RomeFS: A CXL-SSD Aware File System Exploiting Synergy of Memory-Block Dual Paths

Yekang Zhan
HUST
zhanyekang@hust.edu.cn

Haichuan Hu
HUST
hhc_81579@hust.edu.cn

Xiangrui Yang
HUST
yxr620@hust.edu.cn

Shaohua Wang
HUST
m202273497@hust.edu.cn

Qiang Cao*
HUST
caoqiang@hust.edu.cn

Hong Jiang
UT Arlington
hong.jiang@uta.edu

Jie Yao
HUST
jackyao@hust.edu.cn

ABSTRACT

Compute eXpress Link (CXL) based Solid-State Drives (CXL-SSDs), such as the Samsung CMM-H model, promise to offer CXL.mem memory and CXL.io block dual-mode interfaces. Nonetheless, whether and how cloud applications with diverse and varying access patterns benefit from such dual-mode CXL-SSD remains an open question for academia and industry.

This paper proposes RomeFS, the first CXL-SSD aware file system, handling file operations by synergistically yet preferentially utilize complementary CXL.mem and CXL.io data paths. To this end, RomeFS presents key enabling techniques including 1) dual-path data layout to statically partition metadata and file data into CXL.mem and CXL.io data-zones respectively; 2) dual-path access for file data using the two data paths synergistically at runtime; 3) hybrid parallel file indexing for efficient per-file mapping to locate dispersed file data across the two data paths; 4) data defragmentation to merge dispersed file data to the CXL.io data-zone; and 5) metadata journaling and synergistic dual-path transactional write to ensure crash consistency with low overhead. We implement and evaluate RomeFS under two emulated hardware platforms. The experiments show that RomeFS outperforms state-of-the-art block-based file systems and PM-based file systems by up to 14.24× and 4.89× respectively.

CCS CONCEPTS

• **Computer systems organization** → *Heterogeneous (hybrid) systems.*

KEYWORDS

CXL-SSD, File system, Storage architecture, IO scheduling

ACM Reference Format:
Yekang Zhan, Haichuan Hu, Xiangrui Yang, Shaohua Wang, Qiang Cao, Hong Jiang, and Jie Yao. 2024. RomeFS: A CXL-SSD Aware File System Exploiting Synergy of Memory-Block Dual Paths. In *ACM Symposium on Cloud Computing (SoCC '24), November 20–22, 2024, Redmond, WA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3698038.3698539

1 INTRODUCTION

Compute eXpress Link (CXL) [43] via PCI-express (PCIe) interconnects processors, accelerators, memory, and IO devices, driving memory scalability, disaggregation, and pooling in cloud datacenters [25]. Emerging CXL-based Solid-State Drives (CXL-SSDs) [15, 41, 55], which integrate the fore-end build-in DRAM and the backend large-capacity flash, utilize the CXL.mem interface to offer a scalable and cost-effective memory expansion akin to a persistent memory but attached to PCIe bus instead of traditional memory bus [7, 11, 15, 20, 39, 55].

However, for the memory expansion, when the DRAM/flash capacity ratio is low, completely hiding the long latency of the backend flash (e.g., 2TB) is challenging, due to the huge latency gap and granularity mismatch between DRAM and flash, and agnostic application-behaviors [55]. The CXL-SSDs such as Samsung CMM-H [7, 39] promise dual-mode interfaces of CXL.mem and CXL.io and internally partition

*Corresponding author.

SoCC '24, November 20–22, 2024, Redmond, WA, USA

ACM ISBN 979-8-4007-1286-9/24/11.
https://doi.org/10.1145/3698038.3698539

the backend large-capacity flash into the memory area and SSD area to store the data from such two interfaces respectively. This means that the CXL-SSD can support the memory and block-based data paths simultaneously. Nevertheless, so far, CXL-SSD only exists the form of a conceptual protocol without consensual specifications and implementation details. More importantly, whether and how such dual-mode CXL-SSD can improve the performance of diverse applications in cloud remains an unexplored question for academia and industry.

To better understand the performance characteristics of the dual-mode CXL-SSD, we analyze the bandwidth and latency performance of its two data paths. Table 1 lists the bandwidths of CXL memory (similar to accessing the build-in DRAM of the CXL-SSD via the CXL.mem path), representing the ideal performance of the memory expansion mode, and PCIe SSDs, representing the performance of the SSD mode. The CXL memory does not possess an overwhelming bandwidth advantage over the PCIe SSDs. Meanwhile, existing studies demonstrate that the access latency of different CXL memory implementations [25, 28, 48] ranges from 140ns to 300ns, which is significantly lower than that of the SSDs (e.g., several μs to tens of μs for 4KB block [40]).

These observations provide *two key insights* : 1) large-size data accesses can be efficiently handled by the CXL.io data path to directly exploit SSD's high internal parallelism, avoiding passing through the small-size yet expensive build-in DRAM cache; 2) small-size data accesses hit in the cache can be fast processed by the CXL.mem data path. Therefore, compared to either the memory expansion mode only via the CXL.mem path or the SSD mode only via the CXL.io path, synergistically using the two data paths of dual-mode is expected to achieve the best of both worlds. Unfortunately, existing storage systems cannot utilize the memory and block-based data paths simultaneously in a synergistic and complementary manner.

This paper proposes RomeFS, the first CXL-SSD aware file system, to proactively exploit the unique dual-path access feature of the CXL-SSD to efficiently serve cloud applications with high intensity and varying access patterns. RomeFS proactively converts each file request to one or multiple data sub-requests, which are executed via their own preferred data paths in parallel. Specifically, RomeFS designs a dual-path data layout to provide static and coarse-grained data-zone allocation, storing all metadata in the CXL.mem data-zone and storing file data cross the CXL.mem and CXL.io data-zones. RomeFS further allocates large block-aligned file sub-writes to the CXL.io data path while writing small or the residual block-unaligned file sub-writes to the CXL.mem data path, thus synergistically utilizing the two data paths to efficiently handle various file data requests at runtime.

RomeFS with such a synergistic dual-path architecture is fundamentally different from traditional file systems specialized to a single type of data path, such as the block-based data path (block-based file systems [19, 22, 31, 34, 38, 49]) and the memory-semantic data path (PM-based file systems [2, 5, 16, 17, 53, 58, 60, 61]). Additionally, existing cross-media hybrid file systems [21, 37, 52, 57] employ caching or tiering architectures to use memory-semantic devices to accelerate the underlying block IOs, like the memory expansion mode of CXL-SSDs does, mainly focusing on the memory-semantic data path. Therefore, existing file systems are not able to exploit the CXL.mem and CXL.io paths of CXL-SSD synergistically and efficiently.

RomeFS overcomes two key challenges introduced by the synergistic dual-path architecture: 1) file fragmentation and 2) crash consistency. First, the synergistic dual-path access tends to distribute file data across the memory area and SSD area of CXL-SSD, causing file fragmentation and potentially impacting read performance. To address this challenge, RomeFS introduces a hybrid parallel file indexing scheme to efficiently locate the file data distributed across the two areas. RomeFS further designs merge-on-write (MOW) and per-block data log write-back (LWB) mechanisms to mitigate file data fragmentation and timely release the memory area of CXL-SSD. Second, a user write can induce multiple read and write IOs upon both data paths, potentially increasing the complexity of write transaction and crash consistency. Therefore, RomeFS uses a metadata journaling and synergistic dual-path transactional write mechanism to ensure write atomicity and crash consistency with low overhead.

We implement and evaluate RomeFS on two emulated hardware flatforms (i.e., DRAM + SSD and PM + SSD) of the CXL-SSD under a variety of benchmarks and workloads. The experiments show that RomeFS outperforms the block-based file systems (EXT4 [31], CJFS [34], BTRFS [38] and F2FS [22]) by up to 14.24× and 9.44× in latency and throughput respectively, and outperforms the PM-based file systems (NOVA [53] and WineFS [17]) by up to 4.89× and 4.23× in latency and throughput respectively.

In summary, this paper makes the following contributions:

- The first CXL-SSD aware file system that adaptively and jointly utilizes the CXL.mem and CXL.io data paths of CXL-SSD to handle various file requests in a synergistic and complementary manner.
- Several novel enabling techniques such as dual-path data layout, synergistic dual-path access, hybrid parallel file indexing, data defragmentation and dual-path transactional write, to overcome the challenges arising from synergistic dual-path operations.
- Implementing and evaluating RomeFS on two emulated hardware platforms with a variety of workloads.

The remainder of this paper is organized as follows. Section 2 presents necessary background on CXL protocols and CXL-SSD model details, as well as our motivation. Section 3 describes the challenges arising from the synergistic dual-path architecture and presents the design and implementation of RomeFS. Section 4 analyzes the rationality and feasibility of our emulation methodology for CXL-SSDs, and evaluates the performance of RomeFS. We discuss related works in Section 5, and conclude in Section 6.

2 BACKGROUND AND MOTIVATION

2.1 Compute eXpress Link (CXL)

Compute eXpress Link (CXL) over PCIe is an emerging interconnect protocol integrating CPUs, accelerators, memory, and IO devices into a unified resource pooling with low synchronization overheads and high scalability [15, 25, 28, 48]. The CXL standard defines three separate protocols of CXL.io, CXL.cache, and CXL.mem. The CXL.io protocol uses protocol features of the PCIe to provide standard block interface and is also compatible with the legacy NVMe interface. The CXL.cache and CXL.mem protocols are used for the device to access the CPU's memory and for the CPU to access the device's memory, respectively [4].

CXL supports three types of devices (Type-1/2/3). CXL Type-1 devices, such as an accelerator without any attached memory, implement the CXL.io and CXL.cache protocols. CXL Type-2 devices, such as accelerators with attached memory, support all three protocols [43]. CXL Type-3 devices implement the CXL.io and CXL.mem protocols, and are expected to be used for memory expansion. CXL memory [48], CXL-PM [12] and CXL-SSD [7, 15, 39, 55] are primary CXL Type-3 devices. This work focuses on CXL-SSD.

2.2 CXL-SSD model

Research on CXL-SSD study is an open and active area [7, 11, 15, 20, 39, 55]. A CXL-SSD generally comprises the fore-end build-in DRAM and the backend large-capacity flash. The capacity of the build-in DRAM is generally limited due to constraints on budgets of cost, power consumption, and physical size, but the flash can be easily scaled. The memory expansion mode of CXL-SSDs offers load/store interface via the CXL.mem path, which first accesses the build-in DRAM cache and then accesses the flash if the cache misses. Nevertheless, low capacity-ratio, huge latency gap, and granularity mismatch between DRAM and flash limit the performance of the memory expansion mode of CXL-SSDs. The memory expansion mode has to resort to application-aware caching and prefetching mechanisms to hide long internal flash-access latencies [15, 20, 55].

Samsung proposes a CMM-H (CXL Memory Module - Hybrid) CXL-SSD model, to provide the CXL.io and CXL.mem

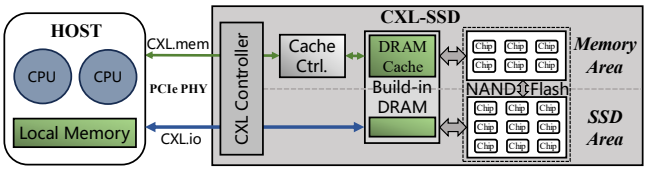


Figure 1: CXL-SSD dual-mode architecture.

interfaces simultaneously, called "dual-mode" [42]. A CMM-H prototype comprises 16GB build-in DRAM and 2TB flash but without the detailed implementation and specification. Combining existing studies and public information, we summarize a general dual-mode architecture of the CXL-SSD in Figure 1. Both the CXL.io and CXL.mem data paths externally share the same underlying PCIe channel. A large portion of the build-in DRAM and (configurable) a portion of the flash are used as the memory area, while the remaining DRAM and flash are used as the SSD area. The memory area and SSD area are exclusively accessed by the CXL.mem and CXL.io interfaces respectively. The two areas also share the SSD-internal flash IO channel between the DRAM and flash. Due to high bandwidth of modern PCIe and high internal parallelism within the flash, the CXL.mem and CXL.io paths are relatively independent in performance.

Compared to block storage, such as high-performance SSDs, and slow memory device, such as persistent memory (PM), dual-mode CXL-SSDs possess the following new features: 1) dual-mode support for load/store memory interface and block IO interface simultaneously; 2) small granularity access, allowing for data transfers of a minimum size of 64B; and 3) crash-persistence guarantee, e.g., supporting the flush-on fail with CXL2.0 GPF (Global Persistent Flush) feature and providing external battery for the build-in DRAM cache [7, 39].

For the CXL.mem path, CPU can map the CXL-SSD's memory area into its own address space (by updating page tables) for memory expansion, and load/store data in the memory area in a cache-coherent fashion. For the CXL.io path, the CXL-SSD supports the legacy block IO stack composed of VFS, page cache, file system, block layer and device driver [23]. In addition, for simplicity yet universality for the dual-mode architecture, we logically assume that the memory area and SSD area are separately mapped to distinct flash zones. Within the CXL-SSD, both areas can be dynamically and transparently mapped to different physical flash zones to ensure wear leveling.

To better understand the performance characteristics of the CXL-SSD, we analyze the bandwidth and latency of its behavior-similar CXL memories and PCIe SSDs. Table 1 lists the bandwidth of CXL memory (similar to accessing the build-in DRAM of the CXL-SSD via the CXL.mem path), representing the ideal performance of the memory area, and NVMe SSDs, representing the performance of the SSD area.

Table 1: Maximum effective bandwidth of CXL memory and NVMe SSDs.

	CXL memory (reported by [48])			**NVMe SSD** [40]	
Bandwidth (GB/s)	**CXL-A (DDR5)**	**CXL-B (DDR4)**	**CXL-C (DDR4)**	**PCIe5.0 SSDs**	**PCIe4.0 SSDs**
Read	17.7	9.0	4.9	10–14	5–7
Write	11.9	3.6	4.4	6–12	3–6

The CXL-A/B/C are three true CXL memory devices reported by [48]. The CXL memory does not possess an overwhelming bandwidth advantage over the SSDs with large capacity and sufficient internal parallelism. However, the access latency of CXL memory, 140ns-300ns reported by recent studies [25, 28, 48], is significantly lower than that of the SSDs (e.g., several μs to tens of μs for 4KB block [40]). These observations suggest that *the two data paths are complementary in that, while the build-in DRAM cache via the CXL.mem data path favors small-size data accesses, the CXL.io data path is friendly to large-size data accesses to effectively exploit flash's IO parallelism*. Therefore, synergistically using the two data paths is expected to achieve the best of both worlds.

Cloud applications exhibit various, mixed, and fluctuating load patterns, such as small and large, random and sequential, block-aligned and block-unaligned, single IO stream and multiple IO streams [29, 33, 51, 63]. The questions of whether and how these applications can benefit from such two complementary data paths provided by the CXL-SSD remain unanswered.

2.3 Motivation

File systems offer a common file/directory view via POSIX interface for upper applications to access data in the underlying storage. This implies that file systems can identify preferred data paths for different data access behaviors, thereby exploiting the two data paths of CXL-SSDs.

However, most file systems are specialized to a single type of data path, such as block-based data path (block-based file systems [19, 22, 31, 34, 38]) and memory-semantic data path (PM-based file systems [17, 53, 61]). Some cross-media hybrid file systems [21, 37, 52, 57] employ caching or tiering architectures with persistent memory devices to accelerate the underlying block IOs, like the memory expansion mode of CXL-SSDs does. Therefore, these file systems are unable to exploit the CXL.mem and CXL.io paths synergistically. We are motivated to design a CXL-SSD aware file system supporting synergistic dual-path exploitation.

3 DESIGN AND IMPLEMENTATION

We propose the first CXL-SSD aware file system, RomeFS, to fully exploit the synergy between the two heterogeneous data paths of CXL-SSD by designing a novel dual-path architecture. Inspired by the *key insight* of "the CXL.io path is friendly to large-size accesses and the CXL.mem path favors small-size accesses" (§ 2.2), the design principle of RomeFS is to adaptively utilize the CXL.mem data path with low latency but limited capacity to handle latency-sensitive small-size tasks while leveraging the CXL.io data path to handle bandwidth-sensitive large-size tasks to fully exploit the internal parallelism of SSDs.

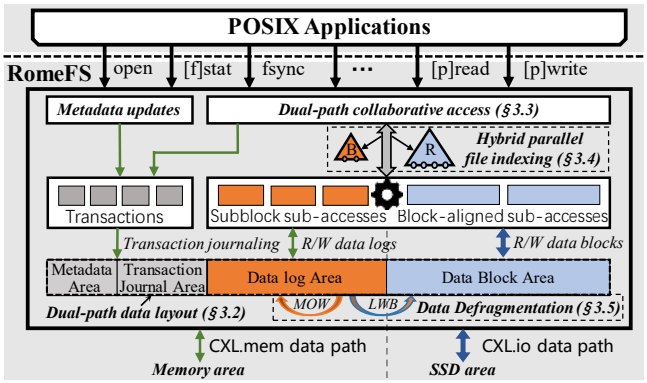


Figure 2: RomeFS overview.

RomeFS faces three key challenges (**C**s) of 1) dual-path selection and synergization (**C1**), 2) data fragmentation (**C2**), and 3) crash consistency (**C3**). For **C1**, RomeFS needs to decide how to use either or both of the two data paths for file operations according to the design principle. For **C2**, dual-path RomeFS synergizing tends to distribute file data across the memory area and SSD area of CXL-SSDs, thereby causing file fragmentation and potentially impacting read performance. For **C3**, a user write can induce multiple read and write IOs upon both data paths, potentially increasing the complexity of write transaction and crash consistency.

3.1 Overview

Figure 2 shows the architecture of RomeFS designed to address these three challenges. For dual-path selection and synergization (**C1**), RomeFS designs a dual-path data layout (§ 3.2) to explicitly divide all metadata into the memory area and file data into the data log area of memory area and the data block area of SSD area respectively. This means all metadata operations are handled via the CXL.mem data path and file data are placed in data logs in the memory area and data blocks in the SSD area. RomeFS further proposes a dual-path access mechanism (§ 3.3) to determine how the two data paths synergistically handle various file data accesses at runtime. RomeFS only writes block-aligned file data to the SSD area to exploit the parallelism of the flash while writing block-unaligned file data into the memory area to avoid its induced read-modify-write operations of the CXL.io path [1, 10, 45]. To overcome file fragmentation (**C2**), RomeFS introduces a hybrid parallel file indexing scheme (§ 3.4) to effectively index the data distributed across the two areas.

RomeFS further designs merge-on-write (MOW) and per-block data log write-back (LWB) mechanisms (§ 3.5) to mitigate file data fragmentation and timely release the memory area of CXL-SSD. To ensure crash consistency (**C3**), RomeFS uses metadata journaling (§ 3.2) and synergistic dual-path transactional write mechanism (§ 3.3) to ensure write atomicity with low overhead.

3.2 Dual-path Data Layout

RomeFS first designs a unified dual-path data layout to manage the memory area and SSD area of CXL-SSD, which consists of *memory zone* (i.e., the address space of the memory area) and *block zone* (i.e., the address space of the SSD area).

The *memory zone* consists of small-size pages (e.g., 4KB) and is further divided into metadata area, transaction journal area and data log area. The metadata area includes superblocks, bitmaps, inode tables, file mapping structures, etc. The transaction journal area is used to record metadata updates and file operations. The data log area is designed to absorb small and subblock file writes. The *block zone* consists of large-size data blocks (e.g., 64KB) that exclusively constitute the data block area. Most file data are stored in the data block area, but a few file data can be temporarily placed in the data log area conditionally. This data layout also improves metadata locality, thereby improving the performance of metadata access via the CXL.mem path. This is because benefiting from the CXL.mem protocol, data fetched from and written to the metadata area can be cached in the cache hierarchy in the CPU [28]. In addition, the *memory zone* and *block zone* are exclusively accessed by the CXL.mem path and the CXL.io path respectively, thereby eliminating potential data-incoherency between the two data paths.

Metadata journaling. A metadata update can induce multiple write-IOs upon the CXL.mem path, hence RomeFS processes metadata updates using a metadata journaling approach (like XFS [49]) to provide transactional guarantees for metadata operations while persisting the journal immediately. Specifically, RomeFS records each metadata update as a transaction in a journal and writes the journal via the CXL.mem path to the transaction journal area. Then, RomeFS updates the corresponding metadata structure in host memory only. At checkpointing, RomeFS finally applies these updates to the metadata area via the CXL.mem path based on the recorded journals. The internal caching mechanism [55] and crash-persistence guarantees (e.g., CXL2.0 GPF feature and external battery for the build-in DRAM cache) of CXL-SSD can ensure crash consistency between the build-in DRAM cache and backend flash of the CXL.mem path.

The crash-persistence guarantee of CXL-SSDs brings about a great leap forward for the consistency mode of file systems. Traditional file systems without the CXL.mem path suffer from slow per-journal persistence and IO amplifications due to performing numerous slow sync-and-ordered small-writes. Therefore, traditional file systems often do not actively persist journals but only do so in scenarios such as journal timeouts, journal buffer full, file system unmounting, and applications demand (e.g., *fsync()* call), thus sacrificing some consistency for performance. In comparison, CXL-SSD aware file systems can fast store each journal to the memory area with persistence, thus providing stronger consistency.

3.3 Synergistic Dual-path Access

RomeFS proposes a synergistic dual-path write mechanism to adaptively allocate tasks to the CXL.mem and CXL.io paths at runtime, i.e., keeping the CXL.io path for always handling (large) block-aligned writes without extra slow read-modify-writes, and utilizing the two paths in parallel. Moreover, to ensure write atomicity, the write mechanism is designed to be transactional.

Write request splitting. When receiving a write request, RomeFS converts it into 1) block-aligned sub-writes and 2) residual subblock sub-writes based on a write splitting threshold. The (large) block-aligned sub-writes can be efficiently handled via the CXL.io path directly without extra read-modify-writes. The residual subblock sub-writes are handled via the CXL.mem path with low latency. In this way, RomeFS can efficiently process both types of sub-writes in parallel to fast handle requests while fully considering the limited capacity of the build-in DRAM. In addition, this cost-effective use of the CXL.mem path makes it easier for the build-in DRAM cache to hide the access latency of the backend flash, thereby potentially improving the performance of the CXL.mem path.

Write splitting threshold. The write splitting threshold is a key parameter for determining both write ranges and workloads upon the two data paths. This threshold selection represents a trade-off in usage between the CXL.mem data path and the CXL.io data path. When the threshold is too low, most of the file writes are allocated into the *block zone* via the CXL.io data path, resulting in high latency of small writes. When the threshold is too high, most of the file writes are judged as subblock writes and processed by the CXL.mem data path, resulting in frequent out-of-space and flushing to flash of the build-in DRAM cache. Therefore, RomeFS needs to set the threshold carefully based on the actual effective performance of the CXL.mem and CXL.io data paths. RomeFS supports flexible configuration of the threshold at initialization. In this paper, RomeFS sets the threshold to 64KB by default because the write latencies of the CXL.mem and CXL.io data paths in our emulated devices are closest when the IO size is 64KB.

Synergistic dual-path transactional write. After write request splitting, a user write can induce multiple sub-writes

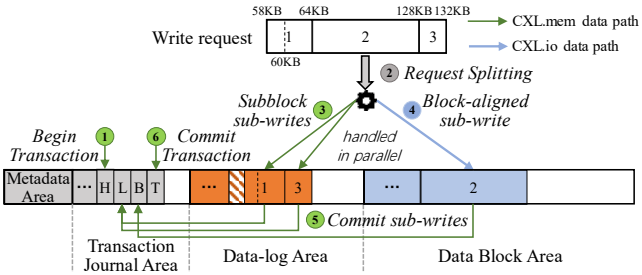


Figure 3: An example of synergistic dual-path write.

upon the CXL.mem and CXL.io data paths. To provide write atomicity guarantees, RomeFS designs a synergistic dual-path transactional write mechanism. This write mechanism first converts user writes into multiple sub-writes by the write request splitting mechanism, then handles these sub-writes transactionally and in parallel.

Figure 3 shows an example of the synergistic dual-path transactional write. RomeFS processes the write request in six stages. ① RomeFS first creates a write transaction in a new journal. The journal only records metadata updates and transaction information. Specifically, RomeFS records transaction header, corresponding metadata updates and sub-write properties (e.g., type, merge or not) to this journal. The write request is logically split immediately for the purpose of convenient recording of the journal. ② RomeFS creates and updates the corresponding write request structures in the host memory based on the request splitting result, and sends them to the corresponding data paths. RomeFS handles ③ subblock sub-writes and ④ block-aligned sub-write in parallel. Among them, subblock sub-writes are executed serially by default. The low-latency CXL.mem path can quickly complete these small writes. Enabling parallel processing of subblock sub-writes introduces additional overhead and resource consumption, so it is optional. Note that, for simplicity and clarity, merge-on-write (MOW) (§ 3.5.1) is not introduced in this example. In reality, RomeFS uses the MOW approach to handle each subblock sub-write atomically. The block-aligned sub-write can be stored directly and atomically in data blocks without the need for copy-on-write [38]. ⑤ For each completed sub-write, RomeFS commits a sub-write transaction via the CXL.mem path. ⑥ After completing all sub-writes, RomeFS atomically updates this journal tail to commit the write transaction and updates its metadata structure in host memory accordingly.

The "H/L/B/T" in Figure 3 means transaction header (H), log (L) / block (B) sub-write transaction, and transaction tail (T), respectively. For the dual-path commit communication, RomeFS pre-creates several dedicated IO threads for the CXL.io path. Each IO thread takes responsibility the dedicated address space range in an interleaved manner. After entering kernel and then completing the write splitting, a user thread sends the split block-aligned sub-writes to the corresponding IO threads based on their addresses, then continues to execute the tasks of the CXL.mem path. After the tasks of the CXL.mem path are completed, the user thread waits for the IO threads to finish by polling a semaphore, and finally performs the corresponding commits. Multiple user threads are handled in a similar manner.

In this way, the transactional overheads are borne by the low-latency CXL.mem data path while the CXL.io data path effectively handles block-aligned sub-writes without additional transactional overhead.

When RomeFS commits a journal, a portion of the data is placed in data logs and is not immediately written back to improve performance. Although this increases file fragmentation, it is a worthwhile option for CXL-SSDs. Because the low-latency CXL.mem data path can ensure the efficiency of reading data logs. Our evaluation also proves this (e.g., Figure 8). RomeFS further uses the per-block data log write-back mechanism (LWB) (§ 3.5.2) to complete the write-back of merged data logs, thus reducing the double write overhead of journaling. Moreover, most file data is directly written to data blocks, thus eliminating a large amount of file-write data writeback. In addition, this transactional write mechanism is also used to ensure atomicity of LWB.

In comparison, traditional journaling file systems [31, 34] complete the write-back of all file-write data on commit, thus incurring expensive double writes and high transactional overhead. Overall, the synergistic dual-path transactional write mechanism enables RomeFS to fast persist user write requests of arbitrary patterns atomically (e.g., various request sizes, request offsets and load intensities) and enhances the load adaptability of RomeFS.

Synergistic dual-path read. RomeFS passively selects data paths based on the location of the requested data to handle read requests in parallel. Specifically, RomeFS uses the hybrid parallel file indexing scheme (§ 3.4) to efficiently look up the data logs and data blocks relevant to read requests and process reads.

3.4 Hybrid Parallel File Indexing

Write splitting and data logging of RomeFS distribute file data across data logs and data blocks at different granularities. Therefore, RomeFS needs a uniform and efficient indexing mechanism to fast locate requested data in the data logs, the data blocks, or both in parallel. To this end, RomeFS designs a hybrid parallel file indexing scheme for per-file mapping, combining a B+tree [3] indexing the data logs with random and sparse addresses and a radix tree [24] indexing the address-contiguous data blocks, respectively.

File mapping. The data log always keeps the latest data for a file if it exists. When locating file data, a naive scheme of reading data after locating data for the two data areas

Algorithm 1 Early termination mechanism for parallel file indexing

```
Require: File logical offset and size of a read request: offset, size.
 1: procedure EARLY_TERMINATION_MECHANISM(offset, size)
 2:     DS ← 0
 3:     Call get_B+tree(offset, size) and get_radixtree(offset, size) in parallel
 4:     while true do
 5:         if the radix tree completes the data locating && DS == 0 then
 6:             Wait for get_B+tree(offset, size)
 7:         if the B+tree locates all the required data then
 8:             DS ← 1
 9:             Stop get_radixtree(offset, size)
10:             Read the data logs according to get_B+tree(offset, size)
11:         else if the B+tree completes the locating of data then
12:             DS ← 2
13:             #Execute the following statements in parallel:
14:             Read the data logs according to get_B+tree(offset, size)
15:             Read the residual requested data according to the locating of
                both trees via the CXL.io data path
16:         if DS == 1 or DS == 2 then
17:             break
```

may introduce a long and unnecessary waiting delay. For example, the requested data is entirely located in the data log area but requires waiting for old-data indexing in the data block area, or the requested data is entirely located in the data block area but requires waiting for null-indexing in the data log area. Therefore, we design an early termination mechanism to adaptively terminate unnecessary waiting in a data log first manner. As shown in Algorithm 1, the dedicated semaphore (DS) is first reset to 0 when receiving a request (line 2), indicating that the locating of the B+tree has not yet completed, making the radix tree wait if it completes first (line 5-6). If the B+tree locates all the required data for a request, the DS is set to 1 to indicate that there is no need to read data blocks located by the radix tree (line 7-10). Otherwise, the DS is set to 2 to indicate that the requested data is distributed across both data logs and data blocks (line 12-15). The located data logs can be read directly without waiting for the radix tree after the B+tree completes the locating (line 11 and 14), while the determination of which data blocks to read is based on the B+tree's locating results, thereby avoiding reading outdated data (line 15).

When updating index structures, RomeFS uses the metadata journaling (§ 3.2) to record updates to a journal, and modifies the B+tree and radix tree in host memory according to their respective methods. The B+tree and radix tree are kept in host memory to speedup file operations. At checkpointing, RomeFS applies these updates to the metadata area based on the recorded journal. Besides, we employ legacy range locks [56, 61] on both trees to ensure data consistency of dual-path accesses and concurrent accesses.

Optimized B+tree. Reading the latest data of a data block may require locating multiple data logs. To ❶ limit the number of data logs corresponding to a data block and ❷ reduce the indexing overheads for per-block data log write-back (LWB) (§ 3.5.2), which requires locating all data logs corresponding to a data block, we modified the B+tree leaf nodes to ensure that each leaf node is dedicated to indexing data logs belonging to a single data block. In this way, setting the number of index entries for the leaf node can achieve ❶, and each LWB only needs to look up a single leaf node thus achieving ❷. The modified leaf node contains 16 index entries by default. Each index entry contains an 8B logical address of a data log. Moreover, after each LWB, RomeFS does not reclaim the empty leaf node but instead reclaims all empty leaf nodes in a batch at checkpointing thus avoiding frequent B+tree reconstruction. Such delayed reclaiming is a trade-off between performance and memory-usage, which is typically worthwhile because writing data logs (and updating corresponding leaf nodes) is more frequent than LWB.

In addition, MOW (§ 3.5.1) requires the file logical offset and size of merged data logs. To effectively obtain the information required by MOW, RomeFS records them to the index entry of the B+tree leaf node. In this way, MOW can obtain the information directly after accessing the B+tree, which is a necessary process for file writes, thereby avoiding additional accesses for MOW.

3.5 Data Defragmentation

When the small-size updates are increasingly placed in the data logs, RomeFS suffers from the file data fragmentation problem, which introduces increased random reads, and more complex file mapping and *memory zone* space cleaning. Therefore, RomeFS proposes a merge-on-write (MOW) approach for data logs to mitigate data fragmentation in *memory zone*, which ensures atomic data log write by out-of-place write. Moreover, to release the *memory zone* space timely and eventually eliminate file data fragmentation between the two zones, RomeFS proposes a per-block data log write-back mechanism (LWB) to write (random) data logs back to (large and continuous) data blocks.

3.5.1 Merge-on-write (MOW). The key idea of MOW is to merge new data log writes with existing address-overlapping data logs on write. Benefiting from the optimized B+tree (§ 3.4), MOW can obtain the data log properties for merging without additional accesses.

Algorithm 2 describes the MOW process. RomeFS first looks up the B+tree to locate all data logs involved in the data log write block by block (line 2). RomeFS attempts to sequentially merge those address-overlapping data logs into a new data log write, but not immediately modify the data (line 3-7). Next, RomeFS scans the corresponding leaf node of B+tree to determine if there are enough free index entries for the merged data log write. If yes, the merge is successful, and RomeFS persists this merged write into empty data log pages (out-of-place write for atomicity) and releases the outdated

Algorithm 2 Merge-on-write

Require: offset and size of the new data log write: newoffset, newsize
1: **procedure** *MOW*(newoffset, newsize)
2: indexentries[][] ← Search B+tree to get locations of relevant logs
3: **for** each leaf_indexentries[] in indexentries[][] **do**
4: **for** each (logoffset,logsize) in leaf_indexentries[] **do**
5: **if** the data log overlaps with the new data log write **then**
6: Merge the data log to the new data log write
7: Update newoffset and newsize, and invalid the data log
8: **if** the available index entries are sufficient **then**
9: Update the leaf node and perform the merged data log write
10: **else** Write all data logs back to corresponding data block
11: Update newoffset and newsize for the next logical block merge

data log pages. Otherwise, the new data log write and those data logs are forced to be written back to the corresponding data block (line 8-10). Note that, to simplify the merge, a data log is only stored in a single page, so a merged data log write may be placed in multiple pages. RomeFS continues to merge the data logs of the next data block until all relevant data logs are processed (line 3 and 11). Figure 4 (left) illustrates a data block example of MOW. This data log write's 6KB-8KB logical address area is merged with the first data log, its 8KB-12KB area is logged as a new data log, and its 12KB-14KB area is merged with the second data log. In addition, if a data log write is an append to a data log, this write will be directly appended to the end of this log (if the page containing this log has sufficient space), thereby avoiding the need to rewrite the entire merged data again.

3.5.2 Per-block data log write-back (LWB). For a file that is being written to, a naive coarse-grained (e.g., per-file) write-back mechanism must either rely on complex lock mechanisms for consistency or strictly prohibit writeback for this file. Moreover, a coarse-grained write-back operation can take a considerable amount of time and cause the dual-path write to be blocked during this period when *memory zone* space is insufficient. Therefore, RomeFS uses a fine-grained per-block writeback mechanism to overcome these issues, thereby 1) limiting the update region of each writeback to a single data block and leveraging the simple range locks of hybrid parallel file indexing (§ 3.4) to ensure consistency, and 2) quickly clearing out available *memory zone* space to maintain the dual-path write and allowing RomeFS to perform well even under heavy loads.

LWB is triggered when MOW fails or when available *memory zone* space is insufficient. Figure 4 (right) illustrates an example of LWB triggered by a failed MOW. The data block contains 16 data logs that are totally dispersed. The data log write cannot be merged with existing data logs and all 16 index entries of the B+tree leaf node are occupied, thus LWB is triggered. Specifically, RomeFS first reads the data block via the CXL.io data path and its corresponding data logs via

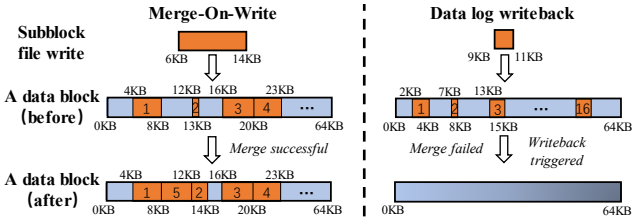


Figure 4: A data block example of merge-on-write (MOW) of data logs and per-block data log write-back (LWB) triggered by a failed MOW.

the CXL.mem data path, then merges and writes them into a newly allocated block (for atomic writing) via the CXL.io data path. The number of index entries of the B+tree leaf node is configurable, and setting more index entries means LWB is triggered less frequently, but reading the latest data of a data block may require reading more data logs. LWB is efficient because it reads (small) data logs via the CXL.mem data path, and reads and writes the entire (large) data block via the CXL.io data path.

Moreover, when the available *memory zone* space is less than a predefined threshold, LWB is triggered by RomeFS directly. By default, RomeFS prioritizes writing the data logs of a data block that occupies the most *memory zone* space back to the data block when the available *memory zone* space is less than 10%. The priority policy for write-back and the predefined threshold are configurable.

3.6 Crash consistency

RomeFS provides both atomic metadata and data operations. RomeFS ensures crash consistency by metadata journaling (§ 3.2) and synergistic dual-path transactional write (§ 3.3). After a crash (e.g., power outage), RomeFS recovers the state in two steps: (1) it rolls back to the latest consistent checkpoint by scanning the metadata in the *memory zone*, and (2) it performs roll-forward recovery by redoing the operations recorded on the transaction journals serially.

Consistency mode. Benefiting from fast persistence and byte access granularity of PM, PM-based file systems persist user writes immediately by default. In comparison, block-based file systems tend to write back *page cache* without persistence, and suffer from slow data persistence, especially metadata updates.

Owing to the crash-persistence guarantees of CXL-SSDs (§ 2.2), CXL-SSD aware file systems can fast persist metadata updates. However, due to the limited capacity of the build-in DRAM cache within CXL-SSDs, it is difficult for CXL-SSD aware file systems to fast persist all data updates via the CXL.mem path like PM-based file systems. Nonetheless, RomeFS can fast persist data updates via the CXL.io path. This is because RomeFS always allocates (large) block-aligned writes to the CXL.io path, which eliminates the IO

Table 2: Experimental setup.

Setup	Devices for emulation	File systems for evaluation
CXL-SSD A	10GB DRAM with additional latency and 3.84TB SAMSUNG PM9A3 SSD	EXT4, CJFS, BTRFS, F2FS, RomeFS
CXL-SSD B	20GB Intel Optane DC PMM and 3.84TB SAMSUNG PM9A3 SSD	NOVA, WineFS, RomeFS-PM

amplification overhead during data persistence, especially for slow read-modify-write operations. Furthermore, considering the increasingly high bandwidth and internal parallelism of SSDs, RomeFS can even be more efficient than PM-based file systems when persisting large writes. Therefore, RomeFS persists all metadata updates (as journals) immediately by default, while making persisting all data updates immediately as an option for *buffered IO* [36].

4 EVALUATION

In this section, we evaluate the performance of RomeFS with two CXL-SSD emulation hardware configurations (DRAM + SSD and PM + SSD) under various load patterns. We analyze the advantages and innovations of RomeFS relative to mainstream block-based and PM-based file systems. Specifically, in § 4.1, we describe our experimental setup and emulation configurations, and discuss the rationality and feasibility of our emulation methodology for CXL-SSDs. In § 4.2, we evaluate the write and read performance of RomeFS under loads with various IO-size ranges and numbers of user threads. Moreover, we also analyze a write performance breakdown of RomeFS. In § 4.3, we evaluate the performance of RomeFS under loads with mixed reads and writes, and mixed metadata and data accesses. In § 4.4, we evaluate the performance of RomeFS under real-application loads with mixed small and large, reads and writes.

4.1 Experimental Setup

Environment. Our evaluation machine has two Intel Xeon Gold 6348 processors (2.60 GHz, 28 CPUs) and 256GB of DDR4 DRAM to run Ubuntu 20.04 with Linux kernel 5.18.18 for CXL-SSD A and 5.1.0 for CXL-SSD B, detailed in Table 2 and Figure 5. For all experiments, we pin threads to the cores, disable CPU frequency scaling, and clear the kernel cache before each run. All experimental results are the average of at least three runs.

Emulation. Currently, we have no access to any commodity CXL-SSD products because none is available on the market to the best of our knowledge. We use separated DRAM/PM and SSD to mimic the performance characteristics of the dual-mode CXL-SSD, as shown in Figure 5. The CXL-SSD A uses a DDR4 DRAM with additional access latency to mimic the performance characteristics of accessing the build-in DRAM cache of the memory area via the CXL.mem path, reflecting the best performance of the CXL.mem path without synchronously writing the flash. We

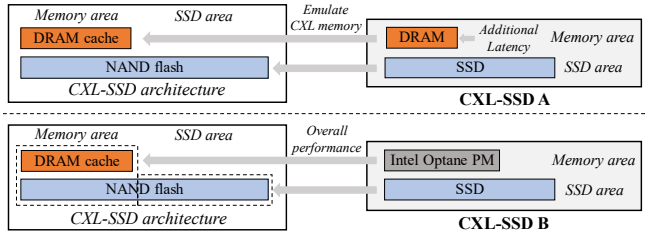


Figure 5: Emulation methodology for the CXL-SSDs.

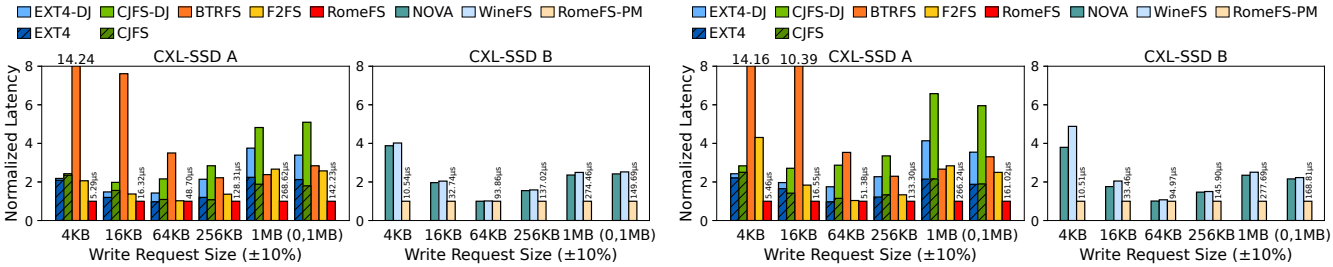
Table 3: The average random access latency of the emulated build-in DRAM of CXL-SSD A.

Latency (μs)	4KB	16KB	64KB	256KB	1MB
Write	2.56	6.83	23.41	81.93	308.40
Read	2.93	5.85	17.25	59.58	227.95

follow the access latency of a real DDR4 CXL memory (reported in [48]) to configure the access latency of the DDR4 DRAM, as shown in Table 3. Note that, although different real CXL memories may not exhibit the same performance due to their own hardware configurations such as FPGA or ASIC [28, 48], we focus on the representative latency and bandwidth of the CXL memory. The CXL-SSD B uses an Intel Optane PM [54] to mimic the performance characteristics of accessing the memory area of the CXL-SSD via the CXL.mem path, reflecting the actual performance of the CXL.mem path with internal (black-box) persistence delay. In both configurations, the backend SSD is the 3.84TB Samsung PM9A3 SSD (PCIe4.0) [13] with the 6.8GB/s read and 4GB/s write bandwidths.

Note that, although both the DRAM of CXL-SSD A and the PM of the CXL-SSD B are not actually connected to a physical PCIe bus, they are set to similar performance characteristics of existing CXL memory and memory area of CXL-SSDs, respectively. An intuitive concern is that there may be potential interference in the external PCIe and internal flash bus between the CXL.io and CXL.mem paths. This can be largely neglected for the following reasons. First, the CXL-SSD employs PCIe5.0 with a bandwidth twice that of PCIe4.0. We set both bandwidths of the two paths not to exceed PCIe4.0 X4 (i.e., 8GB/s). Second, the CXL-SSD can asynchronously flush the cached data into the flash in background. Moreover, the flash has high internal parallelism. These allow the CXL-SSD to flexibly schedule the flash IOs from both the CXL.mem and CXL.io paths to avoid IO contention. Therefore, we argue that our emulation can reasonably reflect the performance characteristics of future real dual-mode CXL-SSDs.

File systems on CXL-SSDs. To demonstrate the performance of file systems running on the CXL-SSDs under different loads, we set the available sizes of the DRAM of CXL-SSD A to 10GB and the PM of CXL-SSD B to 20GB by default, and adjust the load sizes to indirectly represent different memory-area usage of CXL-SSDs.



(a) Block-unaligned random writes. 50GB and 100GB write for CXL-SSD A and CXL-SSD B respectively for the evaluation of memory area capacity : load size = 1 : 5. (***1:5 case***)

(b) Block-unaligned random writes. 100GB and 200GB write for CXL-SSD A and CXL-SSD B respectively for the evaluation of memory area capacity : load size = 1 : 10. (***1:10 case***)

Figure 6: Normalized average write latency of file systems under different request size and the amount of data written. We set the performance of RomeFS(-PM) as the normalized reference, the same applies below.

We compare RomeFS against mainstream block-based file systems: EXT4 [31], CJFS [34], BTRFS [38] and F2FS [22], and PM-based file systems: NOVA [53] and WineFS [17]. EXT4, BTRFS and F2FS are mature and widely used real-world block-based file systems. CJFS is the state-of-the-art block-based journaling file system. EXT4-DJ/CJFS-DJ refers to EXT4/CJFS enabling data journal. Note that, EXT4-DJ, CJFS-DJ, BTRFS, F2FS and RomeFS ensure atomic operations for both metadata and data, but EXT4 and CJFS ensure only atomic metadata operations, which are used to present a more comprehensive comparison. NOVA is a highly acclaimed in-kernel PM-based file system. WineFS is the state-of-the-art in-kernel PM-based file system. Both NOVA and WineFS are configured to ensure both atomic metadata and data operations consistent with RomeFS.

For RomeFS, both CXL-SSDs are configured as dual-mode. RomeFS uses the DRAM/PM as the memory area and uses the SSD as the SSD area. Because the block-based file systems cannot run on the memory device, we compare RomeFS against them on CXL-SSD A, which is transparently used as an SSD using the build-in DRAM as cache via the CXL.io path only. We compare RomeFS against the PM-based file systems on CXL-SSD B, labeling RomeFS on CXL-SSD B by RomeFS-PM for distinction. The PM-based file systems transparently use CXL-SSD B as the memory device via the CXL.mem path only. Note that, the block-based file systems cannot actively keep all metadata in the build-in DRAM of CXL-SSDs because they can use the CXL.io interfaces only. In contrast, the PM-based file systems and RomeFS can explicitly keep the metadata in the memory area.

The questions of how block-based file systems utilize the DRAM of CXL-SSD A and how PM-based file systems run after the PM of CXL-SSD B (i.e., memory area) is exhausted are unanswered. To this end, we employ a classic caching strategy for these baseline file systems to perform the passive data switch between the DRAM/PM and the backend SSD. The incoming data are first written to the DRAM/PM and then are conditionally flushed into the backend SSD. A read missing in the DRAM/PM is replaced from the backend SSD to the DRAM/PM using Least Recently Used (LRU) strategy. When the cache is exhausted, incoming writes are written to the backend SSD directly. Although the cache strategy is relatively simple, it has a similar overall effectiveness to sophisticated cache strategies under moderate or heavy loads. Note that, due to limited 10GB/20GB-sized memory area, the additional overhead of the caching strategy is insignificant compared to IOs.

4.2 Microbenchmarks

We first evaluate the write and read performances of RomeFS. For a fair comparison, under the write-only evaluations (i.e., Figure 6 and Figure 9), we use a single flush thread for each of the baseline file systems to actively flush cached data in the background after the DRAM/PM is exhausted, so as to make fuller use of the DRAM/PM. Note that, after verification, the single-threaded flushing not only does not harm the write performance of the baseline file systems but also makes them perform better.

4.2.1 Single-Threaded Performance. For both write and read evaluations, we set the amount of data of the evaluated loads to 50GB and 100GB for CXL-SSD A and B respectively (i.e., memory area capacity : load size = 1 : 5) to represent a continuous mid-term load, and 100GB and 200GB for CXL-SSD A and B respectively (i.e., memory area capacity : load size = 1 : 10) to represent a continuous long-term load. The file size is consistent with the load size.

Write performance. Initially, the file is placed in the backend SSD (area). We perform six types of block-unaligned random writes with different request sizes, representing various write-load patterns. Specifically, the corresponding request size fluctuates by ±10% but the mean is fixed. The (0, 1MB) case means randomly generating values within this range as the request size and the mean is 512KB. We call *fsync()* to persist writes after each request.

The results are shown in Figure 6. RomeFS is faster (with shorter latency) than block-based file systems: EXT4-DJ,

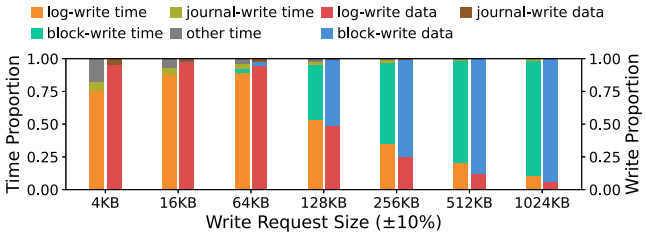


Figure 7: RomeFS write breakdown. The bars on the left and right are the time breakdown and file-write data breakdown, respectively.

CJFS-DJ, BTRFS and F2FS by up to 4.14×, 6.59×, 14.24×, and 4.31×, respectively. RomeFS-PM is faster than PM-based file systems: NOVA and WineFS by up to 3.89× and 4.89×, respectively. The advantages of RomeFS stem from four aspects. 1) RomeFS ensures that writes to the backend SSD are always block-aligned, preventing slow read-modify-writes and potential increased atomicity overheads. In comparison, the baseline file systems have to endure slow read-modify-writes after the DRAM/PM is exhausted. Similarly, an intuitive approach of directly sending large writes to the backend SSD [14, 62] also suffers from slow read-modify-writes, regardless of whether DRAM/PM is available or not. 2) RomeFS always keeps metadata in the DRAM/PM, allowing metadata operations (which often involve small writes and reads) to be quickly completed at the DRAM/PM speed. In comparison, part of the metadata operations of the block-based file systems are slowly handled on the backend SSD. 3) RomeFS adaptively writes converted sub-requests in the two data paths in parallel. In comparison, the baseline file systems can only process the entire request using a single type of data path without the opportunity of splitting requests to match the respective advantages of the two data paths of CXL-SSDs. 4) RomeFS can keep efficient dual-path writes by using the effective LWB and matched MOW (§ 3.5), without degrading to writing to the backend SSD directly. For small-write loads, LWB can efficiently write back merged data logs in aligned large blocks (e.g., 64KB). For large-write loads, most data are efficiently stored to the backend SSD directly, LWB only needs to handle a small amount of data.

RomeFS performs comparably between the *1:5 case* and the *1:10 case*, because RomeFS, which benefits from MOW and LWB (§ 3.5), only requires a small amount of data log area, in addition to accommodating metadata, to run effectively. The small performance difference (about 5%) between RomeFS and RomeFS-PM in the (0, 1MB) case illustrates that under workloads with a large variation in IO size, the performance of RomeFS is less dependent on the performance of the memory area. Such workloads are common on cloud nodes [29, 51]. The significant advantage of RomeFS over the baseline file systems in the (0, 1MB) case further demonstrates RomeFS's load adaptability. Moreover, RomeFS outperforms

Table 4: Sensitivity study of write splitting threshold to dual-path write proportion.

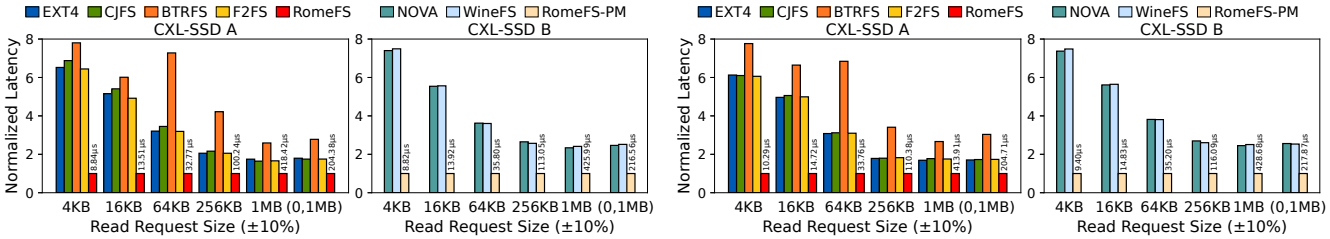
Write size (±10%)	4KB	16KB	64KB	256KB	1MB	(0,1MB)
Threshold	Write proportions of the CXL.mem and CXL.io paths (%)					
4KB	97.0 / 3.0	24.8 / 75.2	6.2 / 93.8	1.6 / 98.4	0.4 / 99.6	0.8 / 99.2
16KB	100.0 / 0.0	97.0 / 3.0	24.8 / 75.2	6.2 / 93.8	1.6 / 98.4	3.1 / 96.9
64KB	100.0 / 0.0	100.0 / 0.0	97.0 / 3.0	24.8 / 75.2	6.2 / 93.8	12.4 / 87.6
256KB	100.0 / 0.0	100.0 / 0.0	100.0 / 0.0	97.0 / 3.0	24.8 / 75.2	50.1 / 49.9
1MB	100.0 / 0.0	100.0 / 0.0	100.0 / 0.0	100.0 / 0.0	97.0 / 3.0	100.0 / 0.0

EXT4 and CJFS in almost all write cases, even though the latter do not ensure the atomicity of data operations. This indirectly demonstrates the efficiency of synergistic dual-path transactional write (§ 3.3).

Overall, the baseline file systems degenerate to writing to the backend SSD directly for most requests under continuous mid-term and long-term loads, thereby suffering from slow read-modify-write operations on the backend SSD. The single-path baseline file systems cannot leverage dual-path synergy to avoid rapid DRAM/PM exhaustion and unaligned writes for the backend SSD. In comparison, RomeFS uses the CXL.mem path to handle subblock sub-writes, ensuring that the CXL.io path always processes large block-aligned sub-writes, thereby maximizing the CXL.io path performance. Through such dual-path synergy, RomeFS is able to quickly process and persist various writes with diverse sizes and offsets, even under long-term loads.

Write performance breakdown. To better understand how RomeFS benefits from its synergistic dual-path write, we perform a breakdown analysis on random writes with sufficient build-in DRAM capacity of CXL-SSD A, as shown in Figure 7. To facilitate a proportional display, we stack the log-write time and block-write time together, although in reality, data logs and data blocks are written in parallel. As the size of write requests increases, the proportion of data handled by the backend SSD becomes larger, such as in the 1024KB case, approximately 94% of the data is directly written to the backend SSD. Moreover, when the write request size is greater than or equal to 128KB, RomeFS uses the CXL.io path to process more data with less time overhead compared to using the CXL.mem path. This demonstrates that the synergistic dual-path write of RomeFS fully utilizes the performance of the CXL.io path by leveraging the CXL.mem path to handle subblock sub-writes.

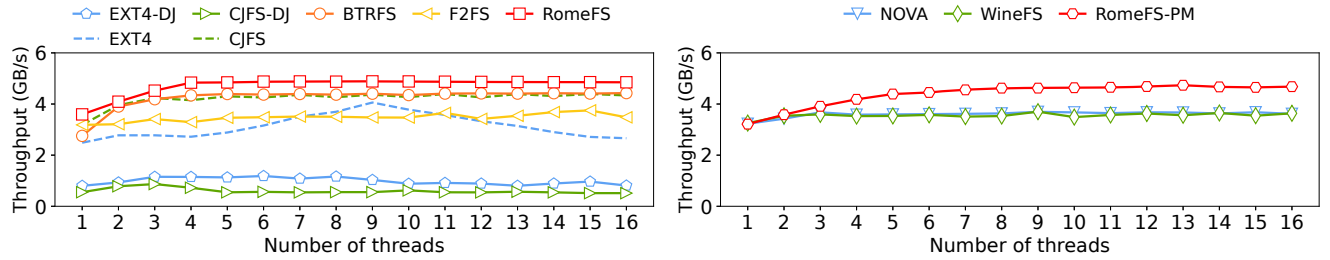
Sensitivity study of write splitting threshold. We further measure the write proportions of the CXL.mem and CXL.io paths for the above random write experiments with sufficient memory area capacity under different thresholds, as shown in Table 4. The larger the threshold size, the more writes are judged as subblock writes and processed by the CXL.mem path. This implies that in order to maximize overall performance, the threshold selection needs to take into account the capacity of the build-in DRAM cache, the performance of both paths, as well as access patterns of applications. Especially, the CXL.mem path suffers from significant



(a) 4KB-aligned random reads with direct IO mode. 50GB and 100GB read for CXL-SSD A and CXL-SSD B respectively for the evaluation of memory area capacity : load size = 1 : 5.

(b) 4KB-aligned random reads with direct IO mode. 100GB and 200GB read for CXL-SSD A and CXL-SSD B respectively for the evaluation of memory area capacity : load size = 1 : 10.

Figure 8: Normalized average read latency of file systems under different request size and the amount of data read.



(a) 100GB random writes on CXL-SSD A.

(b) 200GB random writes on CXL-SSD B.

Figure 9: Multi-threaded write throughput with the request size range of (0, 1MB) and direct IO mode.

performance degradation after the build-in DRAM cache is exhausted. Therefore, RomeFS with a default 64KB threshold tends to allocate most write-data to the CXL.io path for medium to large writes, even if it may result in the CXL.mem path waiting after completing its tasks. Nevertheless, the threshold selection is a complex but worthwhile topic to further explore in future works.

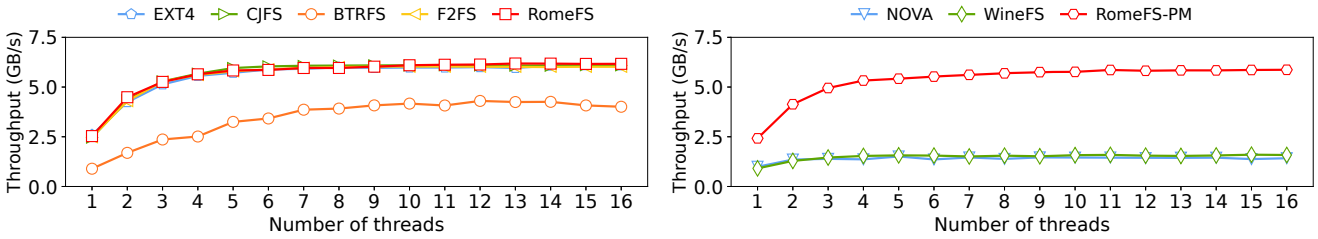
Read performance. Initially, 10GB and 20GB file data are placed in the DRAM and PM respectively, while the remaining file data are stored in the backend SSD. RomeFS performs dual-path writes with the IO size of 512KB ± 10% to generate the file with conditional write-backs to maintain only 10GB/20GB of the DRAM/PM usage. We perform 4KB-aligned (i.e., corresponding request size ±10% and rounded to 4KB-aligned) random reads with direct IO mode to bypass *page cache* for evaluation. We are unable to perform reads with direct IO mode on EXT4-DJ and CJFS-DJ.

The results are shown in Figure 8. RomeFS is faster (shorter latency) than block-based file systems EXT4, CJFS, BTRFS and F2FS by up to 6.52×, 6.89×, 7.80×, and 6.44×, respectively. RomeFS-PM is faster than PM-based file systems NOVA and WineFS by up to 7.39× and 7.49×, respectively. The advantages of RomeFS stem from three aspects. 1) Dual-path RomeFS does not perform LRU replacements of CXL-SSDs but rather reads data from where it is, without the overhead of replacement (i.e., reading missed request data from the backend SSD and then writing it to the DRAM/PM). The baseline file systems only use a single type of data path and cannot avoid LRU replacements. An intuitive concern is that for the read loads with high locality, RomeFS might degenerate to reading only from the backend SSD. Simply enabling *page cache* for RomeFS can efficiently handle such loads. 2) The file data reads in the two data paths of RomeFS are parallel. RomeFS's hybrid parallel file indexing scheme (§ 3.4) further accelerates dual-path reads. 3) MOW (§ 3.5.1) effectively mitigates file fragmentation caused by data logs, thus improving read efficiency. Besides, the overhead of a small amount of file fragmentation is hidden by the low-latency CXL.mem path and dual-path reads in parallel.

4.2.2 Multi-Threaded Performance. For both write and read evaluations, we set both the file size and the total amount of load data to 100GB and 200GB for CXL-SSD A and CXL-SSD B respectively. To evaluate RomeFS's concurrency capability, multiple IO streams are directed to a single file, and we measure the aggregate throughput. We perform 4KB-aligned random writes/reads with the request size range of (0, 1MB) and enable direct IO mode to exclude the interference of *page cache* under multiple IO streams [32, 35].

Concurrent writes. Initially, the file is placed in the backend SSD (area). As shown in Figure 9, RomeFS outperforms block-based EXT4-DJ, CJFS-DJ, BTRFS and F2FS by up to 6.06×, 9.44×, 1.31×, and 1.47×, and RomeFS-PM outperforms PM-based NOVA and WineFS by up to 1.30× and 1.33× in throughput respectively.

BTRFS performs second only to RomeFS because it employs the copy-on-write mechanism that is friendly to 4KB-aligned writes, without additional IO amplification. EXT4-DJ and CJFS-DJ employ the legacy journaling mechanism,



(a) 100GB random reads on CXL-SSD A.

(b) 200GB random reads on CXL-SSD B.

Figure 10: Multi-threaded read throughput with the request size range of (0, 1MB) and direct IO mode.

thus inducing double writes and other costly overheads for crash consistency. Interestingly, even EXT4 and CJFS, which without double writes, are still inferior to RomeFS. This implies that RomeFS's synergistic dual-path transactional write remains efficient under concurrent writes, largely because RomeFS's novel commit strategy, which commits a journal without immediately writing back data logs (§ 3.3).

BTRFS, F2FS, NOVA and WineFS saturate the backend SSD with a small number of threads, so as threads increase, their throughputs do not continue to increase. In comparison, RomeFS utilizes the two data paths of CXL-SSD synergistically to handle user requests in parallel, thus achieving a higher saturated throughput. Moreover, RomeFS always writes to the backend SSD in the large and block-aligned pattern, thus it more effectively exploits the bandwidth of the backend SSD. Overall, this amply demonstrates the efficiency and effectiveness of RomeFS's synergistic dual-path architecture under concurrent writes.

Concurrent reads. Initially, 10GB / 20GB file data are placed in the DRAM/PM, while the remaining file data are stored in the backend SSD. RomeFS's multi-threaded read throughput is comparable to that of the block-based file systems, as shown in Figure 10(a).

The block-based file systems can quickly complete LRU replacements due to large average IO size (i.e., 512KB) and low-latency DRAM, and their saturated throughput mainly depends on the backend SSD. BTRFS performs the worst, which may be due to its design for write optimization, resulting in small random reads becoming the throughput bottleneck. RomeFS stores less than 10% file data in the DRAM, and there is a slight overhead of dual-path read thread interaction, thus RomeFS's saturation throughput is only slightly higher than the block-based file systems. RomeFS-PM outperforms NOVA and WineFS by up to 4.23× and 3.79× in multi-threaded read throughput respectively, as shown in Figure 10(b). Because under heavy loads, the LRU replacements between the SSD area and the memory area become the throughput bottleneck for these single-path PM-based file systems, especially writing the memory area (i.e., the PM). In comparison, dual-path RomeFS can avoid the passive data switch. This indicates that a file system solely using the CXL.mem path is highly reliant on the performance of the memory area of CXL-SSDs. RomeFS with synergistic dual-path architecture avoids this strict dependency. Moreover, RomeFS can further utilize *page cache* of the CXL.io path directly to speedup reads. Therefore, RomeFS can also work well on CXL-SSDs with different performance levels.

4.3 Macrobenchmarks

We use Filebench [50] with three representative workloads: Fileserver, Webproxy, and Varmail to evaluate the overall performance of RomeFS, i.e., mixed reads and writes, and mixed metadata and data accesses. Table 5 summarizes the characteristics of these workloads. We call *fsync()* to persist writes after each file-write and enable *page cache* for the backend SSD. To evaluate the overall performance of these file systems under different memory-area usage of CXL-SSDs, rather than evaluation of processing all requests on the memory area only, we set the available sizes of the DRAM/PM to 512MB for the three workloads with file-data space occupation of 2.44GB, 19.53GB and 0.61GB, respectively.

For the three workloads, RomeFS requires only 200MB DRAM/PM or less space to store metadata, affording sufficient remaining DRAM/PM space for RomeFS to work well. The baseline file systems write to backend SSD directly after the DRAM/PM is full, and perform LRU replacements when reads miss on the DRAM/PM.

The results are shown in Figure 11. For write-heavy Fileserver, the read hit rates of the DRAM/PM of the baseline file systems are approximately 20%. The performance of the block-based file systems is mainly limited by their slow metadata operations on backend SSD. The performance of the PM-based file systems is mainly limited by LRU replacements. Therefore, RomeFS(-PM) outperforms the block-based file systems and the PM-based file systems by 2.05×-4.98× and 2.59×-2.64× in throughput, respectively. Besides, the performance difference between RomeFS and RomeFS-PM under the Fileserver is less than 7%. This implies that RomeFS not only does not require a large-capacity memory area but also does not depend on the performance of the memory area under the typical and common workloads of a file server.

Table 5: Filebench workload characteristics.

Workload	#Files	Avg. File size	Total File size	IO size (r/w)	R/W
Fileserver	10K	256KB	2.44GB	1MB/256KB	1:2
Webproxy	10K	2MB	19.53GB	1MB/256KB	5:1
Varmail	10K	64KB	0.61GB	1MB/64KB	1:1

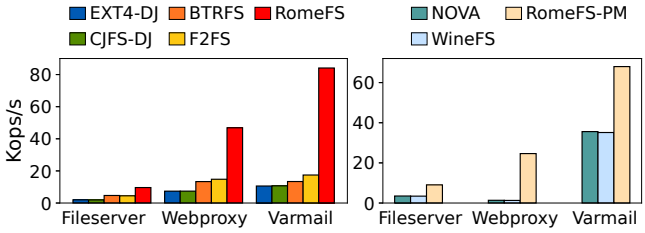


Figure 11: Throughput of Filebench.

For read-heavy Webproxy, the read hit rates of the DRAM /PM of the baseline file systems are approximately 3%. The block-based file systems process fewer metadata, thus performing better than under Fileserver. The PM-based file systems suffer from more severe LRU replacement overhead, thus performing worse than under Fileserver. RomeFS, which does not have passive data switch, can significantly benefit from *page cache*. Therefore, RomeFS(-PM) outperforms the block-based file systems and the PM-based file systems by 3.17×-6.34× and 18.10×-18.98× in throughput, respectively.

For metadata-intensive Varmail, the read hit rates of the DRAM/PM of the baseline file systems are approximately 83%. Interestingly, despite most metadata operations being handled in the DRAM, the block-based file systems still perform poorly due to their slow metadata operations on backend SSD. The PM-based file systems experience far less LRU replacement overhead, but their passive data switches still significantly restrict their performance. Therefore, RomeFS(-PM) outperforms the block-based file systems and the PM-based file systems by 4.82×-7.94× and 1.91×-1.94× in throughput, respectively. Overall, this demonstrates that RomeFS only requests a small memory area to perform well under mixed reads and writes, and metadata and data operations. Therefore, RomeFS can be well and widely used on future CXL-SSD models with different build-in DRAM capacities.

4.4 Real-World Application

GridGraph [63] is a system for processing large-scale graphs on a single machine. We utilize GridGraph as a real-world application to further evaluate the performance of RomeFS in serving emerging applications. Following GridGraph, we preprocess the Livejournal [47], Twitter [18], and Friendster [46] datasets and then execute the Pagerank algorithm [8] with an 8GB memory limit (emulating scenarios of large-scale graph processing), running 20 iterations on each graph. We trace all file operations during the processing and evaluate the replay times of file systems. Table 6 presents the workload characteristics for each dataset. Initially, the DRAM and PM are empty.

Table 6: Graph processing workload characteristics.

Dataset	|V|	|E|	Dataset Size	Writes	Reads	Average Write Size	Average Read Size
LiveJournal	4.85M	69.0M	539MB	1.8GB	12.8GB	123.1KB	16.5MB
Twitter	61.6M	1.5B	11.5GB	35.4GB	25.5GB	1.02MB	4.9MB
Friendster	68.3M	2.6B	28.2GB	86.8GB	93.4GB	129.4KB	11.0MB

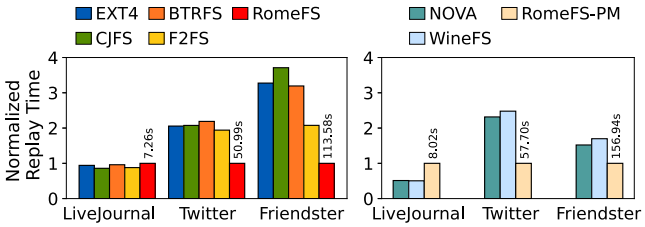


Figure 12: Normalized replay time of graph processing.

Similarly, we set the available capacity of both DRAM and PM to be 10GB to reflect different memory-area usage in the three workloads. We call *fsync()* after each file-write to ensure data persistence. Unlike the microbenchmark with completely random reads, the reads in these three workloads have some locality. To stress test the performance of RomeFS without benefiting locality, we disable its *page cache* under these workloads with locality. In contrast, due to LRU replacements, the baseline file systems can significantly benefit from such workloads with locality, even though bypassing *page cache*. For this purpose, we use read with direct IO mode to bypass *page cache*. Specifically, for unaligned reads, we read data outside the user buffer up to the alignment boundary. Although this may introduce some read amplification, it is negligible for these large-read workloads.

The results are shown in Figure 12. Apart from the LiveJournal load, RomeFS outperforms EXT4, CJFS, BTRFS and F2FS by up to 3.28×, 3.71×, 3.19×, and 2.08× respectively, and RomeFS-PM outperforms NOVA and WineFS by up to 2.32× and 2.48× in replay time respectively.

For the LiveJournal load, after the DRAM/PM is filled, the baseline file systems complete most (more than 95%) requests in the DRAM/PM. In comparison, RomeFS(-PM) only uses less than 1GB DRAM/PM. Therefore, the replay times of PM-based file systems are about half that of RomeFS-PM. The block-based file systems still handle a small amount of metadata operations on the backend SSD, so their replay times are comparable to RomeFS. This indicates that, despite RomeFS not switching data to the memory area for reads, its performance remains competitive even under such small read-heavy loads with locality.

For the Twitter load, RomeFS only uses less than 7GB DRAM/PM, but still significantly outperforms the baseline file systems. Although the latter can benefit from reads with locality, it is difficult for them to efficiently handle writes, similar to the write evaluation of microbenchmarks.

For the Friendster load, it is heavier than the Twitter load and has higher read locality. Due to more metadata updates,

the DRAM exhaustion has a more severe impact on the performance of the block-based file systems. The PM-based file system performs better under this load than under Twitter load due to Friendster's higher read locality, but their replay times are still significantly longer than RomeFS-PM. This indicates that even when running the real application with some read locality, dual-path RomeFS remains superior to the legacy single-data-path file systems. Enabling *page cache* for RomeFS will significantly amplify this advantage.

5 RELATED WORK

CXL memory. Pond [25] utilizes CXL memory to improve DRAM memory pooling in cloud environments. TPP [30] proposes a novel OS-level application-transparent page placement mechanism for CXL memory. Caption [48] comprehensively evaluates a true CXL-ready system with three CXL memory devices and proposes a CXL-memory-aware dynamic page allocation policy to more efficiently use CXL memory as a bandwidth expander. DirectCXL [9] connects host processors with external DRAM via CXL in real hardware and develops a software runtime to directly access the resources. Liu et al. [28] study how HPC applications and large language models can benefit from the CXL memory, and study the interplay between memory tiering and page interleaving. Intel Flat Memory Mode [59] combines hardware-managed tiering with software-managed performance isolation to improve performance of the memory tiers system for CXL. Due to the similarity between the build-in DRAM cache of CXL-SSDs and CXL memory, RomeFS can indirectly draw inspiration and benefit from these works.

CXL-SSD. A main body of CXL-SSD research focuses on the memory expansion mode via the CXL.mem path [11, 15, 20, 55]. Myoungsoo Jung [15] advocates combining CXL and SSD to expand host memory. CXL-flash [55] and ExPAND [20] further study the cache policies and prefetching algorithms to hide long internal flash-access latencies of CXL-SSD. RomeFS is orthogonal to these works and benefits from these works directly to improve the performance of the CXL.mem path.

Block-based file systems, such as EXT4 [31], XFS [49], BTRFS [38] and F2FS [22], are dedicated to block devices and the block-based data path via the legacy IO stack [23]. The latest works mainly focus on the scalability of file systems. For example, CJFS [34] introduces concurrent journaling based on EXT4, MAX [27] proposes multicore-accelerated optimization based on F2FS, and IPLFS [19] develops a log-structured file system that is free from garbage collection based on F2FS. In addition, RomeFS's journal commit strategy is orthogonal to FastCommit [44]. RomeFS decouples data and metadata committing on the basis of "logical journaling". FastCommit performs "logical journaling" for simple and frequent modifications, while relying on JBD2 for more complex and rare modifications.

PM-based file systems, such as PMFS [6] and NOVA [53], are designed specifically for byte-addressable persistent memory and the memory-semantic data path. The latest WineFS [17] is a hugepage-aware PM-based file system and designed to eliminate the aged effect, and OdinFS [61] is a NUMA-aware scalable datapath PM-based file system and is used for parallelizing the access to multiple PM across NUMA nodes. In addition, many PM-based file systems focus on further bypassing the kernel and accessing PM in userspace, at a potential compromise for security and consistency, such as SplitFS [16], ZoFS [5], KucoFS [2], ctFS [26] and MadFS [58]. The latest ArckFS [60] is designed for enabling high-performance and secure userspace simultaneously. For a fair comparison, we compare (in-kernel) RomeFS against in-kernel NOVA and WineFS, rather than the userspace file systems mentioned above.

Cross-media hybrid file systems tend to deploy memory-semantic devices (e.g., PM) and block devices using caching or tiering architectures. NVMFS [37] uses PM to store hot data while using SSD to store cold data. Strata [21] and Ziggurat [57] are tiered file systems and consider PM as the upper performance tier and disks as the lower capacity tier. The latest SPFS [52] further exploits the VFS cache of the underlying block-based file system for non-synchronous writes. Overall, they focus on the memory-semantic data path and use PM and SSD by various caching strategies without considering per-request dual-path synergy, like the memory expansion mode of CXL-SSDs does.

6 CONCLUSION AND FUTURE WORK

This paper presents RomeFS, the first CXL-SSD aware file system, that utilizes the CXL.mem and CXL.io data paths of CXL-SSD simultaneously by synergistically and preferentially handling file requests in parallel, thus improving overall performance. Experimental results show that, compared to the file systems which only use a single type of data path, RomeFS has notably superior performance across a wide range of load evaluations.

Our future work plans to explore the dynamic write splitting threshold based on the memory area capacity, dual-path performance, and access patterns of applications.

ACKNOWLEDGMENTS

We thank our shepherd Abhishek Chandra and the anonymous reviewers for their insightful comments. This work was supported by NSFC No.62172175, No.61821003, and Key Research and Development Project of Hubei No.2022BAA042.

REFERENCES

[1] Zhigang Cai, Chengyong Tang, Minjun Li, François Trahay, Jun Li, Zhibing Sha, Jiaojiao Wu, Fan Yang, and Jianwei Liao. 2023. Re-aligning across-page requests for flash-based solid-state drives. In *Proceedings of the 52nd International Conference on Parallel Processing*. 736–745.

[2] Youmin Chen, Youyou Lu, Bohong Zhu, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Jiwu Shu. 2021. Scalable Persistent Memory File System with Kernel-Userspace Collaboration. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 81–95.

[3] Douglas Comer. 1979. Ubiquitous B-tree. *ACM Computing Surveys (CSUR)* 11, 2 (1979), 121–137.

[4] CXL Consortium. 2023. Compute Express Link (CXL). https://www.computeexpresslink.org.

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

[6] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, and et al. 2014. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*. 1–15.

[7] Flashsumit2023. 2023. Data Centric Compute and Data Tiering With CXL. https://www.flashmemorysummit.com/English/Conference/Proceedings_Chrono.html.

[8] Dániel Fogaras, Balázs Rácz, Károly Csalogány, and Tamás Sarlós. 2005. Towards scaling fully personalized pagerank: Algorithms, lower bounds, and experiments. *Internet Mathematics* 2, 3 (2005), 333–358.

[9] Donghyun Gouk. 2022. Direct access,{High-Performance} memory disaggregation with {DirectCXL}. In *USENIX ATC 22*. 287–294.

[10] Shuibing He, Matthew Myers, Xuehao Duan, Keegan Sanchez, and Xuechen Zhang. 2022. WAFLASH: Taming Unaligned Writes in Solid-State Disks. In *2022 IEEE International Conference on Networking, Architecture and Storage (NAS)*. IEEE, 1–8.

[11] Weizhou Huang, Jian Zhou, and et al. 2024. TieredHM: Hotspot-Optimized Hash Indexing for Memory Semantic SSD Based Hybrid Memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2024).

[12] Intel. 2022. CXL™-Attached Memory. https://www.intel.cn/content/dam/www/central-libraries/us/en/documents/2022-11/optane-pmem-to-cxl-tech-brief.pdf.

[13] Intel. 2023. Samsung PM9A3 SSD. https://semiconductor.samsung.com/ssd/datacenter-ssd/pm9a3/.

[14] Intel. 2024. Distributed Asynchronous Object Storage (DAOS). https://docs.daos.io/v2.7/.

[15] Myoungsoo Jung. 2022. Hello bytes, bye blocks: PCIe storage meets compute express link for memory expansion (CXL-SSD). In *HotStorage 22*. 45–51.

[16] Rohan Kadekodi. 2019. SplitFS: Reducing software overhead in file systems for persistent memory. In *SOSP 19*. 494–508.

[17] Rohan Kadekodi, Saurabh Kadekodi, and et al. 2021. WineFS: a hugepage-aware file system for persistent memory that ages gracefully. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 804–818.

[18] KAIST. 2023. Twitter. http://an.kaist.ac.kr/traces/WWW2010.html.

[19] Juwon Kim, Minsu Kim, Muhammad Danish Tehseen, Joontaek Oh, and YouJip Won. 2022. IPLFS: Log-Structured File System without Garbage Collection. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 739–754.

[20] Miryeong Kwon. 2023. Cache in Hand: Expander-Driven CXL Prefetcher for Next Generation CXL-SSD. In *HotStorage 23*. 24–30.

[21] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 460–477.

[22] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. 2015. F2FS: A New File System for Flash Storage. In *FAST 15*. 273–286.

[23] Gyusun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W Lee, and Jinkyu Jeong. 2019. Asynchronous I/O stack: A low-latency kernel I/O stack for Ultra-Low latency SSDs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 603–616.

[24] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE 13*. IEEE, 38–49.

[25] Huaicheng Li. 2023. Pond: CXL-based memory pooling systems for cloud platforms. In *ASPLOS 23, Volume 2*. 574–587.

[26] Ruibin Li, Xiang Ren, Xu Zhao, Siwei He, Michael Stumm, and Ding Yuan. 2022. ctFS: Replacing File Indexing with Hardware Memory Translation through Contiguous File Allocation for Persistent Memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. 35–50.

[27] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. 2021. Max: A Multicore-Accelerated File System for Flash Storage. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 877–891.

[28] Jie Liu, Xi Wang, Jianbo Wu, Shuangyan Yang, Jie Ren, Bhanu Shankar, and Dong Li. 2024. Exploring and Evaluating Real-world CXL: Use Cases and System Adoption. *arXiv preprint arXiv:2405.14209* (2024).

[29] Shuyang Liu, Shucheng Wang, Qiang Cao, Ziyi Lu, Hong Jiang, Jie Yao, Yuanyuan Dong, and Puyuan Yang. 2019. Analysis of and optimization for write-dominated hybrid storage nodes in cloud. In *Proceedings of the ACM Symposium on Cloud Computing*. 403–415.

[30] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent page placement for CXL-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 742–755.

[31] Avantika Mathur. 2007. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, Vol. 2. Citeseer, 21–33.

[32] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. 2016. Understanding manycore scalability of file systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 71–85.

[33] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. 2021. CheckFreq: Frequent, Fine-Grained DNN Checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 203–216.

[34] Joontaek Oh, Seung Won Yoo, Hojin Nam, Changwoo Min, and Youjip Won. 2023. CJFS: Concurrent Journaling for Better Scalability. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 167–182.

[35] Kiet Tuan Pham, Seokjoo Cho, Sangjin Lee, Lan Anh Nguyen, Hyeongi Yeo, Ipoom Jeong, Sungjin Lee, Nam Sung Kim, and Yongseok Son. 2024. ScaleCache: A Scalable Page Cache for Multiple Solid-State Drives. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 641–656.

[36] Yingjin Qian, Marc-André Vef, Patrick Farrell, Andreas Dilger, Xi Li, Shuichi Ihara, Yinjin Fu, Wei Xue, and André Brinkmann. 2024. Combining Buffered I/O and Direct I/O in Distributed File Systems. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*. 17–33.

[37] Sheng Qiu and AL Narasimha Reddy. 2013. NVMFS: A hybrid file system for improving random write in nand-flash SSD. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 1–5.

[38] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS)* 9, 3 (2013), 1–32.

[39] Samsung. 2023. CMM-H (CXL Memory Module, H: Hybrid). https://samsungmsl.com/cmmh/.

[40] Samsung. 2023. Samsung SSD. https://semiconductor.samsung.com/ssd/.

[41] Samsung. 2024. CMM-H. https://semiconductor.samsung.com/us/news-events/tech-blog/webinar-memory-semantic-ssd/.

[42] Samsung. 2024. Understand how the CXL SSD can aid performance. https://www.techtarget.com/searchstorage/feature/Understand-how-the-CXL-SSD-can-aid-performance.

[43] Debendra Das Sharma. 2023. Compute Express Link™(CXL™): An Open Interconnect for Cloud Infrastructure. In *DAC 23*. IEEE, 1–4.

[44] Harshad Shirwadkar, Saurabh Kadekodi, and Theodore Tso. 2024. Fast-Commit: resource-efficient, performant and cost-effective file system journaling. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 157–171. https://www.usenix.org/conference/atc24/presentation/shirwadkar

[45] Jiwu Shu, Fei Li, Siyang Li, and Youyou Lu. 2020. Towards unaligned writes optimization in cloud storage with high-performance ssds. *IEEE Transactions on Parallel and Distributed Systems* 31, 12 (2020), 2923–2937.

[46] Stanford. 2023. Friendster. https://snap.stanford.edu/data/com-Friendster.html.

[47] Stanford. 2023. LiveJournal. https://snap.stanford.edu/data/soc-LiveJournal1.html.

[48] Yan Sun, Yifan Yuan, and et al. 2023. Demystifying cxl memory with genuine cxl-ready systems and devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 105–121.

[49] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. 1996. Scalability in the XFS File System.. In *USENIX Annual Technical Conference*, Vol. 15.

[50] Vasily Tarasov. 2018. Filebench-A Model Based File System Workload Generator. https://github.com/filebench/filebench.

[51] Shucheng Wang, Ziyi Lu, Qiang Cao, Hong Jiang, Jie Yao, Yuanyuan Dong, and Puyuan Yang. 2020. BCW: Buffer-Controlled Writes to HDDs for SSD-HDD Hybrid Storage Server. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 253–266.

[52] Hobin Woo, Daegyu Han, Seungjoon Ha, Sam H Noh, and Beomseok Nam. 2023. On stacking a persistent memory file system on legacy file systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 281–296.

[53] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *FAST 16*. 323–338.

[54] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 169–182.

[55] Shao-Peng Yang. 2023. Overcoming the Memory Wall with CXL-Enabled SSD. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 601–617.

[56] Yang Yang, Qiang Cao, Jie Yao, Yuanyuan Dong, and Weikang Kong. 2021. Spmfs: A scalable persistent memory file system on optane persistent memory. In *Proceedings of the 50th International Conference on Parallel Processing*. 1–10.

[57] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. 2019. Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 207–219.

[58] Shawn Zhong, Chenhao Ye, Guanzhou Hu, Suyan Qu, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, and Michael Swift. 2023. MadFS:Per-File Virtualization for Userspace Persistent Memory Filesystems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 265–280.

[59] Yuhong Zhong, Daniel S Berger, Carl Waldspurger, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D Hill, Mosharaf Chowdhury, and Asaf Cidon. 2024. Managing Memory Tiers with CXL in Virtualized Environments. In *Symposium on Operating Systems Design and Implementation*.

[60] Diyu Zhou, Vojtech Aschenbrenner, and et al. 2023. Enabling High-Performance and Secure Userspace NVM File Systems with the Trio Architecture. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 150–165.

[61] Diyu Zhou, Yuchen Qian, and et al. 2022. ODINFS: Scaling PM Performance with Opportunistic Delegation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 179–193.

[62] Qingsong Zhu, Qiang Cao, and Jie Yao. 2023. UHS: An Ultra-fast Hybrid Storage Consolidating NVM and SSD in Parallel. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1–6.

[63] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 375–386.