Rethinking the Request-to-IO Transformation Process of File Systems for Full Utilization of High-Bandwidth SSDs

Yekang Zhan, Haichuan Hu, Xiangrui Yang, and Qiang Cao, *Huazhong University of Science and Technology;* Hong Jiang, *University of Texas at Arlington;* Shaohua Wang and Jie Yao, *Huazhong University of Science and Technology*

https://www.usenix.org/conference/fast25/presentation/zhan

This paper is included in the Proceedings of the 23rd USENIX Conference on File and Storage Technologies.

February 25–27, 2025 • Santa Clara, CA, USA

ISBN 978-1-939133-45-8

Open access to the Proceedings of the 23rd USENIX Conference on File and Storage Technologies is sponsored by

Rethinking the Request-to-IO Transformation Process of File Systems for Full Utilization of High-Bandwidth SSDs

Yekang Zhan[1], Haichuan Hu[1], Xiangrui Yang[1], Qiang Cao[1*], Hong Jiang[2], Shaohua Wang[1] and Jie Yao[1]

[1]Huazhong University of Science and Technology,
[2]University of Texas at Arlington

Abstract

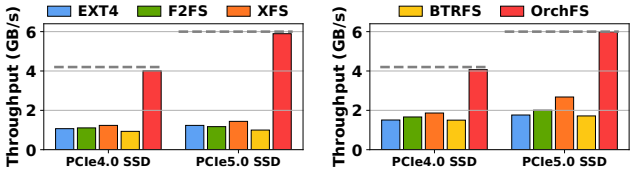
The capacity and bandwidth of modern Solid-State Drives (SSDs) have been steadily increasing in recent years. Unfortunately, existing SSD file systems that transform user requests to memory-page aligned homogeneous block IOs have by and large failed to make full use of the superior write bandwidth of SSDs even for large writes. Our experimental analysis identifies three main root causes of this write inefficiency, namely, 1) SSD-page alignment cost, 2) page caching overhead, and 3) insufficient IO concurrency.

To fully exploit the potentials offered by modern SSDs, this paper proposes a heterogeneous-IO orchestrated file system with an alignment-based write-partition, or OrchFS, that leverages a small-size NVM (Non-Volatile Memory) to maximize SSD performance. OrchFS extends and improves the request-to-IO transformation functionality of file systems to proactively transform file-writes into SSD-page aligned SSD-IOs and/or remaining SSD-page unaligned NVM-IOs, and then to perform these IOs via their respective optimal data paths and in an explicit multi-threaded manner. To this end, OrchFS presents several novel enabling techniques, including heterogeneous-unit data layout, alignment-based file write partition, unified per-file mapping structure and embedded parallel IO engine. The experimental results show that OrchFS outperforms 1) EXT4 and F2FS on SSD, 2) NOVA, OdinFS and ArckFS on NVM, and 3) Strata, SPFS and PHFS on hybrid NVM-SSD by up to 29.76× and 6.79× in write and read performances, respectively.

1 Introduction

Solid-State Drives (SSDs) with increasing capacity and performance are supplementing and poised to replace Hard Disk Drives (HDDs) as primary block storage. SSD file systems [6, 33, 37, 44, 50, 57, 68] maintain a general file/directory abstraction for a myriad of applications while internally performing request-to-IO transformation to generate one or more memory-page aligned block IOs (*bio*s) and then sending them to the underlying SSD.

*Corresponding author. Email: caoqiang@hust.edu.cn



(a) Write size range: 1B - 2MB (b) Write size range: 0.9MB - 1.1MB

Figure 1: Random write throughput of four representative SSD file systems and OrchFS with FIO [3] single-threaded benchmark (ioengine=psync) on a PCIe4.0 SSD [59] and a PCIe5.0 SSD [58], respectively. In all cases, OrchFS writes less than 5% of the data into NVM.

However, existing SSD file systems fail to keep pace with the significant advances in write performance of SSDs. Real-world data-intensive applications, e.g., graph processing [90], scientific computing [54], and cloud computing [74] generally perform writes with diverse offsets and sizes. Figure 1(a) shows that the throughput of random writes, under four representative SSD file systems of EXT4 [6], F2FS [37], XFS [68] and BTRFS [57], are only about 1/3 and 1/4 of the raw-bandwidth of the PCIe4.0 and PCIe5.0 SSDs, respectively. Even for large random writes of size 1MB ± 10%, as shown in Figure 1(b), their throughput is still less than 1/2 of the corresponding SSD raw bandwidth. This means that the write inefficiency is not solely caused by small writes.

We conduct a set of experiments to holistically identify and analyze root causes of the write inefficiency of existing SSD file systems (detailed in § 2.4) and draw three key observations. ***Observation #1*** reveals that SSD-page (e.g., 16KB) unaligned writes suffer from high alignment costs, e.g., by up to 10.71× higher latency than aligned writes. Although page cache can cache reads and buffer writes to reduce the alignment cost, ***Observation #2*** indicates that writes via the buffered IO mode with page cache introduce costly software overheads, while writes via the direct IO mode can avoid such overheads under strict alignment guarantees. Furthermore, ***Observation #3*** discovers that a large and aligned write via the direct IO mode still fails to fully utilize SSD bandwidth, while explicit IO-splitting and multi-threaded IO execution can. As

a result, we argue that an optimal SSD-write path is possible, but with strict conditions, including SSD-page-alignment, the direct IO mode, and explicit multi-threaded IO processing. Existing SSD file systems, which transform any-pattern requests with diverse offsets and sizes to memory-page aligned and homogeneous *bio*s, rarely meet such strict conditions simultaneously to perform the SSD-write optimally.

Emerging Non-Volatile Memories (NVMs) [2, 18, 26, 67, 80] with byte-accessibility and fast persistence favor small accesses without high alignment costs, which are notably complementary in IO-pattern preferences to SSDs. This inspires our ***insight***: the increasingly high write performance of SSDs can be maximally utilized if all IOs to SSDs are fully aligned while NVM can fast absorb residual unaligned small IOs.

Motivated by this insight to rethink the request-to-IO transformation functionality of file systems, this paper proposes a heterogeneous-IO orchestrated file system, or OrchFS. OrchFS's request-to-IO transformation first proactively partitions a file write to SSD-page aligned SSD-IOs and/or the residual small and unaligned NVM-IOs. Then, this transformation adopts a novel triple-data-path approach that utilizes the above optimal SSD-write path for the partitioned SSD-IOs, and uses the legacy memory-semantic NVM data path for the partitioned NVM-IOs, while enabling the buffered IO mode for the SSD-read path because reads with page cache naturally benefit from (large) SSD-page aligned writes.

OrchFS's alignment-based request-to-heterogeneous-IO transformation introduces three key challenges: how to partition and map any-pattern writes to heterogeneous-storage IOs effectively and efficiently (**C1**), how to fast index for all partitions across NVMs and SSDs (**C2**), and how to effectively integrate and orchestrate the aforementioned triple data paths (**C3**). To address **C1**, OrchFS presents: 1) a heterogeneous data layout with three types of storage-units and 2) an efficient file write partition strategy with two specific policies of alignment-prioritization and fragmentation-minimization. To overcome **C2**, OrchFS designs a unified and cache-friendly indexing scheme to locate the split and scattered data. For **C3**, OrchFS builds an embedded parallel IO engine to coordinate the triple-path IOs and ensure data consistency. In addition, OrchFS inherits and extends prior techniques of both SSD and NVM file systems to further reduce IO penalties (e.g., LibFS-KernelFS architecture) and ensure crash consistency (e.g., journaling). Overall, OrchFS highly coordinates and synergizes the file system functionalities and IO characteristics of hybrid storage by transforming various user accesses into device-preferred heterogeneous IOs at runtime.

In comparison, existing hybrid NVM-SSD systems either use the upper-layer NVMs as a cache to the lower-layer SSDs to absorb most workloads [8, 36, 41, 45, 55, 76, 85], thus underutilizing high-bandwidth SSDs, or leverage NVM and SSD to handle small and large writes respectively [27, 89] but ignore the write inefficiency of SSDs.

We implement an OrchFS prototype to provide over 30 commonly used POSIX APIs [71] for applications and evaluate it under a variety of workloads and real applications. The results show that OrchFS can efficiently handle a wide variety of workloads with diverse access patterns and outperforms representative SSD file systems (EXT4 [6] and F2FS [37]), NVM file systems (NOVA [78], OdinFS [88], and ArckFS [87]), and hybrid NVM-SSD file systems (Strata [36], SPFS [76], and PHFS [27, 89]) by up to 29.76×, 3.49× and 7.16× in write latency, and by up to 3.08×, 6.79× and 6.34× in peak read throughput, respectively.

The major contributions of this work are:

- We experimentally identify and analyze the root causes of write inefficiency of existing file systems running on high-bandwidth SSDs.
- We propose OrchFS, the first SSD-NVM heterogeneous-IO orchestrated file system with a unique alignment-based request-to-heterogeneous-IO transformation, and present several novel enabling techniques.
- We implement an OrchFS prototype and evaluate it against the state-of-the-art and representative file systems on SSD, NVM and hybrid SSD-NVM under a variety of workloads and real applications.

2 Background and Motivation

In this section, we first provide the necessary background for SSDs, NVMs, and file systems, then describe our experimental analysis to identify the root causes of write inefficiency of SSD file systems, which motivates our OrchFS design.

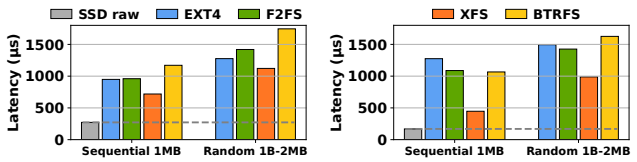
2.1 SSDs and NVMs

SSDs have been ubiquitously deployed, from Internet of Things (IoT) [1], mobile devices [37], servers [4], to large-scale datacenters [74]. The storage density of SSDs has been increasing steadily by increasing bits per NAND cell and using 3D-structured flash fabric [16, 46]. An SSD-page, as a basic read/write flash-unit, e.g., 16KB, is larger than a 4KB memory-page. In the meantime, the SSD bandwidth also keeps increasing, due to scalable IO parallelism among multiple flash, chips and dies within an SSD. For example, the read and write bandwidths of PCIe4.0 NVMe SSDs are approximately 5GB/s-7GB/s and 3GB/s-6GB/s, respectively, which are further doubled in current PCIe5.0 SSDs [60, 75].

Emerging NVMs support byte-addressability with fast persistence, but have much higher per capacity price and smaller capacity than SSDs. Although Intel Optane DCPMM [26] has discontinued, RRAM [67], PCRAM [18], MRAM [2] and memory-semantic SSDs [80] have continued to evolve.

2.2 File Systems

File systems have been playing a pivot role in exploiting underlying storage devices to offer a logical file/directory view and file operations for upper applications. Using the legacy POSIX interface [71], applications invoke reads/writes with



(a) on a PCIe4.0 SSD (b) on a PCIe5.0 SSD

Figure 2: Average write latency of four representative SSD file systems with FIO single-threaded benchmark (ioengine=psync and fsync=1) on the PCIe4.0 SSD [59] and the PCIe5.0 SSD [58], respectively.

byte-addressable offset and length. File systems transform any-pattern user-requests to device-allowable IOs.

Mainstream SSD file systems [37, 48, 57, 68] considering SSD characteristics still inherit a legacy block storage stack [38] with page cache to transform a file request to one or more 512B/page-aligned block IOs (*bio*s) according to the underlying physical page-address. Each *bio* corresponds to one or more contiguous physical pages. For page-unaligned writes missed in page cache, file systems use a read-modify-write (RMW) process [37, 48, 54, 63] to forcefully align write requests to page-boundary, e.g., 4KB, before submitting *bio*s. The RMW process first reads the corresponding page from the SSD, then updates the page, and finally writes the aligned page to the SSD. Furthermore, when a host write is SSD-page unaligned and missed SSD's internal cache, the SSD has to perform similar RMWs to align SSD-pages internally [5, 22, 24]. Both page cache and SSD's internal cache can mitigate IO misalignment when the requested data is cached.

2.3 Write Performance of SSD File Systems

With the increasing bandwidth of SSDs, SSD file systems are expected to improve their overall performance accordingly. However, mainstream SSD file systems with the default buffered IO mode fail to effectively exploit the high write-bandwidth promised by SSDs.

Figure 1 (single-threaded FIO benchmark, with sufficient memory-pages) demonstrates that such write-inefficiency is a common phenomenon for prevalent unaligned write-requests, regardless of the write sizes.

To further understand the actual write performance unhidden by page cache, we use FIO [3] with the same configuration as Figure 1 but with fsync=1 (i.e., calling `fsync()` after each write) to perform two representative workloads [7, 23, 47, 74, 84, 90], i.e., the sequential and large 1MB-sized writes as an ideal case, and the random 1B-2MB-sized writes as a common case. As shown in Figure 2, the sequential write latencies of the file systems are 2.64×- 4.31× and 2.65×- 7.54× of the raw 1MB-write latency of the PCIe4.0 SSD and the PCIe5.0 SSD respectively, and the random write latencies of the file systems are 4.13×- 6.42× and 5.85×- 9.62× of that respectively. This means that, under strict persistence, 1) even aligned large writes suffer from write-inefficiency, and 2) unaligned writes suffer from more severe write-inefficiency.

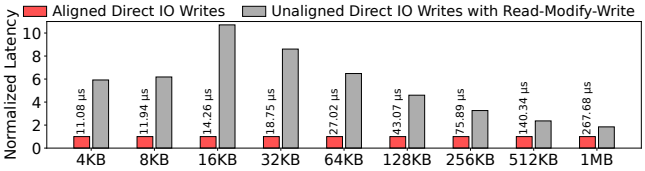


Figure 3: Normalized random write latency of aligned direct IO writes and unaligned direct IO writes with read-modify-write on the PCIe4.0 SSD [59].

2.4 Analysis on Write Inefficiency

Next, we comprehensively and experimentally explore and identify the root causes of the write-inefficiency of SSD file systems on high-bandwidth SSDs.

2.4.1 IO Alignment

For an unaligned write, the request-to-IO transformation process of file systems first aligns it. To better understand IO alignment costs of file systems without an impact of page cache, we use the direct IO mode to perform random writes and measure their latencies. Compared to the default buffered IO mode, the direct IO mode bypasses page cache but demands strict alignment among the IO size, offset, and memory-buffer address [54]. We use the common RMW process (§ 2.2) on host to align unaligned writes.

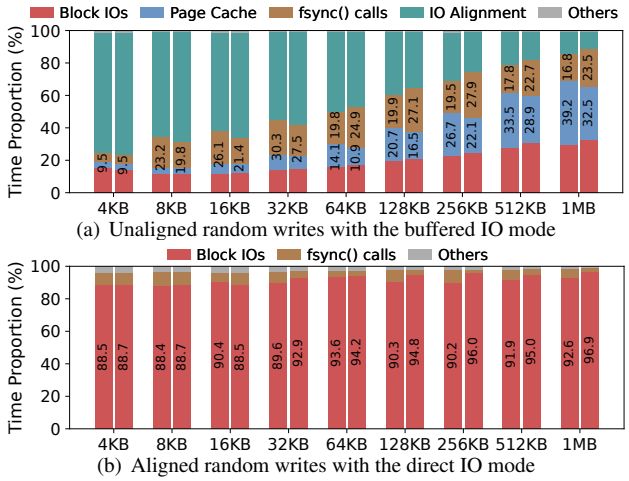
Figure 3 shows that unaligned direct IO writes have significantly higher latency (1.85×-10.71×) than aligned direct IO writes. Because an unaligned direct IO write is forced to perform one or two extra read-IOs before writing. This introduces the additional read latency and IO amplification, referred to as *alignment cost*. In particular, the 16KB unaligned writes read and then write two adjacent SSD-pages [5], thus causing the highest alignment cost (10.71×). Unaligned writes smaller than 16KB read and then write one or two entire SSD-pages. Even for the 1MB large-writes, the unaligned case still results in 1.85× higher write latency than the aligned case mainly due to reading two SSD-pages at the head and tail of the write [22, 24]. In conclusion, we draw ***Observation #1****: unaligned writes have significant alignment costs.*

2.4.2 Page Cache

Page cache is widely used to speed up reads/writes at the cost of memory usage and complex cache management. With the narrowing performance gap between memory and fast storage, the overhead of page caching is no longer negligible.

To comprehensively understand the entire write data path using page cache upon high-bandwidth SSDs, we perform a 10GB (unaligned) random write experiment with the buffered IO mode as a common case on EXT4 and F2FS. To clearly break down the write time into page caching, IO alignment, `fsync()` calls, block IOs, and others, we clear page cache before each running, call `fsync()` after each write, insert corresponding timing breakpoints into the kernel source codes of EXT4 and F2FS, and run three times to take the average.

Figure 4(a) shows the breakdown results. Compared to Figure 3, Figure 4(a) exhibits a still significant, albeit slightly re-



(a) Unaligned random writes with the buffered IO mode

(b) Aligned random writes with the direct IO mode

Figure 4: Write time breakdown of EXT4 (left bars) and F2FS (right bars).

duced, alignment cost. This is because, as data is progressively cached, a small number of unaligned writes are absorbed by page cache without extra read-IOs [63]. Nevertheless, this demonstrates that when data is not cached, it remains costly for file systems with page cache to enforce IO alignment.

Furthermore, other than IO alignment time, performing block IO takes only 11.5%-32.9% of the write time. As the write size increases, page caching takes an increasing portion of time, e.g., 39.2% (average latency of 362.2μs) and 32.5% (average latency of 314.2μs) for EXT4 and F2FS respectively in the 1MB case. The page caching mainly performs page allocation, locking, searching, LRU list management, and copying data between the application and page cache. Previous works have also observed expensive page caching [15, 43, 51, 54]. The fsync, excluding block IO times, accounts for 5.5%-27.9% (average latency of up to 227.3μs) of the write time for EXT4 and F2FS. The fsync mainly performs looking up and assembling dirty pages into *bios*, as well as finally marking flushed pages as clean [53]. Even if applications do not call `fsync()`, they still have to perform page caching and IO alignment, e.g., Figure 1.

Moreover, we experimentally verify that page caching and `fsync()` calls take similar amounts of time in aligned and unaligned buffered IO writes. This means that page cache introduces significant software overhead, e.g., 9.5%-56.0% for unaligned writes and 15.9%-65.8% for aligned writes.

For comparison, we further perform 10GB aligned random writes via the direct IO mode on EXT4 and F2FS respectively. Similarly, we call `fsync()` after each write. Figure 4(b) shows the write time breakdown. Executing block IOs takes most of the write time (i.e., 88.4%-96.9%). The fsync call only accounts for 1.9%-8.5% (average latency of up to 17.3μs) of the write time for EXT4 and F2FS, respectively, because the fsync call without handling page cache only uses the FLUSH command to notify SSD [82]. Moreover, the average latency of aligned direct IO writes is only 12.9%-34.6% of the average latency of the corresponding unaligned buffered IO writes.

Figure 5: Throughput of single-threaded large aligned IOs vs. multi-threaded split 32KB aligned IOs. SSD A: the PCIe4.0 SSD [59]. SSD B: the PCIe5.0 SSD [58].

This demonstrates the performance potential of the direct IO mode. As a conclusion, we draw ***Observation #2***: *buffered IO writes suffer from costly page caching software overhead, while direct IO writes can avoid such overheads under the strict alignment.*

2.4.3 IO Concurrency

Even ideal aligned large writes via the direct IO mode still fail to maximize SSD bandwidth by a single thread, e.g., all cases use at most 89.0% of the SSD raw bandwidth in Figure 3. This is largely due to internal complex IO scheduling [22, 29] and fairness for multi-streams [70] of modern SSDs.

To understand the impact of IO concurrency for bandwidth utilization of SSDs, we set two IO patterns: 1) a single thread with large aligned IOs, and 2) splitting each large IO into multiple 32KB IOs and handling them using multiple threads. We evaluate the throughput of both IO patterns on the PCIe4.0 SSD and the PCIe5.0 SSD via the direct IO mode. Figure 5 shows that the multi-threaded IO pattern outperforms the single-threaded IO pattern in all cases. The write and read throughputs in the multi-threaded IO pattern are 1.02×-1.56× and 1.06×-1.65× higher than that of the single-threaded IO pattern. In conclusion, we draw ***Observation #3***: *a large and SSD-page aligned read/write by a single thread via the direct IO mode still does not fully utilize the bandwidth, but splitting it into multiple smaller SSD-page aligned IOs executed in a multi-threaded manner can maximize SSD bandwidth.*

In summary, existing SSD file systems fail to take full write-advantage of high-bandwidth SSDs because of IO misalignment, page caching overhead, and insufficient IO concurrency. Therefore, we argue that an optimal write path for high-bandwidth SSDs should be via the direct IO mode and multi-threaded IO processing for SSD-page aligned IOs. Further, reads without extra IOs have less penalty of misalignment than writes and can naturally benefit from aligned writes, multi-threaded IO processing and page cache without the fsync cost. Therefore, we believe that the legacy data path with page cache would remain the best for SSD-reads with SSD-page aligned writes.

2.5 Motivation

With the emergence of highly data-intensive applications such as machine learning [49], cloud computing [74], graph processing [90], databases [23], etc., it is desirable for file systems to fully expolit the growing IO capability of modern SSDs with cost-effectiveness to provide all-rounded supports for

intensive and mixed load patterns (e.g., mixed small and large, unaligned and aligned, and reads and writes), as well as high scalability and fast persistence. However, due to the write-inefficiency, existing SSD file systems with the legacy block storage stack fall far short of performing optimally on high-bandwidth SSDs under various workloads. Although there are the optimal-yet-rigorous SSD data paths to overcome the write-inefficiency, the question of how to keep SSDs working in the optimal data paths remains unanswered.

Emerging NVMs are particularly friendly for small accesses without high alignment costs [79], but are limited cost-effectiveness compared to SSDs. Nevertheless, leveraging a small NVM as an auxiliary device could provide an opportunity for SSDs to keep working in the optimal data paths. This insight, combined with the three observations, leads to a key ***design principle***: *proactively partitioning an any-pattern write to an SSD-preferred SSD-page aligned part and a residual, small and unaligned part, so that the former and latter are strategically and fast served by SSD and NVM respectively*. This design principle does not demand an NVM with large capacity, high bandwidth and scalability, but only leverages its capability for low-latency and byte-addressable accesses. This motivates us to design a novel file system using an alignment-based write-partition to extend and improve the request-to-IO transformation functionality of file systems to orchestrate requests as device-preferred heterogeneous IOs, called OrchFS.

Existing hybrid NVM-SSD systems can be roughly classified into two categories: hierarchical architecture and parallel architecture. The former adopts fast NVM as an upper-tier cache and the underlying SSDs as capacity devices [8, 36, 41, 45, 55, 76, 85]. Some works further redirect overloaded or asynchronous requests to SSD [76, 85], or conditionally uses NVM to buffer synchronized dirty pages of SSD [8]. Overall, in the hierarchical architecture, the upper NVMs bear the blunt of the load, thus losing the opportunity to fully utilize the underlying high-bandwidth SSDs. The parallel architecture strategically sends requests to NVMs and SSDs in parallel. NHC [77] mainly utilizes SSDs to accelerate reads. UHS [89] and Intel DAOS [27, 28] send size-threshold-based large and small writes to SSDs and NVMs respectively.

OrchFS proactively partitions a write into aligned part for SSD and the remaining part for NVM, which is fundamentally different from above existing NVM-SSD architectures.

3 OrchFS Design

We propose OrchFS that, to the best of our knowledge, is the first SSD-NVM heterogeneous-IO orchestrated file system to judiciously leverage small auxiliary NVM to maximize the performance of high-bandwidth SSDs.

3.1 Design Challenges

Governed by the ***design principle*** (§ 2.5), the OrchFS design must tackle the following three key challenges (**Cs**).

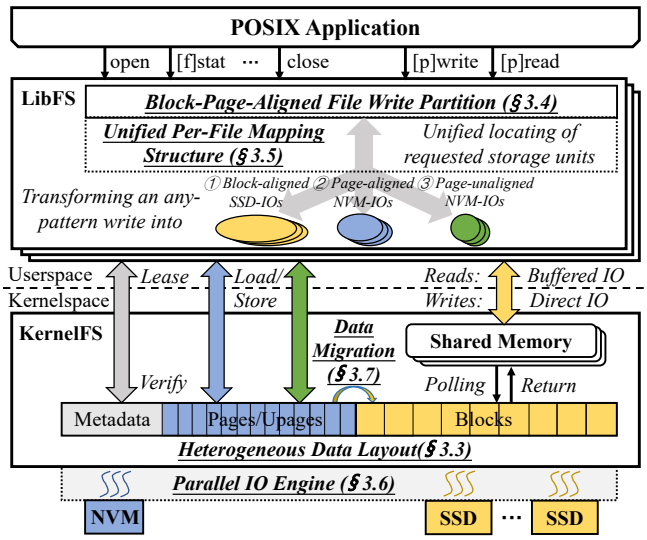


Figure 6: OrchFS overview.

- **C1:** How to effectively partition any-pattern writes with various offset and size and map the partitions into large SSD blocks and/or small NVM pages, while adaptively handling complex overwrites with data splitting and merging?
- **C2:** How to fast index all partitions across NVM and SSD?
- **C3:** How to effectively orchestrate transformed IOs for the triple paths of optimal SSD-write, conventional SSD-read, and memory-semantic NVM while ensuring consistency?

Additionally, OrchFS should also adaptively merge and migrate file data stored on NVM to SSD with low complexity, thus efficiently reducing file fragmentation and NVM usage.

3.2 OrchFS Overview

Figure 6 show an overview of OrchFS. To address **C1**, OrchFS first proposes an NVM-SSD heterogeneous data layout (§ 3.3) and three types of storage units to accommodate different-grained data. OrchFS further designs an alignment-prioritizing and fragmentation-minimizing write-partition (§ 3.4) to efficiently map any-pattern writes into three types of storage units. To overcome **C2**, OrchFS designs a unified per-file mapping structure HRtree (§ 3.5) to efficiently locate split file data across NVM pages and SSD blocks. To solve **C3**, OrchFS proposes an embedded parallel IO engine (§ 3.6) to handle transformed IOs in parallel via their respective paths by binding dedicated IO threads with exclusive address spaces and queues. Moreover, OrchFS inherits and extends the popular LibFS-KernelFS architecture to further reduce IO penalties and the common journaling mechanism to ensure crash consistency, as well as introducing an address-aligned shared memory to satisfy write-buffer-address alignment for the direct IO mode and SSD's access security (§ 4).

3.3 Heterogeneous Data Layout

OrchFS builds a data layout upon SSD and NVM, and defines three types of storage units to strategically store file data.

Data layout. OrchFS's data layout consists of a metadata area on NVM and file data area across NVM and SSD.

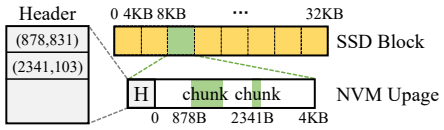


Figure 7: An example of using an NVM Upage to store two small updates of a logical page of an SSD block.

The metadata area defines superblock, bitmaps, inode table, HRtree (§ 3.5), directory, and journals, which are exclusively stored on NVM. The inode table and directory data structure adopt a linked list suitable for NVM [19,40,88]. The metadata updates are often small-sized, latency sensitive, and persistence required, thus preferring NVM's IO characteristics. The file data area includes an NVM page/Upage zone and an SSD block zone, with the latter storing most of the file data.

Three types of storage units. Both NVM and SSD have their respective preferred storage granularities. To minimize misalignment in SSD and NVM, OrchFS employs three types of storage units, i.e., SSD blocks, NVM pages, and NVM unaligned pages (Upages). By default, OrchFS sets the SSD block size to 32KB (two SSD-pages), which is an integral multiple of the NVM page size, i.e., 4KB, the same as most NVM file systems [19, 78, 88]. Setting larger SSD blocks means that more writes are judged as block-unaligned and are transformed into NVM-IOs. NVM Upage is designed to store the residual page-unaligned part of a file write, which is called a chunk, so that all unaligned data are store in Upages.

NVM Upage structure. By default, an NVM Upage consists of an NVM page for placing chunks and a 56B (64B - 8B, the reason in § 3.5) header for locating chunks within the page. The header contains 14 4B entries, each of which records the offset of a chunk within the page (2B) and the chunk's size (2B). Figure 7 shows an example of using an NVM Upage. The Upage stores two small updates of the SSD block's third logical page. The two small updates are stored as chunks whose offsets within the NVM Upage are 878B and 2341B, and whose sizes are 831B and 103B, respectively.

Merge of NVM chunks. When a new chunk is inserted to an NVM Upage, OrchFS first opportunistically merges it with existing address-overlapping chunks, then updates the Upage's header according to the merged result. Moreover, when all entries in a Upage's header are used up, OrchFS writes the Upage's data back to the corresponding SSD block and then reclaims this NVM Upage, thereby eliminating the file fragmentation within this logical page.

3.4 Block-Page-Aligned File Write Partition

OrchFS proposes block-page-aligned write-partition. That is, an any-pattern write is partitioned into block-aligned SSD-IOs, page-aligned NVM-IOs, and page-unaligned chunk-IOs sequentially, called the alignment-prioritizing (AP) policy. With this policy, the aligned parts of a write can be directly written into the corresponding blocks/pages without RMWs of either host or SSD, and then the residual chunks are strategically written to NVM. Further, a file should be mapped to as few storage units as possible to minimize fragmentation, called the fragmentation-minimizing (FM) policy.

There are two types of file writes: append writes and overwrites. An append write does not overlap with the already stored part of the file, allowing for a relatively simple generation of a set of block-IOs, page-IOs, and chunk-IOs. In contrast, an overwrite can overlap with various storage units, their combinations are especially complex. For this, following the FM policy, OrchFS partitions an any-pattern overwrite to a combination of three basic cases, i.e., overwrite to pages, Upages and blocks respectively, and handles them separately.

Algorithm 1 describes the write-partition process. For an incoming user write from the POSIX interface, OrchFS first identifies its type, i.e., append write, overwrite or both, according to the offset and size of the write (line 2 and 4-5).

Append writes. Following the AP policy, OrchFS transforms an append write into block-aligned SSD-IOs (line 22-23) and the remaining NVM-IOs (line 24-25). Following the FM policy, an unaligned chunk is directly appended to the last NVM page or a new page without the header. Moreover, when a file's last logical block is not completely filled, that is, this logical block is stored in NVM pages, OrchFS first partitions a part of the write to NVM-IOs to fill it (line 20-21), thereby simplifying index structures (§ 3.5) and data migration (§ 3.7).

Algorithm 1 Block-Page-Aligned File Write Partition

```
Input: Write Request Offset and Size: reqoffset, reqsize
 1: procedure WRITE_PARTITION(reqoffset, reqsize)
 2:     if reqoffset = filesize then
 3:         Append_write(reqoffset, reqsize)
 4:     else
 5:         if reqoffset + reqsize > filesize then
 6:             Append_write(filesize, reqoffset + reqsize - filesize)
 7:             reqsize ← filesize - reqoffset
 8:         subparts[] ← Split_write_to_3_Cases(reqoffset,reqsize)
 9:         for each subpart in subparts[] do
10:             if subpart ∈ NVM pages then # Case 1
11:                 Transform_to_Page_IOs(partoffset, partsize)
12:             else if subpart ∈ NVM Upages then # Case 2
13:                 Transform_to_Upage_IOs(partoffset, partsize)
14:             else Overwrite_to_SSDblocks(partoffset, partsize) # Case 3
15: procedure APPEND_WRITE(offset, size)
16:     # Split into 3 parts, with sizes: fillsize, midsize, remainsize in order
17:     fillsize ← SSDBlockSize - ( filesize mod SSDBlockSize)
18:     remainsize ← (size - fillsize) mod SSDBlockSize
19:     midsize ← size - fillsize - remainsize
20:     if fillsize ≠ 0 then
21:         Transform_to_Page_IOs(offset, fillsize)
22:     if midsize ≠ 0 then
23:         Transform_to_Block_IOs(offset + fillsize, midsize)
24:     if remainsize ≠ 0 then
25:         Transform_to_Page_IOs(offset + fillsize + midsize, remainsize)
26: procedure OVERWRITE_TO_SSDBLOCKS(offset, size)
27:     # Split into 3 parts, with sizes: headsize, midsize and tailsize in order
28:     headsize ← SSDBlockSize - ( filesize mod SSDBlockSize)
29:     tailsize ← (offset + size) mod SSDBlockSize
30:     midsize ← size - headsize - tailsize
31:     if headsize ≠ 0 then
32:         Transform_to_Page_and_Upage_IOs(offset, headsize)
33:     if midsize ≠ 0 then
34:         Transform_to_Block_IOs(offset + headsize, midsize)
35:         Reclaim_Upages(offset + headsize, midsize)
36:     if tailsize ≠ 0 then
37:         tailoffset = offset + headsize + midsize
38:         Transform_to_Page_and_Upage_IOs(tailoffset, tailsize)
```

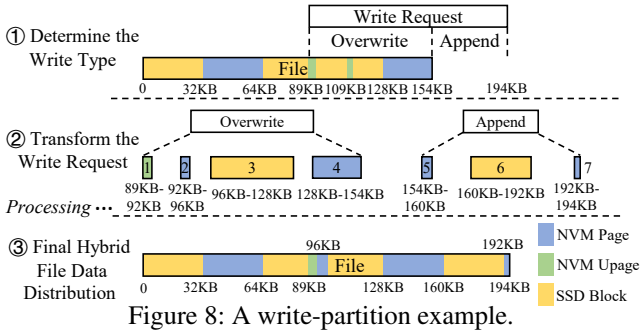


Figure 8: A write-partition example.

Overwrites. The write-partition transforms an overwrite into a combination of three basic cases and handles them separately. Case-1: Overwrite to NVM pages, following the FM policy, this case is transformed into page-IOs (line 10-11). Note that, unaligned page-IOs can be written to existing pages without headers. Case-2: Overwrite to NVM Upages. Following the FM policy, the write-partition transforms it into corresponding Upage-IOs (line 12-13) with the merge of NVM chunks (§ 3.3). Case-3: Overwrite to SSD blocks (line 14). Following the AP and FM policies, the write-partition first transforms the block-aligned part of the write to SSD-IOs while reclaiming the corresponding obsolete NVM Upages (line 33-35), and then the remaining parts of the write are transformed to aligned NVM page-IOs and unaligned Upage-IOs (line 31-32 and 36-38) with the merge of NVM chunks.

In this way, the write-partition adapts to any-pattern writes with a time complexity of O(1). Note that, akin to assembling *bio*s, the write-partition obtains the data distribution of requests during file mapping (§ 3.5) located on the critical path, without the need for additional queries.

Example. Figure 8 shows a write-partition example for an 154KB file. When receiving an 105KB write at the 89KB file offset, OrchFS splits the write into an overwrite with the offset of 89KB to 154KB and an append-write with the offset of 154KB to 194KB. According to the file data distribution, two block-aligned parts of 96KB-128KB within the overwrite and 160KB-192KB within the append-write are transformed to SSD-block IOs, while the NVM Upage corresponding to 109KB-110KB is reclaimed. The rest of the overwrite is transformed to NVM-page IOs and NVM-Upage IOs under Case-1 (128KB-154KB), Case-2 (89KB-92KB) and Case-3 (92KB-96KB), respectively. The rest of append-write is transformed to NVM-page IOs (154KB-160KB and 192KB-194KB). Finally, these 7 IOs are sent to the parallel IO engine to processed by NVM and SSD in parallel.

Reads. OrchFS passively transforms read requests to blocks/pages/Upages read-IOs based on the location of the requested data, and then sends them to the parallel IO engine.

3.5 Unified Per-File Mapping Structure

OrchFS's files can be distributed across NVM and SSD in three types of storage units, complicating file mapping. Therefore, OrchFS proposes a novel cache-friendly unified per-file mapping structure called Heterogeneous Radix tree, or HRtree, to efficiently map from file's logical offset to SSD blocks, NVM pages, and NVM Upages simultaneously. OrchFS avoids complex parallel searches or slow serial searches of two separate indexing structures for NVM and SSD, which are common schemes for existing hybrid NVM-SSD systems [27, 36, 45, 55, 76, 85].

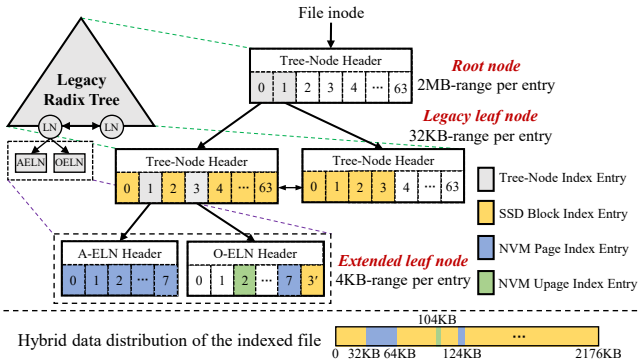


Figure 9: An HRtree structure example of a 2176KB file.

HRtree can be seen as merging two separate indexing structures of NVM and SSD by only adding a heterogeneous layer on the bottom of a legacy radix tree [39, 64]. As shown in Figure 9 (top left), HRtree's non-bottom part maps a file's logical offset to a 32KB logical block. A logical block is placed in either several NVM pages (Case-1) or an SSD block with small updates stored in NVM pages and/or Upages (if present) (Case-2). The heterogeneous layer contains two types of extended leaf nodes (ELNs): Append-write ELN (A-ELN) and Overwrite ELN (O-ELN), which are designed to locate the data of Case-1 and Case-2, respectively. The A-ELN consists of a 64B A-ELN header (storing node's metadata) and 8 8B index entries for locating NVM pages. The O-ELN consists of a 64B O-ELN header, an 8B index entry for locating an SSD block, and 8 64B index entries for locating NVM pages or Upages. Specifically, the 64B index entry is composed of 8B page index entry and 56B Upage header (§ 3.3), while when locating NVM pages, the Upage header is invalid.

Figure 9 shows the HRtree of a 2176KB file. Most data of this file are stored in SSD blocks. The file's logical block with offset=1 is stored entirely in NVM pages. The file logical block with offset=3 is stored in an SSD block, and the latest updates of the 8KB-12KB area and the 28KB-32KB area of the SSD block is placed in an Upage and a page respectively.

HRtree is kept in memory to speedup file mapping, while its updates are recorded as journals in NVM, which are applied conditionally (§ 4). The performance of HRtree is almost the same as that of a legacy radix tree, except for one additional memory-access when using an ELN. Moreover, HRtree connects leaf nodes similarly to B+tree [11, 35] to speedup range search, thereby improving the locating of large requests. OrchFS further deploys a readers-writer range lock [9, 56, 88] on HRtree to support fine-grained concurrent accesses to disjoint regions of a file while allowing concurrent reads on the same region. In addition, when migrating data from NVM to SSD,

HRtree only modifies the relevant ELNs without reconstructing, thus the HRtree design also simplifies data migration.

3.6 Parallel IO Engine

In OrchFS, file requests are transformed into SSD-IOs and NVM-IOs. To efficiently and transparently orchestrate these IOs with their respective data paths while ensuring data consistency, OrchFS presents a novel embedded parallel IO engine.

OrchFS deploys the parallel IO engine with a dedicated IO thread pool. By default, OrchFS sets 4 and 32 threads in the IO thread pool for NVM and SSD, respectively. Setting 4 threads for NVM is usually sufficient and suitable [79, 88], and NVM is only used to handle small IOs. The 32 IO threads for SSD are chosen to strike a balance between the load of each thread and system CPU usage, while making full use of SSD bandwidth. OrchFS further inherits OdinFS's spin-then-park strategy [88] for the IO thread pool to avoid wasting CPU cycles at idle and minimize the latency of naively parking and waking-up threads [34].

The address space of OrchFS is evenly divided among these IO threads in an interlaced manner. Specifically, each NVM-IO thread or SSD-IO thread is interlaced and bound to default 32KB aligned address-range in the NVM space or the SSD space. In this way, each large SSD-IO is transparently split into multiple aligned 32KB IOs, thereby further explicitly enabling the multi-threaded IO execution without the dependency on application threads. In comparison, traditional file systems passively map a request into one or more continuous physical addresses to generate memory-page aligned variable-sized IOs by expensive RMWs and without the proactive multi-threaded IO processing. Moreover, each IO thread has a ring queue buffer, and each new IO is sequentially added to the corresponding queue. Each IO thread uniquely executes read and write IOs sequentially within its corresponding exclusive address range, thus ensuring data consistency. Besides, OrchFS supports free configuration of the numbers of IO threads and interlaced binding granularity to adapt to NVMs and SSDs with different IO capabilities.

The parallel IO engine transparently uses the direct IO mode for SSD-write IOs, the buffered IO mode for SSD-read IOs, as well as the LibFS-KernelFS memory-semantic data path for NVM IOs. Linux kernel has provided a method to ensure the data consistency of direct IO writes and buffered IO reads (e.g., the generic_file_direct_write() function in Linux kernel 5.18.18 mm/filemap.c). Moreover, OrchFS introduces an address-aligned shared memory to satisfy write-buffer-address alignment for the direct IO mode, detailed in § 4.

3.7 Data Migration

To reduce NVM's usage and file fragmentation cross NVM and SSD, OrchFS opportunistically migrates NVM data to SSD, and frees up the used NVM pages/Upages. The reversed SSD-to-NVM migration is feasible but nonearning because SSD-reads benefit from page cache well, e.g., Figure 13.

Under high space utilization of NVM or idleness (optional), OrchFS enables the data migration. OrchFS preferentially migrates the logical-blocks that utilize more pages/Upages. For a migration operation, OrchFS first reads all pages/Upages of a logical-block and corresponding SSD-block (if any), and then writes them into a new block. During this migration, this logical-block is locked by HRtree's range lock. This fine-grained block-level (rather than coarse-grained file-level) migration reduces the impact on foreground writes. Besides, some NVM Upages are transparently written back and reclaimed after their headers are used up (§ 3.3).

Overall, OrchFS tends to store a vast majority of data to SSD directly, thus the data migration is not triggered frequently. In some extreme cases, e.g., all writes are small and placed on NVM, OrchFS would degenerate into a hierarchical NVM-SSD file system [36, 76, 85]. The data migration design can effectively handle such cases (e.g., § 5.5).

4 Implementation

We implement an OrchFS prototype from scratch while referring to NOVA [78], Strata [36], OdinFS [88], ArckFS [87] and F2FS [37]. The OrchFS prototype is available at https://github.com/YekangZhan/OrchFS. We modify Linux kernel's radix tree to implement HRtree. OrchFS inherits and extends the LibFS-KernelFS architecture to further reduce IO latency. OrchFS implements metadata security and NVM's data security based on Strata and ArckFS. OrchFS further introduces an address-aligned shared memory space for SSD-write data exchange between KernelFS and each LibFS. This ensures SSD's data security and satisfies write-buffer-address alignment for the direct IO mode simultaneously with less than 10% performance overhead in our evaluation. Additionally, OrchFS can also support the use of the buffered IO mode for asynchronous SSD-writes.

LibFS-KernelFS metadata updates. In OrchFS, KernelFS (i.e., the trusted entity) leases exclusive metadata resources and storage units to LibFSes. For the leased metadata updates, LibFSes record and store them to the metadata journals, and update them in memory. KernelFS verifies the legality and integrity of these journals, similar to ArckFS's "integrity verifier", but performs the verification asynchronously. If it passes, KernelFS performs this metadata update in NVM. In this way, KernelFS can maintain a global state.

Sharing. OrchFS adopts the conventional sharing model [36,87], which supports leases on files and regions of a file system namespace. The leases enable exclusive writes or shared reads for a specific file or a region of a namespace. When OrchFS reclaims a LibFS's write lease, it notifies KernelFS to verify and accept the LibFS's metadata updates.

Crash consistency. By default, OrchFS ensures atomic metadata-operations but not atomic data-operations, which provides the same consistency guarantees as EXT4 [6], NOVA relaxed mode [78], OdinFS [88] and ArckFS [87]. OrchFS can be easily enhanced to provide other consistency modes

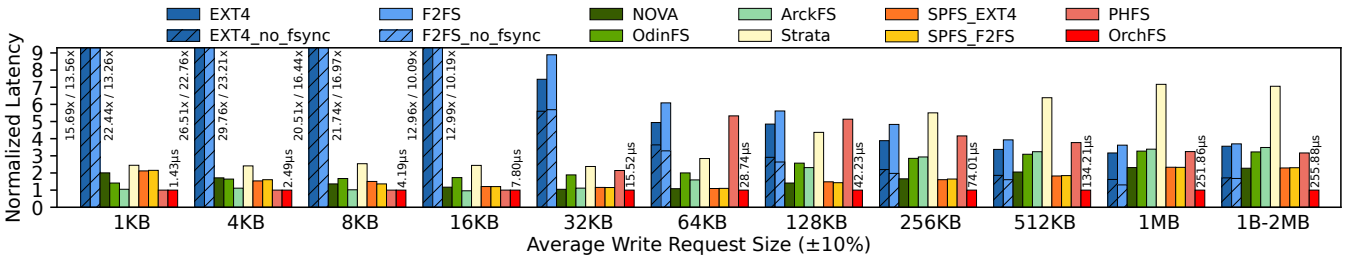


Figure 10: Average write latency of file systems under different write sizes.

Table 1: Experimental workloads.

Section	Workload Characteristics
§ 5.2.1	Single-threaded writes and reads under different IO sizes
§ 5.2.2	Multi-threaded writes under different access patterns
§ 5.3	Mixed metadata and data accesses with different proportions
§ 5.4.1	Small accesses, mixed writes and reads with different proportions
§ 5.4.2	Mixed small and large, writes and reads with different proportions

as needed. For example, using out-of-place updates for NVM pages/Upages and SSD blocks to ensure atomic writes. OrchFS ensures crash consistency by using conventional logical journaling [62, 68]. During crash recovery, OrchFS verifies and replays the operations recorded on the metadata journals serially to recover to a consistent state.

5 Evaluation

In this section, we evaluate the effectiveness of OrchFS under a wide variety of workloads, as shown in Table 1.

5.1 Experimental Setup

Environment. Our evaluation machine has two Intel Xeon Gold 6348 processors (2.60 GHz, 28 CPUs) and 256GB of DDR4 DRAM. By default, we use an 128GB Intel Optane PM [26] as NVM, and a PCIe4.0 3.84TB Samsung PM9A3 SSD [59] with 6.8GB/s read and 4.2GB/s write bandwidths. All experiments related to the PCIe5.0 SSD, i.e., 3.84TB Samsung PM1743 SSD [58] with 14GB/s read and 6GB/s write bandwidths, are conducted on a similar machine with 256GB of DDR5 DRAM. Note that, OrchFS aims to maximize SSD performance by using a small-sized auxiliary NVM, thus a larger and/or faster NVM does not alter the overall conclusions. OrchFS runs on Ubuntu 20.04 and Linux kernel 5.18.18. 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.

Baseline file systems. We compare OrchFS against three types of file systems. (1) SSD file systems: EXT4 [6] and F2FS [37]. (2) NVM file systems: NOVA [78], OdinFS [88] and ArckFS [87], for demonstrating the performance of OrchFS's NVM usage. (3) Hybrid NVM-SSD file systems (*HFS*s): Strata [36], SPFS [76], and PHFS. EXT4 and F2FS are mature and widely used production file systems. NOVA is a presentative in-kernel NVM file system. OdinFS and ArckFS are state-of-the-art in-kernel and userspace NVM file systems, respectively. Strata is a userspace hierarchical HFS. SPFS is a state-of-the-art hierarchical HFS stacked on EXT4 and F2FS for evaluation respectively. For a fair comparison, following UHS [89], which is implemented at block layer, as well as Intel DAOS [27], which is designed for distributed scenarios, *we implement their NVM-SSD parallel principle based on OrchFS, called PHFS*. PHFS stores all metadata in NVM, and sends small and large write requests to NVM and SSD respectively using 32KB as the small-large boundary.

Configuration. EXT4, NOVA, OdinFS, ArckFS, SPFS and PHFS are set to the same crash consistency guarantee as OrchFS. F2FS and Strata provide stricter write-atomicity with extra file-write overhead. Nevertheless, this does not impact the overall conclusions as explained later. EXT4 and F2FS run on SSD. NOVA, OdinFS and ArckFS run on NVM. Strata, SPFS, PHFS and OrchFS run cross NVM and SSD.

Limitation. Ziggurat [85] is not open-source. SplitFS [31] crashes frequently for unaligned writes, and its performance is inferior to that of ArckFS under aligned workloads [87].

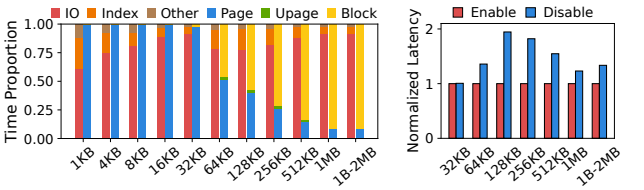
5.2 Microbenchmark

5.2.1 Single-Threaded Performance

We first evaluate single-threaded performance of OrchFS under diverse IO sizes. By default, we call `fsync()` after each write for data persistence. In this section, we also evaluate the performance of EXT4 and F2FS without `fsync()` calls.

Write performance. We first pre-fill a 10GB file with the same write-size range, clearing cache, then perform a 10GB random writes and measure the average write latency. As shown in Figure 10, OrchFS's performance is 0.96×-29.76× that of the baseline file systems. For small-size (<32KB) writes, OrchFS writes all data to NVM, exhibiting a similar performance to the NVM file systems and HFSs, and significantly higher than the SSD file systems.

For large-size (>64KB) writes, OrchFS outperforms the baseline file systems by 1.41×-7.16× in latency. The advantages of OrchFS stem from three aspects. Firstly, OrchFS keeps SSD working for an optimal write pattern, thereby avoiding the alignment cost, page cache overhead (§ 3.4), and fully exploiting SSD's high bandwidth (§ 3.6). 2) OrchFS adaptively utilizes NVM and SSD to handle each request in parallel (§ 3.6). 3) OrchFS provides efficient write-partition and file mapping (§ 3.5). In comparison, the SSD file systems and PHFS suffer from severe write inefficiency (§ 2.4), even without `fsync()` calls. Strata and SPFS use NVM to handle most writes, thus their performances are similar to that of the NVM file systems. The fact that OrchFS has similar performance in the 1MB case and the 1B-2MB case demonstrates



(a) Write time breakdown (size ± 10%) (b) Parallel IO engine

Figure 11: Write time breakdown (a) and sensitivity study of enabling and disabling the parallel IO engine (b).

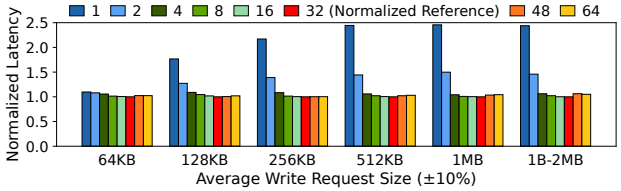


Figure 12: Sensitivity study of the number of SSD-IO threads in the parallel IO engine.

the load-adaptability of OrchFS's write-partition (§ 3.4).

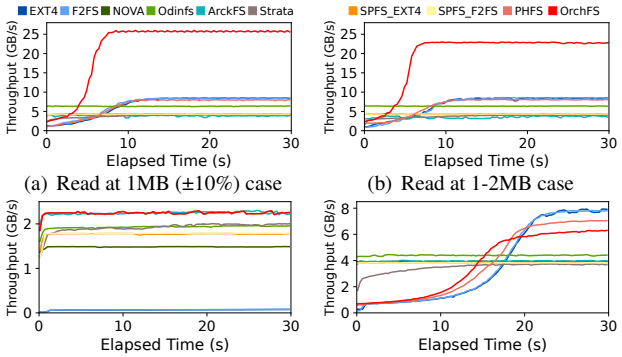
For the 32KB and 64KB writes, about 2.4% and 48.1% of request-data in OrchFS are written to SSD respectively. Hence, OrchFS gains relatively less advantage than the NVM file systems, Strata and SPFS, but still significantly outperforms the SSD file systems (4.94×-8.89×). PHFS under the 32KB and 64KB cases sends 1/2 of and all writes to SSD respectively, rendering its 32KB performance in between those of the NVM and SSD file systems and its 64KB performance similar to that of the SSD file systems.

Write breakdown and sensitivity study of parallel IO engine. To better understand the performance gains by OrchFS, we breakdown the write time (left bars) and the IO time (right bars), as shown in Figure 11(a). Note that, we stack the IO times of NVM and SSD in the same bar, but NVM-IOs and SSD-IOs are actually handled in parallel. For small (<32KB) writes, NVM handles all write-IOs. For medium to large writes, the write-IOs are handled by both NVM and SSD. The larger the write size, the smaller the proportion of NVM-IOs. This demonstrates that OrchFS does not rely on NVM performance for medium to large writes.

Then, we analyze the efficacy of parallel IO engine. We first measure the write latency with/without the parallel IO engine (i.e., enabling parallel NVM-SSD IO handling and SSD's multi-threaded IO execution or not), as shown in Figure 11(b). OrchFS with parallel IO engine outperforms its counterpart without the engine in IO latency by up to 48.6%. This demonstrates that the parallel IO engine can effectively improve the write latency for various write sizes without side effects. We further study the impact of the number of IO threads in the parallel IO engine on the write latency. Considering that the NVM under 4 IO-threads behaves best [79, 88], we focus on the number of SSD-IO threads. Figure 12 shows that the default setting of 32 SSD-IO threads has the best write performance. Moreover, although 8 SSD-IO threads can almost maximize SSD performance, the default 32 threads

Table 2: OrchFS storage space usage breakdown.

IO Size (±10%)	1,4,8,16KB	32KB	64KB	128KB	256KB	512KB	1MB	1B-2MB
	Space occupation after the 10GB pre-fill (i.e., append writes) (GB)							
NVM Page	10	9.76	5.19	2.50	1.25	0.63	0.31	0.32
NVM Upage	0	0	0	0	0	0	0	0
SSD Block	0	0.24	4.81	7.50	8.75	9.37	9.69	9.68
	Space occupation after the 10GB random writes on the above filled file (GB)							
NVM Page	10	9.89	7.06	4.19	2.34	1.25	0.63	0.64
NVM Upage	0	0.03	0.47	0.42	0.27	0.15	0.08	0.08
SSD Block	0	0.24	4.81	7.50	8.75	9.37	9.69	9.68



(a) Read at 1MB (±10%) case (b) Read at 1-2MB case

(c) Read at 4KB (±10%) case (d) Read at 1B-128KB case

Figure 13: Read throughputs of file systems over time.

can reduce the load of each thread (i.e., the queue depth of each thread § 3.6).

Space usage breakdown. To better understand file data distribution in OrchFS, we measure the usages of its three types of storage units after completing the above pre-fill (i.e., append writes) and random writes respectively, as shown in Table 2. There are two main takeaways. (1) The larger the write request size, the more data are stored in SSD. In the 1MB random write case, OrchFS only writes about 4.7% data to NVM, but OrchFS improves the write latency by 68.4% and 72.4% over EXT4 and F2FS that solely use SSD, respectively. This demonstrates that OrchFS effectively maximizes SSD performance by only using a small amount of NVM. (2) The space usages under the 1B-2MB case and the 1MB (±10%) case are similar, which demonstrates the adaptability of OrchFS's storage units to various write sizes.

Reads. Next, we conduct four experiments to assess OrchFS's read performance. We first fill a 10GB file with write-size of 1MB (±10%), 1B-2MB, 4KB (±10%) and 1B-128KB respectively, clearing cache, and then perform random reads with the corresponding size-ranges.

The results are shown in Figure 13. The NVM file systems without page cache, and Strata and SPFS (almost exclusively using NVM) have relatively stable throughputs over time. The SSD file systems have a slow start in throughput due to reading from SSD, and gradually gain speed over time as more data are cached by page cache, eventually reaching and maintaining a peak with a stable page cache hit ratio.

For OrchFS and PHFS, approximately 3.1% and 0%, 3.2% and 1.6%, 100% and 100%, and 52% and 25% of the file data reside in NVM for the four cases respectively. In the 1MB (±10%) case, OrchFS quickly reaches its peak throughput, which is up to 6.79× higher than the baseline file systems. Because OrchFS largely reads multiple 32KB-aligned regions

of SSD and NVM in parallel, only the head and tail of each request may result in unaligned reads that are partially served by NVM. PHFS's performance is similar to those of the SSD file systems, because PHFS only uses SSD in the 1MB case.

In the 1B-2MB case, OrchFS exhibits a similar trend to and a slightly lower peak throughput than the 1MB (±10%) case, due to slightly more unaligned reads. Nevertheless, OrchFS's peak throughput is up to 6.33× that of the baseline file systems. PHFS uses NVM to serve small reads, thus exhibiting a higher initial throughput than the SSD file systems.

In the 4KB (±10%) case, both OrchFS and PHFS use NVM only, their behaviors are similar to those of the NVM file systems, Strata and SPFS. The SSD file systems exhibit slow performance growth due to slow small and unaligned reads.

The 1B-128KB case aims to present a special scenario, i.e., in OrchFS, half of the file data is on NVM, and the other half is on SSD. Compared to the SSD file system, OrchFS has a faster growth rate in throughput, but bounded by the NVM's read bandwidth, thus exhibiting a slightly lower peak throughput. In PHFS, 25% of the file data is on NVM, thus its throughput growth rate and peak throughput fall between those of OrchFS and the SSD file systems. Certainly, OrchFS can use page cache for NVM-side reads [73] to improve its peak throughput.

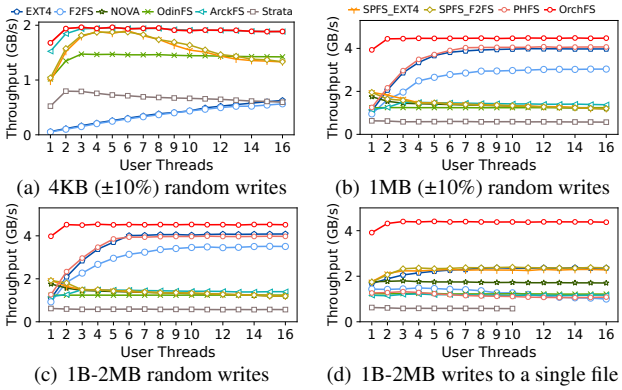
Overall, these experiments demonstrate that OrchFS effectively exploits SSD's potentials by only using a small-sized NVM. The block-aligned write-partition not only makes writes efficient but also improves read performance.

5.2.2 Multi-Threaded Performance

We evaluate the multi-threaded performance of OrchFS to demonstrate its effectiveness under concurrent accesses.

Concurrent access to multiple files. We conduct random overwrite tests with three different write-size ranges: 4KB (±10%), 1MB (±10%), and 1B-2MB. We pre-fill 16 5GB files with the same write-size range. Each user thread accesses only one file. We measure the aggregate throughput, as shown in Figure 14(a)(b)(c). In the 4KB (±10%) case, OrchFS's performance is comparable to that of SPFS and the NVM file systems (1.00×-1.72×). In the 1MB (±10%) and 1B-2MB cases, OrchFS reaches the peak throughput at 2 user threads and remains stable as the number of threads scales. In comparison, PHFS and the SSD file systems reach peak throughput at more than 6 user threads. SPFS and Strata almost use NVM only, hence their performance is similar to the NVM file systems. Strata performs poorly in all cases, due to its atomic writes and limited concurrency.

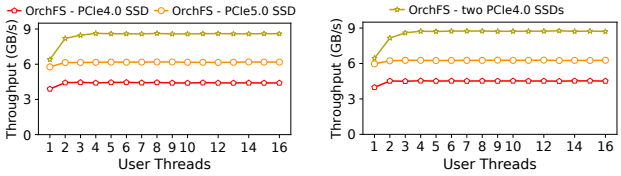
Concurrent access to a single file. We further evaluate the concurrency of OrchFS on a single file. We pre-fill a 10GB file with write-size ranging from 1B to 2MB, and conduct random write tests with the same write size range. Each user-thread writes 3GB data. Figure 14(d) shows that OrchFS outperforms the baseline file systems in aggregate throughput by 1.86×-7.84×. At 2 user-threads, OrchFS saturates the SSD's



(a) 4KB (±10%) random writes (b) 1MB (±10%) random writes

(c) 1B-2MB random writes (d) 1B-2MB writes to a single file

Figure 14: Write throughputs under multiple user-threads.



(a) 1B-2MB append writes (b) 1B-2MB random writes

Figure 15: Write throughputs under different SSD setups.

bandwidth and maintains stable performance with increasing threads. PHFS and the SSD file systems suffer from slow concurrent `fsync()` on a single file [50, 53], while OrchFS avoids this due to the direct IO mode for SSD writes (§ 2.4.2).

Sensitivity analysis of different SSD setups. We further evaluate the performance of OrchFS under three SSD setups, namely, one default PCIe4.0 SSD, one PCIe5.0 SSD with 6GB/s write bandwidth, and two default PCIe4.0 SSDs with a RAID0 configuration [52]. Each user thread accesses one file using a write-size range of 1B-2MB. We measure aggregate throughputs of append writes and random writes respectively. Figure 15 shows that OrchFS reaches its peak throughput at 2 and 3 user-threads for the one PCIe4.0/PCIe5.0 SSD setups and the two PCIe4.0 SSDs. Moreover, the performance of append and random writes of OrchFS are similar, validating the effectiveness of OrchFS's write-partition (§ 3.4).

Overall, OrchFS supports concurrent accesses effectively.

5.3 Macrobenchmark

In this section, we use Filebench [69] with three representative workloads of Fileserver, Webproxy and Varmail to evaluate the overall performance of OrchFS under mix reads and writes of metadata and file data. Table 3 summarizes the characteristics of these three workloads. We persist writes immediately. Strata is unable to run Filebench.

Single thread. Figure 16(a) shows that, under a single thread, OrchFS outperforms the baseline file systems by 2.33×-10.25×, 1.81×-7.44×, and 1.04×-13.35× in throughput for Fileserver, Webproxy and Varmail respectively. For metadata operations, the performance of OrchFS and PHFS is comparable to that of the NVM file systems, and superior to that of the SSD file systems. For reads and writes, OrchFS outperforms the baseline file systems, similar to § 5.2.1.

Table 3: Filebench workload characteristics.

Workload	#Files	Avg. File size	Avg. IO size (r/w)	Threads	R/W
Fileserver	10K	256KB	1MB/256KB	1/16	1:2
Webproxy	10K	2MB	1MB/256KB	1/16	5:1
Varmail	10K	64KB	1MB/64KB	1/16	1:1

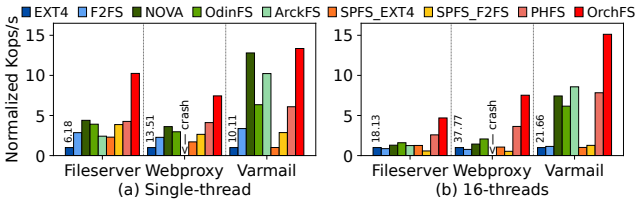


Figure 16: Normalized throughput of Filebench.

Therefore, for write-dominated Fileserver and read-dominated Webproxy, OrchFS's throughput is the highest. For metadata-dominated workload Varmail, OrchFS exhibits 3.96×-13.35× higher throughput than that of the SSD file systems and SPFS. SPFS performs poorly because it is a stacked file system with more complex metadata updates. Compared to the NVM file systems, the advantage of OrchFS comes from a small number of large reads and medium-size writes.

16 threads. Figure 16(b) further shows the results under 16 threads. OrchFS outperforms the baseline file systems by 1.76×-15.12× in throughputs. The SSD file systems still perform poorly due to slow metadata operations and write inefficiency. PHFS utilizes NVM to handle all metadata operations, while using SSD for reads and writes akin to the SSD file systems, and its performance is significantly inferior to OrchFS in all cases. This demonstrates that the parallel HFS design cannot fully utilize SSD performance directly. The NVM file systems and SPFS also underperform due to limited bandwidth and concurrency of the NVM. Using faster and/or more NVMs can improve their performance at the cost of higher price. In comparison, the majority of OrchFS's performance attributes to SSD.

Overall, OrchFS exhibits high performance under mixed metadata and data accesses.

5.4 Real-World Applications

In this section, we evaluate the efficiency and adaptability of OrchFS under two real-world representative applications: key-value store [23], and large-scale graph processing [90].

5.4.1 YCSB + LevelDB

We use LevelDB [23] as a representative key-value store. The Yahoo! Cloud Serving Benchmark (YCSB) [14] is a common benchmark for database applications, as a workload generator. Table 4 summarizes the characteristics of the YCSB workloads. We set the SSTable size to 64MB to follow the recommended configuration [20]. We configure the key-value operations of LevelDB to be synchronous, so that writes are immediately persisted. We issue 5 million operations with a value size of 1KB. ArckFS is unable to run YCSB + LevelDB.

The results of the overall throughput are shown in Figure 17. In these cases, OrchFS, SPFS and PHFS use NVM to handle

Table 4: YCSB workload characteristics.

Load	Write only: 100% insert
RunA	Update heavy: 50%/50% read/write mix
RunB	Read mostly: 95%/5% read/write mix
RunC	Read only: 100% read
RunD	Read latest: new records are inserted, and the most recently inserted records are read
RunE	Short ranges: short ranges of records are queried, instead of individual records
RunF	Read-modify-write: read a record, modify it, and write back the changes

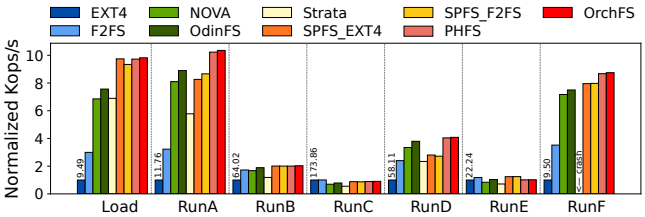


Figure 17: Normalized throughput of YCSB+LevelDB.

small writes (and corresponding reads) and metadata accesses, while using SSD to process a small amount of large and asynchronous compaction writes. Therefore, OrchFS, SPFS and PHFS have similar performance.

For Load with intensive and small write, OrchFS has a greater advantage over the SSD file systems (3.28×-9.82×). Compared to the NVM file systems, OrchFS's advantage (1.3×-1.43×) stems from its use of SSD to handle large compaction writes, saving the NVM bandwidth. For RunA, RunB, RunD and RunF, which have different proportions of writes, OrchFS significantly outperforms the SSD file systems and slightly outperforms the NVM file systems in throughput.

For read-only RunC and RunE, OrchFS's performance is competitive with the NVM file systems (0.99×-1.64×). Since more than 95% data of OrchFS are located in NVM and are not cached, OrchFS's read performance is slightly lower than the SSD file systems effectively benefiting from page cache, similar to Figure 13(d). OrchFS performs better than SPFS in RunC but is slightly inferior to SPFS in range-read-based RunE, due to SPFS's read-optimized extent hashing.

5.4.2 Large-Scale Graph Processing

GridGraph [90] is a system for processing large-scale graphs on a single machine. To further evaluate OrchFS's load adaptability, we employ GridGraph to execute the Pagerank [21] algorithm with 8GB dedicated memory (emulating scenarios of large-scale graph processing) under the Livejournal [66], Twitter [32], and Friendster [65] datasets respectively, and run 20 iterations on each graph. Table 5 summarizes the workload characteristics. We persist writes immediately.

We measure the execution time under the three datasets respectively, as shown in Figure 18, OrchFS outperforms the baseline file systems by 1.69×-3.20×. The SSD file systems and PHFS suffer from the write-inefficiency. The NVM file systems and SPFS rely on expensive NVM to process requests but are limited by the performance of NVM. This suggests that under various mixtures of reads and writes with diverse unaligned request sizes and offsets, OrchFS can always handle them efficiently, thereby accelerating the graph processing and the scenarios with similar access behaviors, such as cloud computing [74] and scientific computing [54].

Table 5: Workload characteristics of graph processing.

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

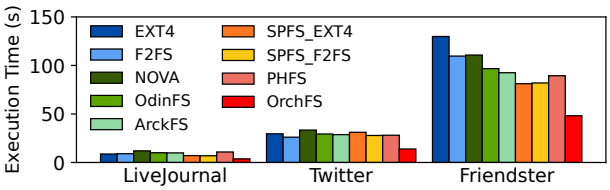


Figure 18: Execution time of graph processing.

Overall, OrchFS exhibits strong load-adaptability for emerging real-world applications.

5.5 Analysis of File Fragmentation and Aging

In SSD-side, the allocation unit is an SSD-block, e.g., 32KB, and OrchFS always places file data in an SSD-block aligned manner. The SSD-blocks of a file are never further fragmented, thus relieving potential aging of SSD-side [29]. In NVM-side, OrchFS allocates NVM pages and smaller chunks in Upages for block-unaligned writes, causing potential data fragmentation over aging. However, the fragmented chunks can be merged in Upages (§ 3.3). Furthermore, the NVM Upages/pages can be strategically migrated and merged into the SSD blocks, thus mitigating file fragmentation in NVM.

To better understand data fragmentation of OrchFS, we first disable the data migration of OrchFS and perform continuous unaligned random writes with different sizes to a 10GB file, which is initially stored on SSD. We measure the percentage of the number of NVM Upages relative to the total number of logical pages in the file, reflecting the degree of subpage-level data fragmentation. Figure 19 shows that, for all cases, as the amount of written data increases, the percentage of Upages first increases and then decreases, because the chunks within an Upage are gradually merged into a full-page. The 1KB, 4KB, and 16KB cases exhibit a high percentage of Upages, because all writes are stored in NVM. Moreover, the larger the write size, the less the Upages, because more data are written to SSD directly. When the data migration is enabled, OrchFS gradually migrates all NVM Upages/pages to SSDs. For example, for the extreme 1KB case file, OrchFS migrates all its NVM Upages/pages to the SSD within 12 seconds. Additionally, OrchFS can adopt WineFS's anti-aging approach [30] to further reduce NVM-side page-level fragmentation.

6 Related Work

This section discusses the related work concerning file systems, multiple IO paths, and alignment optimizations.

SSD file systems have been undergoing continuous development and evolution [4, 37, 48, 57, 68]. The latest improvements focus on the scalability, e.g., CJFS [50] presents concurrent journaling for EXT4 [48], SpanFS [33] and MAX [44] propose scalability optimization based on F2FS [37]. They all suffer from the write inefficiency on high-bandwidth SSDs.

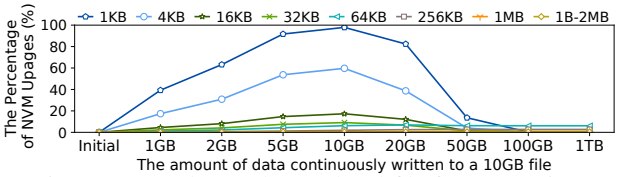


Figure 19: OrchFS subpage-level file fragmentation.

NVM file systems are designed to fully exploit NVM performance. In-kernel NVM file systems [12, 19, 30, 78, 88] bypass page cache and allow applications to directly access file data stored on NVM. Userspace NVM file systems [9, 10, 17, 31, 40, 86, 87] further bypass the kernel and access NVM in userspace to mitigate IO penalties. However, the relatively high per capacity price of NVMs limits the widespread deployment of such systems.

Hybrid NVM-SSD systems either use the upper NVMs to endure most loads [8, 36, 41, 45, 55, 76, 85], thus underutilizing the underlying SSDs, or leverage NVM and SSD to handle small and large writes respectively [27, 89] but ignore the write inefficiency of SSDs, especially high alignment costs.

Multiple IO paths. Some distributed file systems (DFSs) use the buffered IO mode and the direct IO mode for SSDs based on a fixed or dynamic IO-size threshold [13, 25, 42, 54, 61, 72]. Most hybrid NVM-SSD systems use the memory-semantic path for NVM and the buffered IO mode for SSD, and some works [27, 36] use SPDK [81] for SSD. They do not consider high alignment costs and write-partition, while using the expensive RMW process (§ 2.4.1) to align writes.

Alignment optimizations. Some works [5, 22, 24, 63, 83, 84] have also observed high alignment costs of SSDs. To reduce it, NVStore [63] uses page cache to merge unaligned writes. Re-aligning [5] improves FTL to remap unaligned writes to a single SSD-page. WAFLASH [24] proposes a write-aligned FLASH drive. iBridge [84] leverages SSD to serve stripe-unaligned writes of DFSs. They are orthogonal to OrchFS. Besides, WineFS [30] proposes an alignment-aware NVM allocator to resist the effects of aging for NVM file systems. OrchFS focuses on the SSD-side alignment optimization and can inherit WineFS's designs to enhance the NVM-side performance and anti-aging capabilities.

7 Conclusion

This paper identifies and analyzes the root causes of the write inefficiency of file systems on high-bandwidth SSDs, and proposes OrchFS to maximize SSD performance by transforming requests into device-preferred SSD-IOs and NVM-IOs. Experimental results demonstrate the efficacy of OrchFS.

Acknowledgement

We thank our shepherd Saurabh Kadekodi and the anonymous reviewers for their insightful comments. This work was supported by NSFC (No.62172175), National Key Research and Development Program of China (No.2024YFB4505105, No.2022YFB2804302), and Key Research and Development Project of Hubei (No.2022BAA042).

References

[1] Imran Ahmed, Misbah Ahmad, Awais Ahmad, and Gwanggil Jeon. Iot-based crowd monitoring system: Using ssd with transfer learning. Computers & Electrical Engineering, 93:107226, 2021.

[2] Dmytro Apalkov, Alexey Khvalkovskiy, Steven Watts, Vladimir Nikitin, Xueti Tang, Daniel Lottis, Kiseok Moon, Xiao Luo, Eugene Chen, Adrian Ong, et al. Spin-transfer torque magnetic random access memory (stt-mram). ACM Journal on Emerging Technologies in Computing Systems (JETC), 9(2):1–35, 2013.

[3] AXBOE. Fio: Flexible i/o tester. 2020. https://github.com/axboe/fio.

[4] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. The zettabyte file system. In Proc. of the 2nd Usenix Conference on File and Storage Technologies, volume 215, 2003.

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

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

[7] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In 18th USENIX Conference on File and Storage Technologies (FAST 20), pages 209–223, 2020.

[8] Youmin Chen, Youyou Lu, Pei Chen, and Jiwu Shu. Efficient and consistent nvmm cache for ssd-based file system. IEEE Transactions on Computers, 68(8):1147–1158, 2018.

[9] Youmin Chen, Youyou Lu, Bohong Zhu, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Jiwu Shu. Scalable persistent memory file system with kernel-userspace collaboration. In 19th USENIX Conference on File and Storage Technologies (FAST 21), pages 81–95, 2021.

[10] Jungsik Choi, Jaewan Hong, Youngjin Kwon, and Hwansoo Han. Libnvmmio: Reconstructing software io path with failure-atomic memory-mapped interface. In 2020 USENIX Annual Technical Conference (USENIX ATC 20), pages 1–16, 2020.

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

[12] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, pages 133–146, 2009.

[13] Orangefs configuration file. Orangefs documentation. https://docs.orangefs.com/configuration/admin_ofs_configuration_file/, 2023.

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

[15] Jonathan Corbet. Buffered i/o without page-cache thrashing. https://lwn.net/Articles/806980/, 2024.

[16] Jinhua Cui, Zhimin Zeng, Jianhang Huang, Weiqi Yuan, and Laurence T Yang. Improving 3d nand ssd read performance by parallelizing read-retry. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2022.

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

[18] Xiangyu Dong, Naveen Muralimanohar, Norm Jouppi, Richard Kaufmann, and Yuan Xie. Leveraging 3d pcram technologies to reduce checkpoint overhead for future exascale systems. In Proceedings of the conference on high performance computing networking, storage and analysis, pages 1–12, 2009.

[19] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In Proceedings of the Ninth European Conference on Computer Systems, pages 1–15, 2014.

[20] Facebook. Rocksdb tuning guide. 2023. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide.

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

[22] Congming Gao, Liang Shi, Kai Liu, Chun Jason Xue, Jun Yang, and Youtao Zhang. Boosting the performance of ssds via fully exploiting the plane level parallelism. IEEE Transactions on Parallel and Distributed Systems, 31(9):2185–2200, 2020.

[23] Sanjay Ghemawat and Jeff Dean. Leveldb is a fast key-value storage library written at google that provides an ordered mapping from string keys to string values, 2014.

[24] Shuibing He, Matthew Myers, Xuehao Duan, Keegan Sanchez, and Xuechen Zhang. Waflash: Taming unaligned writes in solid-state disks. In 2022 IEEE International Conference on Networking, Architecture and Storage (NAS), pages 1–8. IEEE, 2022.

[25] Frank Herold. An introduction to beegfs. https://www.beegfs.io/docs/whitepapers/Introduction_to_BeeGFS_by_ThinkParQ.pdf, 2014.

[26] Intel. Intel optane dc persistent memory. 2021. https://www.intel.com/content/www/us/en/architecture-and-technology.

[27] Intel. Distributed asynchronous object storage (daos). https://docs.daos.io/v2.7/, 2024.

[28] Intel. Distributed asynchronous object storage (daos). https://github.com/daos-stack/daos/tree/master, 2024.

[29] Yuhun Jun, Shinhyun Park, Jeong-Uk Kang, Sang-Hoon Kim, and Euiseong Seo. We ain't afraid of no file fragmentation: Causes and prevention of its performance impact on modern flash SSDs. In 22nd USENIX Conference on File and Storage Technologies (FAST 24), pages 193–208, Santa Clara, CA, February 2024. USENIX Association.

[30] Rohan Kadekodi, Saurabh Kadekodi, Soujanya Ponnapalli, Harshad Shirwadkar, Gregory R Ganger, Aasheesh Kolli, and Vijay Chidambaram. Winefs: a hugepage-aware file system for persistent memory that ages gracefully. In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, pages 804–818, 2021.

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

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

[33] 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 2015 USENIX Annual Technical Conference (USENIX ATC 15), pages 249–261, 2015.

[34] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Scalable numa-aware blocking synchronization primitives. In 2017 USENIX Annual Technical Conference (USENIX ATC 17), pages 603–615, 2017.

[35] Wook-Hee Kim, R Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. Pactree: A high performance persistent range index using pac guidelines. In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, pages 424–439, 2021.

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

[37] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2fs: A new file system for flash storage. In 13th USENIX Conference on File and Storage Technologies (FAST 15), pages 273–286, 2015.

[38] 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 2019 USENIX Annual Technical Conference (USENIX ATC 19), pages 603–616, 2019.

[39] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In 2013 IEEE 29th International Conference on Data Engineering (ICDE), pages 38–49. IEEE, 2013.

[40] 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 20th USENIX Conference on File and Storage Technologies (FAST 22), pages 35–50, 2022.

[41] Wenjie Li, Dejun Jiang, Jin Xiong, and Yungang Bao. Hilsm: an lsm-based key-value store for hybrid nvm-ssd storage systems. In Proceedings of the 17th ACM International Conference on Computing Frontiers, pages 208–216, 2020.

[42] Xiuqiao Li, Limin Xiao, Meikang Qiu, Bin Dong, and Li Ruan. Enabling dynamic file i/o path selection at runtime for parallel file system. The Journal of Supercomputing, 68:996–1021, 2014.

[43] Yu Liang, Jinheng Li, Rachata Ausavarungnirun, Riwei Pan, Liang Shi, Tei-Wei Kuo, and Chun Jason Xue. Acclaim: Adaptive memory reclaim to improve user experience in android systems. In 2020 USENIX Annual Technical Conference (USENIX ATC 20), pages 897–910, 2020.

[44] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. Max: A multicore-accelerated file system for flash storage. In 2021 USENIX Annual Technical Conference (USENIX ATC 21), pages 877–891, 2021.

[45] Zhen Lin, Lingfeng Xiang, Jia Rao, and Hui Lu. P2cache: Exploring tiered memory for in-kernel file systems caching. In 2023 USENIX Annual Technical Conference (USENIX ATC 23), pages 801–815, 2023.

[46] Chun-Yi Liu, Jagadish B Kotra, Myoungsoo Jung, Mahmut T Kandemir, and Chita R Das. Soml read: Rethinking the read operation granularity of 3d nand ssds. In ASPLOS, pages 955–969, 2019.

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

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

[49] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. Checkfreq: Frequent, fine-grained dnn checkpointing. In 19th USENIX Conference on File and Storage Technologies (FAST 21), pages 203–216, 2021.

[50] Joontaek Oh, Seung Won Yoo, Hojin Nam, Changwoo Min, and Youjip Won. Cjfs: Concurrent journaling for better scalability. In 21st USENIX Conference on File and Storage Technologies (FAST 23), pages 167–182, 2023.

[51] Yoshihiro Oyama, Shun Ishiguro, Jun Murakami, Shin Sasaki, Ryo Matsumiya, and Osamu Tatebe. Reduction of operating system jitter caused by page reclaim. In Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers, pages 1–8, 2014.

[52] David A. Patterson, Garth A. Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 1-3, 1988, pages 109–116, 1988.

[53] Kiet Tuan Pham, Seokjoo Cho, Sangjin Lee, Lan Anh Nguyen, Hyeongi Yeo, Ipoom Jeong, Sungjin Lee, Nam Sung Kim, and Yongseok Son. Scalecache: A scalable page cache for multiple solid-state drives. In Proceedings of the Nineteenth European Conference on Computer Systems, pages 641–656, 2024.

[54] Yingjin Qian, Marc-André Vef, Patrick Farrell, Andreas Dilger, Xi Li, Shuichi Ihara, Yinjin Fu, Wei Xue, and André Brinkmann. Combining buffered i/o and direct i/o in distributed file systems. In 22nd USENIX Conference on File and Storage Technologies (FAST 24), pages 17–33, 2024.

[55] Sheng Qiu and AL Narasimha Reddy. 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), pages 1–5. IEEE, 2013.

[56] Yujie Ren, Changwoo Min, and Sudarsun Kannan. Crossfs: A cross-layered direct-access file system. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pages 137–154, 2020.

[57] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. ACM Transactions on Storage (TOS), 9(3):1–32, 2013.

[58] Samsung. Samsung pm1743 ssd. https://semiconductor.samsung.com/ssd/enterprise-ssd/pm1743/, 2024.

[59] Samsung. Samsung pm9a3 ssd. 2024. https://semiconductor.samsung.com/ssd/datacenter-ssd/pm9a3/.

[60] Samsung. Samsung ssd. https://semiconductor.samsung.com/ssd/, 2024.

[61] Frank Schmuck and Roger Haskin. Gpfs: A shared-disk file system for large computing clusters. In Conference on file and storage technologies (FAST 02), 2002.

[62] Harshad Shirwadkar, Saurabh Kadekodi, and Theodore Tso. FastCommit: resource-efficient, performant and cost-effective file system journaling. In 2024 USENIX Annual Technical Conference (USENIX ATC 24), pages 157–171, Santa Clara, CA, July 2024. USENIX Association.

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

[64] Keith Sklower. A tree-based packet routing table for berkeley unix. In USENIX Winter, volume 1991, pages 93–99, 1991.

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

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

[67] Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. The missing memristor found. nature, 453(7191):80–83, 2008.

[68] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the xfs file system. In USENIX Annual Technical Conference, volume 15, 1996.

[69] Vasily Tarasov. Filebench-a model based file system workload generator. https://github.com/filebench/filebench, 2018.

[70] Arash Tavakkol, Mohammad Sadrosadati, Saugata Ghose, Jeremie Kim, Yixin Luo, Yaohua Wang, Nika Mansouri Ghiasi, Lois Orosa, Juan Gómez-Luna, and Onur Mutlu. Flin: Enabling fairness and enhancing performance in modern nvme solid state drives. In 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), pages 397–410. IEEE, 2018.

[71] Stephen R Walli. The posix family of standards. StandardView, 3(1):11–17, 1995.

[72] Feiyi Wang, Sarp Oral, Saurabh Gupta, Devesh Tiwari, and Sudharshan S Vazhkudai. Improving large-scale storage system performance via topology-aware and balanced data placement. In 2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS), pages 656–663. IEEE, 2014.

[73] Guoyu Wang, Xilong Che, Haoyang Wei, Shuo Chen, Puyi He, and Juncheng Hu. Nvpc: A transparent nvm page cache. arXiv preprint arXiv:2408.02911, 2024.

[74] Shucheng Wang, Ziyi Lu, Qiang Cao, Hong Jiang, Jie Yao, Yuanyuan Dong, and Puyuan Yang. Bcw: Buffer-controlled writes to hdds for ssd-hdd hybrid storage server. In 18th USENIX Conference on File and Storage Technologies (FAST 20), pages 253–266, 2020.

[75] Wiki. Pci express. https://semiconductor.samsung.com/ssd/, 2024.

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

[77] Kan Wu, Zhihan Guo, Guanzhou Hu, Kaiwei Tu, Ramnatthan Alagappan, Rathijit Sen, Kwanghyun Park, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. The storage hierarchy is not a hierarchy: Optimizing caching on modern storage devices with orthus. In 19th USENIX Conference on File and Storage Technologies (FAST 21), pages 307–323, 2021.

[78] Jian Xu and Steven Swanson. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In 14th USENIX Conference on File and Storage Technologies (FAST 16), pages 323–338, 2016.

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

[80] Shao-Peng Yang. Overcoming the memory wall with cxl-enabled ssd. In 2023 USENIX Annual Technical Conference (USENIX ATC 23), pages 601–617, 2023.

[81] 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 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), pages 154–161. IEEE, 2017.

[82] Jeseong Yeon, Minseong Jeong, Sungjin Lee, and Eunji Lee. RFLUSH: Rethink the flush. In 16th USENIX Conference on File and Storage Technologies (FAST 18), pages 201–210, Oakland, CA, February 2018. USENIX Association.

[83] Yekang Zhan, Haichuan Hu, Xiangrui Yang, Shaohua Wang, Qiang Cao, Hong Jiang, and Jie Yao. Romefs: A cxl-ssd aware file system exploiting synergy of memory-block dual paths. In Proceedings of the 2024 ACM Symposium on Cloud Computing, SoCC '24, pages 720–736, New York, NY, USA, 2024. Association for Computing Machinery.

[84] Xuechen Zhang, Ke Liu, Kei Davis, and Song Jiang. ibridge: Improving unaligned parallel file access with solid-state drives. In 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, pages 381–392. IEEE, 2013.

[85] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. Ziggurat: A tiered file system for non-volatile main memories and disks. In 17th USENIX Conference on File and Storage Technologies (FAST 19), pages 207–219, 2019.

[86] 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 21st USENIX Conference on File and Storage Technologies (FAST 23), pages 265–280, 2023.

[87] 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 Proceedings of the 29th Symposium on Operating Systems Principles, pages 150–165, 2023.

[88] Diyu Zhou, Yuchen Qian, Vishal Gupta, Zhifei Yang, Changwoo Min, and Sanidhya Kashyap. Odinfs: Scaling pm performance with opportunistic delegation. In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), pages 179–193, 2022.

[89] Qingsong Zhu, Qiang Cao, and Jie Yao. Uhs: An ultra-fast hybrid storage consolidating nvm and ssd in parallel. In 2023 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 1–6. IEEE, 2023.

[90] Xiaowei Zhu, Wentao Han, and Wenguang Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In 2015 USENIX Annual Technical Conference (USENIX ATC 15), pages 375–386, 2015.

A Artifact Appendix

Abstract

The evaluated artifact is provided in a git repository and contains the scripts used for running the experiments presented in this paper.

Scope

The artifact has been made available. It includes detailed reference instructions to set up, deploy, and configure OrchFS and main experiments.

Contents

The artifact contains the steps to build, install, and configure OrchFS, including setting the size of three types of storage units, and the number of IO threads and the interlaced binding granularity in the parallel IO engine. It also contains the Bash scripts used for performance benchmarks, the user-friendly test examples used for other common evaluations, and corresponding documents for each script and example.

Hosting

The artifact is available at https://github.com/YekangZhan/OrchFS. All necessary instructions are provided in the README.md file. We encourage the users to use the latest version of the repository, since it may include bug fixes and additional functions.

Requirements

We evaluated OrchFS on the machine equipped with common enterprise PCIe4.0 and PCIe5.0 SSDs (§ 5.1). For machines with older PCIe generation devices and slower-speed devices, the benchmarks may not show similar results we present in the paper, but we believe the overall trends should be similar.