] Linux. Memory Managament. https://www.kernel.org/doc/html/v5.8/x86/x86_64/mm.html.

[58] Jing Liu, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Sudarsun Kannan. File Systems as Processes. In *Proc. of USENIX HotStorage*, 2019.

[59] Jing Liu, Anthony Rebello, Yifan Dai, Chenhao Ye, Sudarsun Kannan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Scale and Performance in a Filesystem Semi-Microkernel. In *Proc. of ACM SOSP*, 2021.

[60] Yubo Liu, Hongbo Li, Yutong Lu, Zhiguang Chen, and Ming Zhao. An Efficient and Flexible Metadata Management Layer for Local File Systems. In *Proc. of IEEE ICCD*, 2019.

[61] Hongjun Lu, Jeffrey Xu Yu, Ling Feng, and Zhixian Li. Fully Dynamic Partitioning: Handling Data Skew in Parallel Data Cube Computation. *Distributed and Parallel Databases*, 2003.

[62] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Wisckey: Separating Keys from Values in SSD-conscious Storage. In *Proc. of USENIX FAST*, 2017.

[63] Youyou Lu, Jiwu Shu, and Wei Wang. ReconFS: A Reconstructable File System on Flash Storage. In *Proc. of USENIX FAST*, 2014.

[64] Youyou Lu, Jiwu Shu, and Jiacheng Zhang. Mitigating Synchronous I/O Overhead in File Systems on Open-Channel SSDs. *ACM Transactions on Storage (TOS)*, 2019.

[65] Ziyi Lu, Qiang Cao, Hong Jiang, Shucheng Wang, and Yuanyuan Dong. p2KVS: a Portable 2-dimensional Parallelizing Framework to Improve Scalability of Key-Value Stores on SSDs. In *Proc. of ACM EuroSys*, 2022.

[66] Nafiseh Moti, Frederic Schimmelpfennig, Reza Salkhordeh, David Klopp, Toni Cortes, Ulrich Rückert, and André Brinkmann. Simurgh: A Fully Decentralized and Secure NVMM User Space File System. In *Proc. of ACM SC*, 2021.

[67] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W Moore. Understanding PCIe Performance for End Host Networking. In *Proc. of ACM SIGCOMM*, 2018.

[68] Edmund B Nightingale, Kaushik Veeraraghavan, Peter M Chen, and Jason Flinn. Rethink the Sync. In *Proc. of USENIX OSDI*, 2006.

[69] NVMe. NVM Express. https://nvmexpress.org/.

[70] NVMe. NVMe 1.4 Spec Revision 1.4c. https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4c-2021.06.28-Ratified.pdf.

[71] Oh, Joontaek and Ji, Sion and Kim, Yongjin and Won, Youjip. exf2fs: Transaction support in log-structured filesystem. In *Proc. of USENIX FAST*, 2022.

[72] OHSHIMA, S. Scaling Flash Technology to Meet Application Demands. *In Keynote 3 at Flash Memory Summit*, 2018.

[73] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *Proc. of USENIX ATC*, 2019.

[74] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. Skew-aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems. In *Proc. of ACM SIGMOD*, 2012.

[75] PCIe. PCI Express Base Specification Revision 3.0. https://picture.iczhiku.com/resource/eetop/wHiSRjtztkeJLnVc.pdf.

[76] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. In *Proc. of USENIX OSDI*, 2014.

[77] Devashish R Purandare, Sam Schmidt, and Ethan L Miller. Persimmon: An Append-only ZNS-first Filesystem. In *Proc. of IEEE ICCD*, 2023.

[78] Wenjie Qi, Zhipeng Tan, Ziyue Zhang, Jing Zhang, Chao Yu, Ying Yuan, and Shikai Tan. BlzFS: Crash Consistent Log-structured File System Based on Byte-loggable Zone for ZNS SSD. In *Proc. of IEEE ICCD*, 2023.

[79] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *Proc. of ACM SOSP*, 2017.

[80] Yujie Ren, Changwoo Min, and Sudarsun Kannan. CrossFS: A Cross-Layered Direct-Access File System. In *Proc. of USENIX OSDI*, 2020.

[81] Drew Roselli, Jacob R Lorch, and Thomas E Anderson. A Comparison of File System Workloads. In *Proc. of USENIX ATC*, 2000.

[82] Mendel Rosenblum and John K Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems (TOCS)*, 1992.

[83] SAMSUNG. Samsung SZ1735a Z-SSD. https://download.semiconductor.samsung.com/resources/brochure/Samsung%20SZ1735a%20U.2%20Z-SSD.pdf.

[84] SAMSUNG. Ultra-Low Latency with Samsung Z-NAND SSD. https://www.samsung.com/us/labs/pdfs/collateral/Samsung_Z-NAND_Technology_Brief_v5.pdf.

[85] Pradeep Shetty, Richard Spillane, Ravikant Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. Building Workload-Independent Storage with VT-Trees. In *Proc. of USENIX FAST*, 2013.

[86] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, Efficient In-process Isolation with Protection Keys. In *Proc. of USENIX Security*, 2019.

[87] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M Swift. Aerie: Flexible File-System Interfaces to Storage-Class Memory. In *Proc. of ACM EuroSys*, 2014.

[88] Wenhao Lv, Youyou Lu, Yiming Zhang, Peile Duan, and Jiwu Shu. InfiniFS: An Efficient Metadata Service for Large-Scale Distributed Filesystems. In *Proc. of USENIX FAST*, 2022.

[89] 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 *Proc. of USENIX FAST*, 2015.

[90] Youjip Won, Jaemin Jung, Gyeongyeol Choi, Joontaek Oh, Seongbae Son, Jooyoung Hwang, and Sangyeun Cho. Barrier-Enabled IO Stack for Flash Storage. In *Proc. of USENIX FAST*, 2018.

[91] Hobin Woo, Daegyu Han, Seungjoon Ha, Sam H Noh, and Beomseok Nam. On Stacking a Persistent Memory File System on Legacy File Systems. In *Proc. of USENIX FAST*, 2023.

[92] Kan Wu, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Towards an Unwritten Contract of Intel Optane SSD. In *Proc. of USENIX HotStorage*, 2019.

[93] Zhe Yang, Youyou Lu, Erci Xu, and Jiwu Shu. CoinPurse: A Device-Assisted File System with Dual Interfaces. In *Proc. of ACM/IEEE DAC*, 2020.

[94] Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E Paul. SPDK: A Development Kit to Build High Performance Storage Applications. In *Proc. of IEEE CloudCom*, 2017.

[95] Takeshi Yoshimura, Tatsuhiro Chiba, and Hiroshi Horii. EvFS: User-level,Event-Driven File System for Non-Volatile Memory. In *Proc. of USENIX HotStorage*, 2019.

[96] Qiang Zhang, Yongkun Li, Patrick PC Lee, Yinlong Xu, Qiu Cui, and Liu Tang. UniKV: Toward High-Performance and Scalable KV Storage in Mixed Workloads via Unified Indexing. In *Proc. of IEEE ICDE*, 2020.

[97] Shawn Zhong, Chenhao Ye, Guanzhou Hu, Suyan Qu, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, and Michael Swift. MadFS: Per-File Virtualization for Userspace Persistent Memory Filesystems. In *Proc. of USENIX FAST*, 2023.

[98] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, et al. XRP: In-Kernel Storage Functions with EBPF. In *Proc. of USENIX OSDI*, 2022.

[99] Diyu Zhou, Vojtech Aschenbrenner, Tao Lyu, Jian Zhang, Sudarsun Kannan, and Sanidhya Kashyap. Enabling High-Performance and Secure Userspace NVM File Systems with the Trio Architecture. In *Proc. of ACM SOSP*, 2023.