ByteFS: System Support for Memory-Semantic Solid-State Drives

Shaobo Li*
shaobol2@illinois.edu
University of Illinois Urbana-Champaign

Yirui Eric Zhou*
yiruiz2@illinois.edu
University of Illinois Urbana-Champaign

Hao Ren
haor2@illinois.edu
University of Illinois Urbana-Champaign

Jian Huang
jianh@illinois.edu
University of Illinois Urbana-Champaign

Abstract

Unlike non-volatile memory that resides on the processor memory bus, memory-semantic solid-state drives (SSDs) support both byte and block access granularity via PCIe or CXL interconnects. They provide scalable memory capacity using NAND flash at much lower cost, and have different performance characteristics for their dual byte/block interface respectively, while offering essential memory semantics for upper-level software. Such a byte-accessible storage device provides new implications on the software system design.

In this paper, we develop a new file system, named ByteFS, by rethinking the design primitives of file systems and SSD firmware to exploit the advantages of both byte and block-granular data accesses. ByteFS supports byte-granular data persistence to retain the persistence nature of SSDs. It extends the core data structure of file systems by enabling dual byte/block-granular data accesses. To facilitate the support for byte-granular writes, ByteFS manages the internal DRAM of SSD firmware in a log-structured manner and enables data coalescing to reduce the unnecessary I/O traffic to flash chips. ByteFS also enables coordinated data caching between the host page cache and SSD cache for best utilizing the precious memory resource. We implement ByteFS on both a real programmable SSD and an emulated memory-semantic SSD for sensitivity study. Compared to state-of-the-art file systems for non-volatile memory and conventional SSDs, ByteFS outperforms them by up to 2.7×, while preserving the essential properties of a file system. ByteFS also reduces the write traffic to SSDs by up to 5.1× by alleviating unnecessary writes caused by both metadata and data updates in file systems.

1 Introduction

Modern computer systems has an increasing demand for storage-class memory (SCM) [9, 17], as it promises many advantages such as scalable storage capacity, byte-addressable data accesses, and data durability. A typical example is non-volatile memory (NVM), which attracted much attention over the past decade [29, 35, 44, 47]. However, due to the insufficient supply of mature products on the market [7], we have to seek alternative practical memory technologies.

Thanks to the byte-accessibility of the PCIe interconnect (e.g., NVMe and CXL), the commodity flash-based SSDs can provide byte-addressable data accesses by utilizing the memory-mapped I/O interface (MMIO) [10, 12, 38]. Therefore, we can access SSDs with both byte and block granularity. We call them as *memory-semantic SSDs* (M-SSD). Unlike NVM devices such as Intel Optane Persistent Memory that use PCM-like storage medium and reside on the processor memory bus [42], the M-SSD is PCIe attached and built upon the mature NAND flash technology, which offers unique advantages. First, it relies solely on the byte-accessibility support of the existing PCIe interface, thus a majority of the commodity PCIe-attached SSDs can be reformed as memory-semantic SSDs. For instance, Samsung publicized its CXL-based SSDs built on the commodity NVMe SSDs recently [38]. Second, its backend storage medium NAND flash has much lower cost, and its storage capacity can easily scale up to terabytes per PCIe slot [10]. Third, it provides the memory interface for software systems while preserving compelling storage properties such as data durability and simple deployment.

However, for decades, systems software like file systems are used to the generic block interface to access SSDs, resulting in high I/O amplification [10, 34]. They lack the support for dual byte and block interface for M-SSDs, causing sub-optimal performance and increased complexity of data management. File systems for persistent memory such as PMFS [19] and NOVA [45] focused on the byte interface, however, they were designed for NVM devices whose characteristics are fundamentally different from M-SSDs. Thus, none of the current file systems is a natural fit for M-SSDs. The M-SSD provides a set of unique opportunities and challenges for systems software.

We present ByteFS, an efficient file system for M-SSDs with software/hardware co-design. ByteFS transparently supports dual byte/block interface for high-performance data accesses, while preserving the essential properties of file systems, including crash consistency and data recovery. ByteFS extends the SSD firmware by managing its internal DRAM cache in a log-structured manner for accommodating the byte-granular data accesses, and enabling data coalescing to reduce I/O traffic to flash chips. It also enables coordinated data caching between the host page cache and the internal DRAM of the SSD.

*Co-primary authors.

To develop ByteFS, we first conduct a thorough study of generic Linux file systems to understand the appropriate interface (i.e., byte or block) needed for core filesystem data structures for their interaction with the M-SSD, and their impact on I/O amplification. As we expected, some data structures like inode prefer byte-granular data access, and some structures like page cache prefer dual byte/block-granular data access depending on the data access pattern at runtime (see Table 3). Our study (§3) provides guidelines on developing ByteFS.

ByteFS enables byte-granular persistent writes to reduce I/O amplification for a majority of data structures. For the data structures that prefer dual byte and block interfaces, such as page cache, data block, and data journal, ByteFS employs different policies to decide the appropriate data access granularity. As for the page cache, ByteFS utilizes the copy-on-write mechanism to track the writes to a page, and learn their data locality. ByteFS uses byte-granular writes for hot cachelines, and block-granular writes for a page with lower data locality. As for the writes to data blocks and data journal, ByteFS decides the write granularity based on the amount of data required for the durable writes to the SSD. It employs block-granular reads for all data structures to exploit the locality in the host cache and simplify their management.

Although M-SSDs enable the byte interface via PCIe/CXL, the flash chips inside the SSD support only page-granular accesses due to physical limitations [10, 12], causing extra I/O amplification. To address this challenge, ByteFS extends the SSD firmware by managing the internal SSD DRAM in a log-structured manner at cacheline granularity, and enabling data coalescing with background log cleaning. This reduces unnecessary I/O traffic caused by the mismatch of data access granularity between SSD DRAM (byte-granular) and flash chips (page-granular). ByteFS develops a lightweight indexing mechanism using skip lists for fast log lookup, and implements coordinated data caching between the host and SSD DRAM. Instead of caching flash pages in both SSD DRAM and host page cache, ByteFS caches flash pages only in the host page cache for best utilizing the precious SSD DRAM for persistent writes.

ByteFS preserves essential file system properties, including crash consistency and data recovery. ByteFS develops a low-overhead transactional mechanism for filesystem operations using the byte-granular persistent writes and firmware-level logging. With the write log in the SSD DRAM, ByteFS facilitates the enforcement of crash consistency and data recovery.

We implement ByteFS based on the Ext4 file system, and develop a full system prototype on a programmable SSD FPGA board to validate its functions and efficiency. We extend the PCIe protocol on the FPGA board to support byte-granular persistent writes, and modify the SSD firmware for the write log. We also develop an M-SSD emulator for sensitivity analysis. Compared to the file systems developed for NVM, such as PMFS [19] and NOVA [45], and for block devices, such as Ext4, ByteFS improves the performance by up to 2.7×. ByteFS also shows its friendliness to SSDs by reducing the write traffic by up to 5.1×. In summary, we make the following contributions.

Table 1. Characteristics of different memory devices. M-SSD represents the memory-semantic SSD. DRAM and NVM are measured with single DIMM. SSD and M-SSD use PCIe 3.0 x4.

Memory	R/W Latency (cacheline)	Seq R/W BW (4KB)	$/GB	Persistency
DRAM [4, 29]	100 ns	31.8 GB/s	8.6	No
NVM [3, 25]	300/90 ns	6.6/2.3 GB/s	3.6	Yes
SSD [6]	N/A	3.5/2.5 GB/s	0.22	Yes
M-SSD [10, 12]	4.8/0.6 μs	3.5/2.5 GB/s	~0.22	Yes

- We conduct a characterization study of data access interface preferred by core data structures of file systems, it provides guidelines for enabling system software support for M-SSD.
- We develop a new file system ByteFS for memory-semantic SSDs by supporting adaptive byte and block-granular data accesses, it significantly reduces the I/O amplification.
- We extend the SSD firmware to manage the SSD DRAM in a log-structured manner and enable data coalescing to reduce the I/O traffic caused by the mismatch of access granularity between SSD DRAM and flash chips.
- We present a coordinated data caching mechanism between the host page cache and SSD DRAM for best utilizing the precious SSD DRAM for writes.
- We develop ByteFS with both a real programmable SSD and a memory-semantic SSD emulator, and demonstrate its efficiency with various filesystem benchmarks.

2 Background and Motivation

We first discuss the technical background of M-SSD. And then, we discuss why its system support is highly desirable.

2.1 Memory-Semantic SSDs

Storage devices like SSDs usually use the block interface to interact with upper-level system software like file systems. Recently, industry and academia have been exploiting the byte-addressability of SSDs [10, 26, 38]. Unlike non-volatile memory (NVM) [13, 15, 24] technologies, memory-semantic SSDs (M-SSDs) provide the byte interface by leveraging the in-device DRAM and the PCIe memory-mapped interface [14]. By registering the device buffer address with the *base address register* (BAR), the BIOS and OS discover and map the device buffer into the host memory space, and the host can access the mapped region via load/store instructions [10, 12]. The in-device DRAM is used as a data buffer to serve byte-granular requests. Battery-backed DRAM or large capacitors are used to assist data persistency. The M-SSD still keeps the normal block interface with conventional NVMe I/O commands, allowing the host to operate on dual byte and block interfaces.

We compare the performance and cost of M-SSDs with other types of memory devices in Table 1. We measure the

latency and bandwidth with real devices (see the experimental setup in §5.1). M-SSDs provide a 4.8μs read latency when reading cachelines in the device DRAM and offer a 0.6μs write latency with PCIe posted writes. With the new Compute Express Link (CXL) [22], the latency can be further reduced. However, it is still slower than NVMs that can achieve near-DRAM performance. For data accesses not served by the SSD DRAM, μs-level flash page accesses will result in a high delay.

Although the performance of M-SSDs might be less competitive compared to NVM or DRAM, they offer great cost-effectiveness. We obtained the lowest prices of these devices from popular memory vendors in 2024. The NVM price is derived from the Intel Optane Persistent Memory. Since we cannot obtain any price of M-SSD on the current market, we estimate it would be similar to SSDs, as they are developed upon commodity SSDs and have similar hardware components including the flash controller and flash chips. The cost would be possibly higher with CXL components, but compared to DRAM or NVM, M-SSDs are still significantly cheaper with a cost of around $0.22/GB, making it a cost-effective solution in building high-performance storage systems.

2.2 System Support for Memory-semantic SSDs

Although M-SSDs show great potential in offering high performance and large capacity at a low cost, they lack system software support. A straightforward approach is to directly employ traditional file systems. However, they were designed for the block I/O interface. They manage underlying storage devices with fixed-size blocks (e.g., 512 bytes or 4 KB), and cannot utilize the byte-granular access brought by the M-SSD. We study the limitations of the block interface in detail in §3.

Apart from the traditional block-based file systems, there have been many emerging NVM file systems [17, 19, 27, 29, 45]. These file systems are all developed for byte-addressable NVM devices, which commonly leverage memory load/store interface to exploit ultra-low latency accesses. Reducing software overhead has been a critical design goal for such systems, and many utilize Direct Access techniques to bypass the host page cache or the OS kernel. However, naïvely applying NVM file systems to M-SSDs is inadequate. The performance characteristics of M-SSDs differ from NVMs (§2.1), as the high PCIe latency and inevitable flash accesses are still the bottleneck.

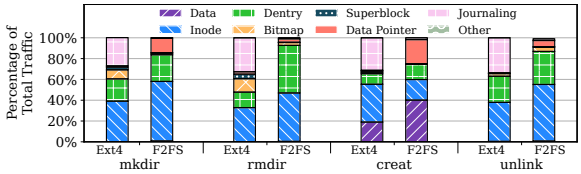
Therefore, it is hard to fully exploit the performance benefits of M-SSDs with existing file systems, as they all lack the support for both byte and block interface, and the consideration of the unique performance characteristics. This motivates us to develop a new system ByteFS for M-SSDs.

3 A Quantitative Study of Block I/O Interface

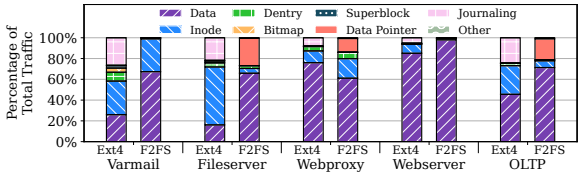
Before designing ByteFS, to further understand the limitations of the block I/O interface, we conduct a thorough study of the generic Linux file system Ext4 and F2FS. We profile the I/O traffic of the two file systems running Filebench [41] and

Table 2. I/O amplification of using block I/O interface in conventional file systems Ext4 and F2FS.

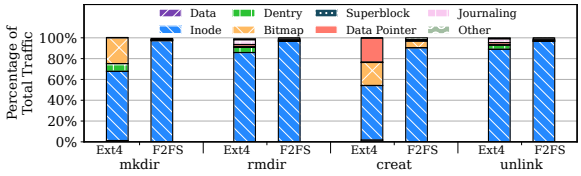
		Varmail	Fileserver	Webproxy	Webserver	OLTP
Ext4	Write	3.85×	6.21×	1.43×	1.66×	2.17×
	Read	1.21×	1.15×	1.25×	1.71×	1.52×
F2FS	Write	2.14×	1.92×	1.67×	1.06×	1.10×
	Read	1.67×	1.42×	1.35×	1.18×	1.13×



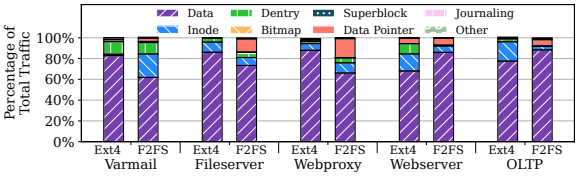
(a) Write traffic breakdown on micro-benchmarks.



(b) Write traffic breakdown on real applications.



(c) Read traffic breakdown on micro-benchmarks.



(d) Read traffic breakdown on real applications.

Figure 1. Read and write traffic breakdown of Ext4 and F2FS.

OLTP [18] benchmarks on the server machine described in §5.1. Unlike previous studies on file system I/O [34], we focus on the impact of each individual filesystem data structure using the block interface. We summarize the key data structures in Table 3 and show the traffic breakdown in Figure 1.

3.1 I/O Amplification in File Systems

We first focus on the read/write amplification introduced by metadata management. We collect the I/O amplification factor when running diverse workloads in Table 2. As expected, file system operations result in significant write amplification (1.1–6.2×), and metadata operations have a large I/O cost in Ext4 and F2FS. Similarly, the read amplification of Ext4 and F2FS is 1.1–1.7×. To further understand the impact of each filesystem data structure, we separate the data structures into metadata and data, and explain each of them in detail.

3.2 File Metadata Structures

Superblock. Superblock maintains the key properties of a file system. As shown in Figure 1, the on-disk superblock is

Table 3. The core components in the generic Linux file systems (Ext4 and F2FS) studied in this paper.

Data Structure	Description	Related File Operations	Disk Access Frequency	Preferred Interface
Superblock	The metadata that defines the file system.	File system mount and remount.	R: Low, refers to the general info of a file system. W: Low, infrequent global metadata update.	R: Block W: Block
Block List	Marks free and used blocks in the file system.	File create, append, and truncate.	R: Depends, the structure can be cached in host DRAM. W: High, when blocks are allocated/freed.	R: Block W: Byte
Inode List	Marks free and used inodes in the inode table.	File create and unlink.	R: Depends, the structure can be cached in host DRAM. W: High, when files are created and deleted.	R: Block W: Byte
Inode	Describes a file system object such as a file or a directory.	File/directory create, file rename, file truncate, file link/unlink, and others.	R: High, most operations involve file/dir metadata. W: High, most operations involve metadata updates.	R: Block W: Byte
Data Pointer	Indexes the location of the file data on the storage device.	File read and write.	R: High, when the file system reads data from any file. W: High, when the file system appends or truncates file.	R: Block W: Byte
Directory Entry	Holds child inode information under the directory.	File/dir create, rename, and link/unlink.	R: Depends, mainly on the file lookup. W: Depends, higher when having more dirs and files.	R: Block W: Byte
Page Cache	Transparent cache that exploits data locality of applications.	File create, read, and write.	R: High, for data-intensive workloads. W: High, for data-intensive workloads.	R: Block W: Block/Byte
Data Block	File data stored in the file system.	File create, read, and write.	R: High, for data-intensive workloads. W: High, for data-intensive workloads.	R: Block/Byte W: Block/Byte
Data Journal	History of operations executed in the file system.	Operation that modifies critical file system data structures.	R: Low, mostly accessed during recovery. W: High, most operations that require journaling.	R: Block W: Block/Byte

rarely accessed in nearly all workloads at runtime, only contributing 0.23% write traffic and 0.02% read traffic on average. Due to the low access frequency, we can still use the block I/O interface to simplify its management.

Block/Inode List. Block or inode lists are responsible for tracking free data blocks and inodes. In Ext4, they are implemented as bitmaps. F2FS [30] uses the segment information table (SIT) and node address table (NAT) to track them. Our study with various file operations, such as `mkdir`, `rmdir`, and `create`, show that bitmap accesses contribute 5.4% of write traffic on average and up to 25.2% of total read traffic in Ext4, due to the frequent inode/block allocations. During block/inode updates, only a few bytes in the bitmaps are flipped. This offers the opportunity to reduce the write traffic with the byte interface. To minimize the read accesses, we can cache the data structure in the host DRAM after loading it from the storage.

Inode. File systems use the inode to record the information of files and directories. The inode traffic contributes to 35% and 24.4% of the total writes on average in Ext4 and F2FS, as inodes are heavily involved in all file and directory updates. Persisting one 128B inode update requires a 4KB write to the disk, further amplifying the write traffic. Therefore, the byte interface can greatly reduce the inode write traffic.

For inode reads, loading an entire inode block brings all the inodes to the host, and they will be cached in the host DRAM. They contribute to 82% of total reads for metadata-intensive workloads. For data-intensive applications, it is reduced to 12.4%. Disk accesses can be reduced with data caching in Ext4 and F2FS when accessing files within the same inode block. Therefore, we can load inodes with the block interface and exploit the data locality with metadata caching.

Directory Entry. Directory entries (dentries) record the child directories and files within a directory. In Figure 1, the micro-benchmarks involve frequent file/directory creation and removal, and dentries contribute to 23% of write traffic on average. We also observe frequent dentry writes in `Varmail` and `Fileserver`, as they operate on many files. For workloads with infrequent directory operations, such as `Webserver` and `OLTP`, the dentry write traffic is negligible. Thus, to reduce write I/O amplification, the byte interface is preferred.

To look up a file or directory under a specific directory, the file system needs to search among the dentries. On average, 8% of the read traffic is spent on reading dentries, and the I/O amplification may further increase as we have deeper or wider directories. Loading the entire dentry structure with the block interface can avoid frequent reads from the storage device.

Data Pointer. Data pointers record the mappings from the file offset to the logical address of the storage device. Data pointers incur up to 26% of the total write traffic and 16% of the read traffic on F2FS. This is because F2FS performs out-of-place updates with frequent data pointer updates. To look up the target block address of the file data, we can read the whole block of data pointers into the host with the block interface.

In summary, most metadata updates in file systems are small, so the byte interface is suitable. For their reads, we can use the block interface to exploit data locality.

3.3 File Data Structures

File Data. File systems allow users to directly access and persist data to the storage bypassing the page cache with Direct I/O. For each I/O request, we can determine the access interface based on the request size in the POSIX call. When a large chunk of data needs to be accessed, we can use the block interface. For small accesses, we use the byte interface.

Page Cache. File systems use page cache to temporarily cache file data. When a page is not presented in DRAM, the entire page is brought into the host with the block interface. To reduce write I/O amplification, we can persist hot cachelines within dirty pages through the byte interface. For largely modified pages, we can write them back via the block interface.

Data Journal. File systems usually employ journaling techniques to prevent metadata/data inconsistency. In Ext4, 30.7% of the total traffic on average is caused by journaling under the

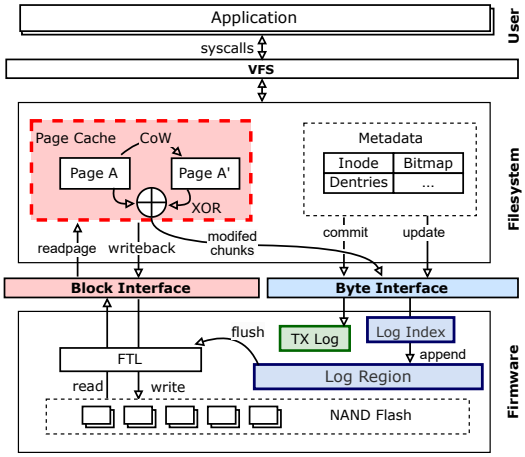


Figure 2. System overview of ByteFS.

ordered mode, as Ext4 performs double writes for critical metadata structures. F2FS manages node and data blocks with a log structure, and has low write amplification. We can use both byte and block interfaces depending on the journal data size.

In summary, file data may prefer both byte and block interfaces depending on the data locality. We should employ a flexible policy for deciding the appropriate interface at runtime.

4 Design and Implementation

Our goal is to develop a file system ByteFS that transparently supports dual byte and block interface for high-performance data access to memory-semantic SSDs.

Design Challenges. To achieve this, we need to overcome the following challenges: (1) ByteFS should minimize the I/O amplification across the storage software and hardware stack. However, there still exists an access granularity mismatch between the byte interface and internal flash chips in M-SSDs. (2) It is unclear how the file system should support both byte/block interfaces. ByteFS should provide the flexibility to exploit the benefits from both interfaces. (3) ByteFS should enforce data consistency with minimum overhead. (4) ByteFS should preserve the essential filesystem properties.

4.1 System Overview

Figure 2 shows the system overview of ByteFS. First, we discuss the techniques for enabling the dual interface in M-SSDs and ensuring persistent writes with the byte interface (§4.2). Second, we bridge the gap between byte-granular access and page-granular flash access with a new SSD firmware design. We develop a log-structured write buffer in the SSD DRAM, which enables data coalescing for the flash chip accesses. We maintain an address mapping to speed up the log lookup and log cleaning process (§4.3). We then describe how ByteFS supports read and write via the dual interface (§4.4). Third, we extend the filesystem metadata and data management to support the dual interface. ByteFS employs an interface selection mechanism based on the data access patterns (§4.5 and §4.6). Fourth, ByteFS ensures crash consistency by enforcing write ordering and atomicity via transactions. ByteFS utilizes the in-device write log as a redo log and supports lightweight data consistency and recovery (§4.7). Finally, we present a few filesystem operation examples in ByteFS (§4.8).

4.2 Enable Byte-granular Data Access/Persistence

The PCIe interface uses Base Address Registers (BARs) to advertise device memory-mappable regions to the host. During system boot-up, the BIOS and OS check the BARs and set up the memory-mapped address space for the device. All memory requests to the mapped region are forwarded to the device via the PCIe root complex. The M-SSD controller is responsible for handling memory requests. ByteFS leverages the BAR register to map the entire SSD as a memory region to the host. The host can access any SSD address with MMIO (byte interface). This can also be realized with CXL.mem protocol, with which the host CPU can issue cacheable load/store accesses to the device.

To enable persistency, the M-SSD can leverage battery-backed DRAM which allows all data in the DRAM to be flushed to the flash media during a power loss. However, the host CPU cache may hold dirty cachelines, or a PCIe transaction may be pending. ByteFS uses two steps to ensure a persistent write. First, it calls `clflush/clwb` after a memory-mapped write request to flush the CPU cache. Second, it issues a read request with zero byte following the write (write-verify read) to ensure the posted PCIe transactions are completed. Since the read/write requests are serialized in the root complex, a non-posted read will enforce the completion of previous writes.

4.3 Manage SSD DRAM as a Log-Structured Memory

Although PCIe or CXL provides byte-granular access to the SSD, there is still a mismatch of access granularity inside the SSD. The flash chips are only accessible via flash page granularity due to hardware limitations [10, 12, 21], although the host can access the SSD at byte granularity. This inevitably incurs extra flash accesses and consumes SSD DRAM cache space.

To bridge the gap between the byte-granular accesses and the page-granular flash accesses, ByteFS reorganizes part of the SSD DRAM cache into a log-structured region at cacheline granularity. Therefore, all writes via the byte interface can be directly appended to the log to avoid flash accesses on the critical path (see Figure 2). For reads, ByteFS implements a coordinated caching mechanism between the host and SSD DRAM. When inevitable flash accesses are required, we only cache the loaded pages in the host DRAM, which saves precious SSD DRAM. When the log utilization exceeds a threshold (85% by default), a background cleaning procedure coalesces the log entries and flushes them back to the flash chips.

Index Structure of the Write Log. As shown in Figure 3, the write log consists of a global log region and an indexing structure to index the log. The global log region buffers the data written via the byte interface. It is organized as a circular buffer (256 MB by default) with head/tail pointers. The written data is appended at the log tail as a 64B-aligned data entry.

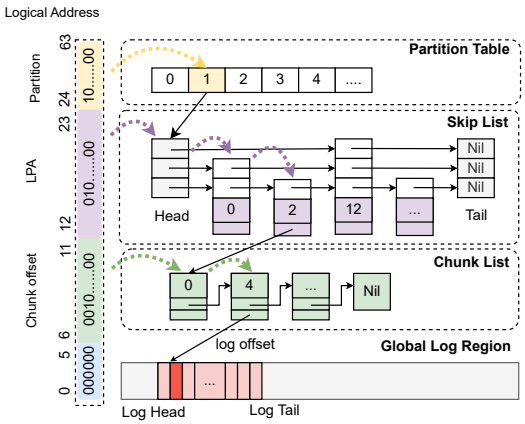


Figure 3. Structure of the write log in the M-SSD firmware.

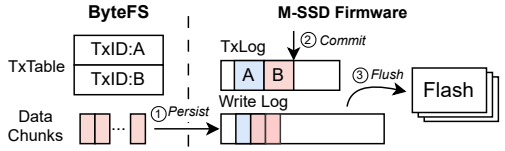


Figure 4. Transaction support with the firmware-level log.

We propose an efficient three-layer skip list to index the write log. Instead of having a single huge skip list, we break it into multiple smaller ones to reduce the indexing cost. In the first layer, we divide the entire SSD address space (e.g., 1TB) into fixed-size 16MB partitions. During lookup, ByteFS can quickly calculate the partition index by dividing the logical page address (LPA) by the partition size. Then, we find the corresponding skip list from the partition table with the partition index. In the second layer, if a flash page has its data stored in the global log region, an entry indexed by its LPA will be in the skip list. Each skip list entry points to an ordered chunk list in the third layer. The chunk list is ordered by the block offset in a page for fast lookup. Each chunk entry records the block offset (1B) in a page, log offset (4B) in the log region, and data length (4B) of the data entry stored in the global log region.

The skip list offers $O(log(n))$ lookup, insertion, and deletion time, where n is the number of entries in the skip list. Compared to hash tables, our indexing structure supports arbitrary chunk sizes and better range lookup performance, as the file system may issue accesses of various sizes. Our experiments with an embedded ARM processor show that the average lookup latency of a fully utilized 256MB log is 89 ns. The entire log index takes 21MB of DRAM space. The overhead is less of a concern, as the flash access latency is at the microsecond scale.

Transaction Support with Firmware-Level Logging. In ByteFS, all metadata updates are appended to the log region. Therefore, we can avoid double logging by directly using the write log as a redo log for metadata updates. This enables ByteFS to support atomic and crash-consistent updates.

We show ByteFS's transaction mechanism in Figure 4. When ByteFS initiates a new transaction, it assigns a unique 4B transaction ID (TxID) from a monotonically increasing global counter. ByteFS maintains a global transaction table (TxTable) to track all ongoing transactions with their TxIDs. Updates in a transaction are then encoded with the TxID and persisted in the write log (① in Figure 4). When all updates are persisted, ByteFS performs a commit by issuing a custom NVMe command `COMMIT(TxID)` with the target TxID. To maintain the commit order and status in firmware, we employ a 2MB transaction log (TxLog) in the SSD DRAM. Upon a `COMMIT(TxID)`, M-SSD firmware appends a 4B commit entry with TxID into TxLog (②). After the transaction is committed, the M-SSD firmware will be responsible for propagating the update back to the flash media via log cleaning. We maintain consistency by flushing the committed updates based on the commit order in TxLog (③). TxLog is cleaned up after the log cleaning.

Algorithm 1: Write log cleaning.

```
1  Function LogFlush(list_head):
2      for node in Traverse(list_head) do
3          buf = AllocWriteBuffer(node.lpa);
4          if CheckPartialUpdate(node) then
5              ReadFlashPage(node.lpa, buf)
6          for entry in node.chunk_list do
7              if !entry.committed then
8                  entry = GetLatestVersion(entry);
9              if entry then
10                 memcpy( entry.log_off, buf + entry.off, entry.size);
11         if WriteBuffer.full then
12             Buffer_Write();
13     CleanLogRegion();
```

Log Cleaning. Log cleaning serves to release the space of the log region and persist the updates back to flash chips. We first locate all modified pages by iterating through the second-layer skip lists (line 2 in Algorithm 1). For each modified page, we reserve a write buffer and check whether the old flash page should be loaded in case of partial updates (lines 3-5). We then traverse all modified blocks recorded in the third-layer chunk list and merge the latest committed version into the buffer (lines 6-10). If the newest update of an entry is not yet committed, we traverse the log region backward to find the latest committed version. We then write the buffer to the flash chips if it is full (lines 11-12). Finally, we clean up the log region and the corresponding log indexing structure (line 13).

To prevent a long cleaning process from blocking normal IO accesses, we employ double buffering to serve ongoing writes in a new log region, and flush the old log in the background.

4.4 System Support for Dual Byte/Block Interface

We now describe how ByteFS performs read and write accesses with the dual byte and block interface.

Read/Write via Byte Interface. ByteFS performs cacheline reads via the mapped SSD memory address space. The host CPU issues memory loads to the M-SSD via PCIe MMIO. For each load, the M-SSD firmware looks up the write log (see §4.3). If the entry is present in the log, the data is directly returned to

the host. Otherwise, the M-SSD fetches the page from the flash chips and only returns the requested cacheline to the host.

ByteFS aligns writes to cachelines as we manage the write log with 64B entries. For a single 64B write, we directly write the cacheline followed by a `clflush/clwb`. To write multiple cachelines atomically, we wrap them in a filesystem transaction. As the data reaches the SSD, the SSD firmware first appends the data into the write log. It then looks up the skip list and inserts/updates the chunk entry for the data.

Read/Write via Block Interface. ByteFS follows the normal block interface to access 4KB blocks with NVMe commands. Upon an NVMe read, the M-SSD firmware loads the page from the flash into a transfer buffer in SSD DRAM. Then, it looks up the skip list using the LPA of the requested flash page. If there are dirty cachelines in the log, the loaded page is merged with the latest data entries. Then, the page is returned to the host.

For write requests, 4KB blocks are transferred through PCIe to a write buffer in the FTL layer and then written to flash media. The SSD firmware scans the skip list for the written page and invalidates all corresponding entries in the write log. These log entries can be invalidated right away since ByteFS ensures that all written-back blocks from the host page cache are up to date. During later accesses to this page or log flushing, the invalid log entries will be ignored.

4.5 Manage Metadata Operations in ByteFS

We extend the core metadata structure in ByteFS and cache them in host DRAM. Upon cache misses, we load the metadata structure via the block interface. We describe the metadata structures and their operations in ByteFS as follows.

Inodes. ByteFS maintains the inode as a 128B entry and groups these entries into 4KB pages. To reduce the write traffic of inode updates, we split each inode into the upper and lower regions (64B each). The lower region contains frequently updated information, such as file size, modification times, and access rights. The upper region includes others. Therefore, each inode update takes as low as 64B via the byte interface, which can be done atomically. For complex operations that involve multiple metadata changes, ByteFS utilizes the transaction support (see §4.3) for atomic updates. ByteFS caches inodes using a radix tree in the host memory. Upon a cache miss, ByteFS loads the entire inode page via the block interface.

Directory Entries. In ByteFS, each directory holds an array of directory entries in its directory blocks. Each entry includes the inode number (4B) of the child file/directory, its file type (2B), filename length (2B), and filename (at most 256B). During directory lookups, we prefer to load the entire directory block once via the block interface, as we need to retrieve the full list of directory entries to read the associated inodes. ByteFS caches directory entries by their hashed directory names [40, 45] using a radix tree. Creating or renaming a directory involves updating a single directory entry, the update size varies from 64B to 320B based on the filename length, and the byte interface is used to perform the updates.

Block/Inode Bitmap. ByteFS maintains the inode and block allocation status with bitmaps. Each bitmap block is divided into multiple 64B groups as the basic unit of update. Upon system boot, ByteFS loads the bitmaps via the block interface. Similar to Ext4, ByteFS keeps a per-CPU free list for scalability and uses an extent-based allocation for data file blocks. Allocating new inodes or blocks incurs frequent small-size bitmap updates, which significantly benefit from the byte interface.

Data Pointers. ByteFS uses Ext4-like extent structure to index a range of contiguous file blocks with small extent nodes [33]. Each leaf extent node (16B) includes the file offset (8B), logical block address (4B), and length (4B). When reading or overwriting file data, ByteFS searches the extent tree to find disk pointers, and loads the entire block that contains all extent nodes. Since ByteFS uses in-place updates for file data, frequent overwrites to data pointers are less common. But it uses byte-granular persistent writes for the updates.

4.6 Manage Data I/O in ByteFS

ByteFS supports multiple file data I/O modes as normal file systems do, including direct I/O mode, buffered I/O mode, and memory-mapped I/O mode, as well as data journaling.

Direct Data Access. When users open a file with `O_DIRECT` flag, the read and write POSIX calls are performed under the Direct mode. When the data access size is no greater than 512B, ByteFS directly accesses the data via the byte interface. Otherwise, ByteFS uses the block interface to handle requests like conventional file systems. This is because when the write size is less than 512B, writing in cachelines offers a lower write latency than persisting an entire 4KB page.

Buffered I/O. In buffered I/O mode, the reads/writes are absorbed by the host page cache, and the data access size does not reflect the I/O traffic between the host and SSD. To select between the byte and block interface upon a dirty page writeback, ByteFS calculates the modified ratio (R) of this page.

To obtain the modified ratio, ByteFS uses the copy-on-write (CoW) mechanism in the page cache. When a cached data page is modified, ByteFS copies the original page to a *duplicate page*. Similar to a normal cached page, we track all duplicated pages with an `XArray` [43] structure indexed by file offset and store them in a per-inode `struct address_space`, where all cacheable and mappable objects are tracked.

Upon a dirty page writeback (e.g., after an `fsync/msync`), ByteFS checks the modified 64B data chunks by XORing the original page and the duplicated page. ByteFS calculates the modified ratio by $R = \frac{N_{Modified}}{N_{total}}$ where $N_{Modified}$ is the number of dirty cachelines and N_{total} is the total number of cachelines in the page. If R is less than $\frac{1}{8}$ (512B of a 4KB page), we use the byte interface to persist small accesses and vice versa. ByteFS uses `avx2` [8] instructions available on a majority of CPUs to perform 256-bit XOR operations. Based on our experiments on an

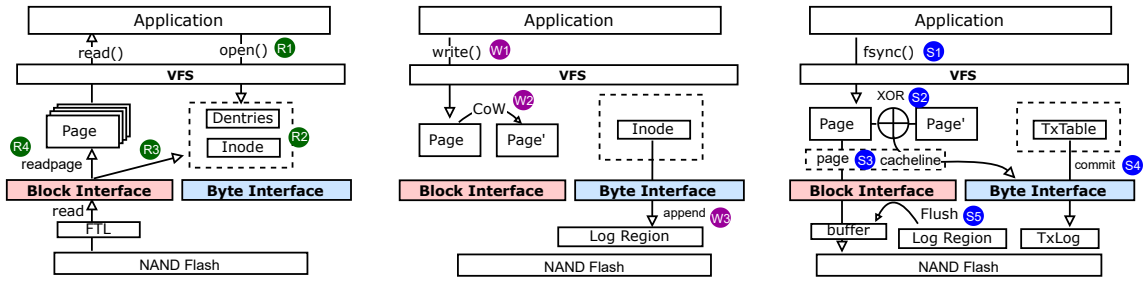


Figure 5. The workflow of ByteFS operations.

Intel server CPU (see §5.1), an XOR operation takes 936 CPU cycles on average and can achieve up to 14 GB/s throughput.

Memory-Mapped I/O. ByteFS allows the user-space program to directly access the file data with MMIO via `mmap()`. To map file data to a specific memory region, ByteFS leverages the cached DRAM pages and maps it to user address space. The interface selection mechanism with buffered I/O can be applied by performing CoW on the mapped file pages.

Data Journaling. ByteFS provides metadata logging that offers the same crash consistency guarantee as the `ordered` mode in Ext4. In addition, it also provides data journaling. For small data writes via the byte interface, we follow the same transaction mechanism in §4.3 to ensure crash consistency. For large ones written via the block interface, ByteFS combines JBD2 [28] with ByteFS transactions to support data journaling. Within a transaction, JBD2 writes updated data blocks in a reserved on-disk journal area. When commit, ByteFS appends a commit entry at the end of the JBD2 journal record to identify the commit status for data recovery.

4.7 Preserve Essential File System Properties

ByteFS preserves the essential properties of most file systems, including crash consistency and data recovery.

Crash Consistency. As discussed in §4.3, to ensure consistency, ByteFS leverages transactions to enforce atomicity and write ordering of metadata and data updates. With the firmware-level log-structured memory and battery-backed DRAM in M-SSD, ByteFS accelerates the transaction commits.

Data Recovery. M-SSD firmware can retain the cached content in the battery-backed SSD DRAM upon system failures. ByteFS can issue a custom NVMe command `RECOVER()` to the M-SSD firmware to start recovery. The M-SSD firmware will perform a complete scan to check all data entries in the log region. For each entry, the firmware checks the 4B TxID encoded at the end of the entry. If the TxID does not appear in the TxLog, we discard it as it is uncommitted. Then, the firmware performs a log flush writing all remaining committed entries back to the flash chips based on the order of the TxID in the TxLog. After the firmware finishes the recovery by cleaning up the log region and TxLog, ByteFS performs recovery on data journals if enabled. It scans the journal area for all transactions with a commit entry and moves the data blocks back in place.

4.8 Put It All Together

We show the workflow of ByteFS operations in Figure 5. We walk through examples of opening and reading a 16KB file, overwriting the first 1KB file chunks, and issuing an `fsync`.

Open/Read. An application first opens the target file with the file path (R1). ByteFS then performs a file lookup on the tree structure (R2). If a component is not cached in DRAM, ByteFS uses the block interface to fetch the entire block (R3). After finding the inode and opening the file, the application then issues a 16KB read based on the data pointer. File data is read in blocks and cached in the host page cache (R4).

Write. The application then issues `write` syscall to overwrite the first 1KB file data (W1). ByteFS checks the inode address_space [32] to find cached pages, and a copy-on-write is performed on the page (W2). ByteFS allocates a transaction with TxID_a for atomic updates. The inode is persisted through the byte interface to the firmware-level write log (W3).

fsync. When an `fsync` is issued (S1), ByteFS first XORs original page and copied page to identify the modified 1KB data chunk (S2). Based on the selection policy, the block interface is selected for the writeback (S3). Since we already persist the metadata updates, we commit the transaction with $\texttt{TxCommit}(\text{TxID}_a)$ after block I/O is completed (S4). We will flush the committed data to flash chips in the background (S5).

4.9 Implementation Details

ByteFS Implementation. We implemented ByteFS as a kernel file system based on Ext4 with 3.9K lines of code (LoC) in Linux kernel 5.1. We reorganize the on-disk metadata structure including superblock, bitmaps, inodes, and directory entries with 1.3K LoC to support the dual byte/block interface. To enable CoW, we add new `XArray` for indexing duplicate pages and modified the `writepage()` operation within the file system address space object [37] with 0.6K LoC.

M-SSD Prototype. To examine the effectiveness of our design, we build a real M-SSD prototype with an OpenSSD FPGA board, which has an onboard ARM core, 1TB flash storage with 16 channels, and 1GB DRAM. We modified the SSD firmware with 1.5K lines of C code. We customize the NVMe protocol to enable byte-granularity accesses, reserve a 256 MB log region in the SSD DRAM as the write log, and implement the three-layer skip list with 0.8K LoC. The log operations and

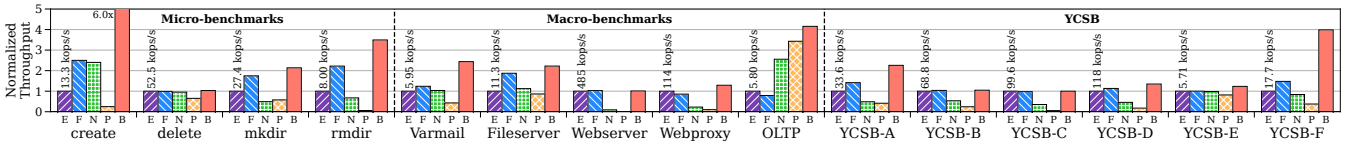


Figure 6. Overall throughput improvement (normalized to Ext4).

Table 4. System configurations.

Host Machine Configuration			
Processor	Intel(R) Xeon(R) E5-2683 v3		
Memory Size	128 GB		
SSD Emulator Configuration			
Capacity	32 GB	R/W Bandwidth	3.5/2.5 GB/s
Page Size	4 KB	Flash R/W Latency	40/60 μs
Channel Count	8	Cacheline R/W Latency	4.8/0.6 μs

Table 5. Workloads used in our evaluation.

Workload	Description
Microbenchmarks	
`create`	File creation, 1M 4KB files, 12 threads
`delete`	File deletion, 1M 4KB files, 12 threads
`mkdir`	Make directories, 1M dirs, 12 threads
`rmdir`	Remove directories, 1M dirs, 12 threads
Applications	
`Varmail`	1M 16KB files, 12 threads
`Fileserver`	100k 128KB files, 12 threads
`Webproxy`	1M 16KB files, 12 threads
`Webserver`	1M 16KB files, 12 threads
`OLTP`	1.6K 10MB files, 200 threads
YCSB on RocksDB	10M KV pairs, 40M ops, zipfian distribution

transaction support take 0.4K LoC in total. We preserve the original SSD FTL layer and its core functionalities.

M-SSD Emulation. We also build an M-SSD emulator with 2.1K lines of C code based on the core logic of FEMU [31]. We dedicate a kernel thread pinned to one CPU core to emulate the normal FTL thread that operates on the embedded SSD processor. We incorporate all core FTL functionalities, such as page allocation, page-level translation, and garbage collection, to emulate the internal structure of M-SSD. To emulate the flash media, we reserve a contiguous region of DRAM memory with `memmap` boot options and add I/O latency to emulate the NAND flash latency. Table 4 shows all the parameters in detail.

5 Evaluation

We show that: (1) ByteFS outperforms existing file systems by up to 2.7× when running filesystem benchmarks and real applications (§5.2); (2) ByteFS leverages the dual interface to reduce I/O traffic between host and device by up to 5.1×. Its log-structured in-device memory and coalescing mechanism largely reduces flash accesses (§5.3); (3) Its individual components are effective (§5.4); (4) It can recover from a crash in a short time (§5.5); and (5) ByteFS is effective with different device configurations (§5.6).

5.1 Evaluation Setup

We run ByteFS on a dual-socket, 28-core Intel(R) E5 platform with a base frequency of 2.7GHz and 128GB memory. Our SSD platform includes both a real SSD prototype and an SSD emulator as discussed in §4.9. For emulation, we emulated SSD with 4KB flash pages and a log region of 256MB. We list all core settings in Table 4. We use various workloads including file system benchmarks Filebench [41] and real-world applications YCSB on RocksDB [5], as shown in Table 5. We compare ByteFS with state-of-the-art file systems, including Ext4, F2FS, NOVA, and PMFS, running on M-SSDs with SSD cache managed in page granularity. In figures, Ext4, F2FS, NOVA, PMFS, and ByteFS are shown as 'E', 'F', 'N', 'P', and 'B'.

5.2 Overall Performance Improvements

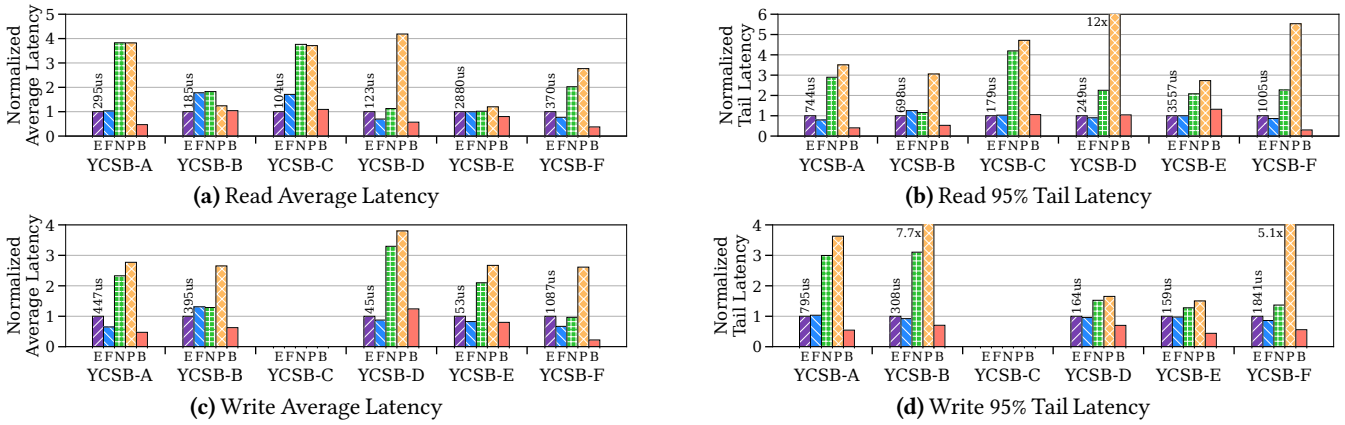
We evaluate ByteFS with filebench micro-benchmarks, macro-benchmarks, and YCSB workloads on the real SSD prototype. We show the throughput improvement in Figure 6.

Micro-benchmarks. ByteFS outperforms Ext4 by 2.5× and F2FS by 1.48× on average across all micro-benchmarks. On file creation especially, ByteFS achieves 6.0× and 2.4× performance improvement compared to Ext4 and F2FS. ByteFS can also obtain 1.2×-1.5× performance speedup on `mkdir` and `rmdir` when compared to F2FS, because ByteFS involves fewer data movements between host and device to perform file system operations with the dual interface. On `delete`, ByteFS can achieve similar performance with Ext4 and F2FS, since delete (i.e., unlink) operations do not require an immediate sync to the device. NOVA and PMFS perform even worse than EXT4 and F2FS in most cases since they are not designed for flash-based SSDs, they purely rely on the byte interface which fails to exploit the spatial locality with the block interface.

Macro-benchmarks. ByteFS achieves a higher or similar throughput across all macro-benchmarks. For `Varmail`, ByteFS outperforms F2FS by 1.9 ×, as `Varmail` frequently creates small files and performs small synchronous I/Os. ByteFS can speed up the frequently issued syscalls including `create` and `fsync` by persisting small metadata via the byte interface.

`Fileserver` is a data-intensive workload that reads and appends to relatively large files. ByteFS uses block interface to persist large amounts of data to the disk with high parallelism, while reducing the critical metadata writes caused by frequent appending with byte interfaces. As a result, ByteFS achieves over 2.2× throughput improvement compared to Ext4, and 1.2× improvement compared to F2FS.

`Webserver` and `Webproxy` are read-heavy workloads. ByteFS shows a similar performance compared with Ext4 and F2FS, as we leverage the block interface and host-side caching to exploit data locality. ByteFS outperforms Ext4 by 1.3× on



(a) Read Average Latency

(b) Read 95% Tail Latency

(c) Write Average Latency

(d) Write 95% Tail Latency

Figure 7. Latency of YCSB workloads (normalized to Ext4). YCSB-C does not have write latency as it is read-only.

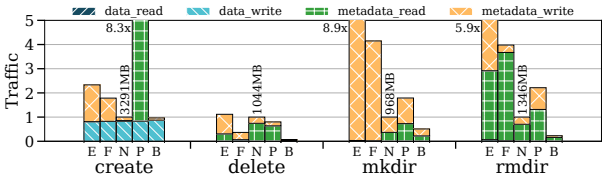


Figure 8. Host-SSD I/O traffic breakdown with Filebench Micro-benchmarks (normalized to NOVA).

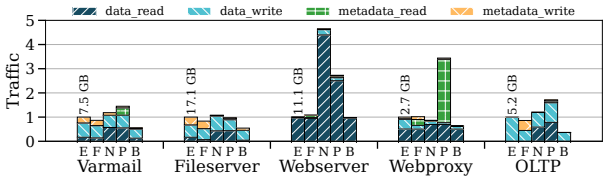


Figure 9. Host-SSD I/O traffic breakdown with Filebench Macro-benchmarks (normalized to Ext4).

`Webproxy`, since `Webproxy` also involves heavy directory operations, which benefits from the byte interface in ByteFS.

Finally, ByteFS outperforms Ext4 by 4.1× on write-intensive workload `OLTP`. It creates over 200 threads creating, appending, and overwriting files with frequent `fdatasync`. NOVA and PMFS can use byte interface to directly persist small updates, which reduces sync overhead and scales better under high contention. ByteFS further outperforms them by reducing consistency overhead with the in-device write log.

YCSB Workloads. To evaluate ByteFS on real-world applications, we run YCSB workloads on RocksDB. We report average latency and 95% tail latency for Read and Update in Figure 7. ByteFS provides 2.4× better throughput compared to F2FS. Similarly, ByteFS offers lower average and tail latency. Its major benefit comes from reducing the critical-path write latency, which also improves read latency as the write requests may block the read requests in RocksDB. For `YCSB-A` and `YCSB-F` which has a read/write ratio of 50/50, ByteFS improves the average/tail latency by 2.3×/2.0× for reads and 1.3×/1.6× for writes compared to F2FS. In workloads with a lower write ratio (i.e., `YCSB-B` and `YCSB-D` with a 95/5 read/update ratio), ByteFS provides relatively less performance improvement over EXT4. In `YCSB-C` with 100% read operations, ByteFS achieves similar performance as EXT4 or F2FS. It also achieves similar performance on `YCSB-E` compared to other baselines, since it performs range scanning following a uniform distribution on the entire dataset, and has almost no locality.

5.3 I/O Traffic Breakdown

I/O Traffic Between Host and SSD. Figure 8 shows the traffic between host and SSD with micro-benchmarks. Compared to Ext4 and F2FS which only use block interface, ByteFS reduces the metadata traffic by up to 25.3× and 17.2× (11.4× and 6.1× on average) with the byte interface. Compared to NOVA or PMFS, which also use the byte interface, ByteFS can still reduce the metadata traffic. This is for two reasons. First, NOVA and PMFS use out-of-place updates to ensure crash consistency, while ByteFS makes in-place updates of metadata and reduces consistency overheads with the write log in the SSD firmware, thus avoiding the double writes on the metadata. Second, ByteFS reduces metadata reads with block interface and host-side metadata caching. Compared to NOVA, ByteFS reduces nearly 43% metadata read traffic on average.

Figure 9 shows the I/O traffic for the macro-benchmarks. Similar to the micro-benchmarks, ByteFS significantly reduces the metadata traffic. In addition, ByteFS also reduces data traffic with the dynamic block/byte interface selection and host-side page caching. `Varmail` and `Fileserver` combine frequent file creation and deletion operations with heavy file append operations. Compared to Ext4 and F2FS, ByteFS reduces the metadata update traffic with the byte interface. Compared to NOVA and PMFS, ByteFS reduces data read overhead by adaptively exploiting the block interface. In read-heavy workloads `Webserver` and `Webproxy`, ByteFS reduces the data traffic by up to 4.7× and 2.7× compared to NOVA and PMFS. `Webproxy` also involves heavy directory operations, so ByteFS reduces metadata traffic compared to Ext4 and F2FS. In `OLTP`, ByteFS reduces the data write traffic by 1.6× and 2.2× compared to NOVA and PMFS. Both of them incur extra write traffic due to their page-granular copy-on-write mechanism to maintain crash consistency, while ByteFS employs the byte-granular in-device write log.

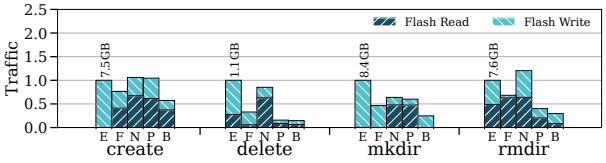


Figure 10. SSD flash traffic breakdown with Filebench Micro-benchmarks (normalized to Ext4).

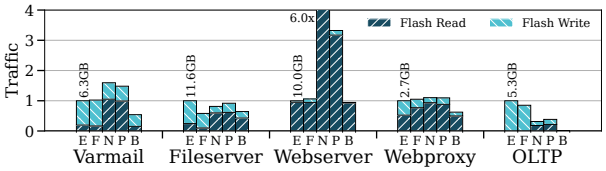


Figure 11. SSD flash traffic breakdown with Filebench Macro-benchmarks (normalized to Ext4).

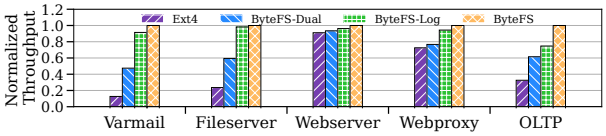


Figure 12. ByteFS perforamce breakdown. *ByteFS-Dual*: ByteFS with only dual interface. *ByteFS-Log*: ByteFS-Dual with log-structured memory. *ByteFS*: the full ByteFS design.

SSD Flash Traffic. We now evaluate the effectiveness of the log-structured memory in the SSD. Figure 10 and Figure 11 show the flash traffic in the SSD. On average, ByteFS reduces the flash traffic by 2.9×, 2.1×, 3.2×, and 2.2× compared to Ext4, F2FS, NOVA, and PMFS. ByteFS reduces flash write traffic by coalescing small writes in the in-device write log. ByteFS also reduces flash read traffic as it does not need to fetch the corresponding page from the flash upon a partial write.

Sometimes, ByteFS may incur higher flash traffic than other file systems. For example, in `create` and `Fileserver`, ByteFS incurs higher flash read traffic than EXT4 and F2FS. In `mkdir` and `rmdir`, ByteFS may incur higher flash write traffic than NOVA and PMFS. This is caused by the read-modify-write pattern with partial writes during the log-cleaning stage. On the other hand, NOVA and PMFS use page-granular cache in the SSD DRAM, so a cached page may absorb more writes when the locality is good enough. The higher flash traffic in ByteFS does not affect performance since the log cleaning happens in the background. Such overhead is completely outweighed by the performance benefits of the in-device write log.

5.4 Performance Breakdown of ByteFS

To understand how each design component in ByteFS contributes to the overall performance, we evaluate three variants of ByteFS in Figure 12. `Varmail` and `Fileserver` benefit from both the dual interface and the log-structured write buffer, as both techniques help reduce the I/O amplification of these two workloads. `Webproxy` mostly benefits from the dual interface, as it involves heavy directory operations. `OLTP` benefits from both log-structured memory and flexible interface selection, as it involves frequent small overwrites.

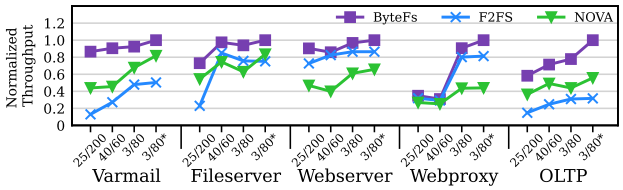


Figure 13. Macro benchmark performance under different flash latency (`Read/Write`).

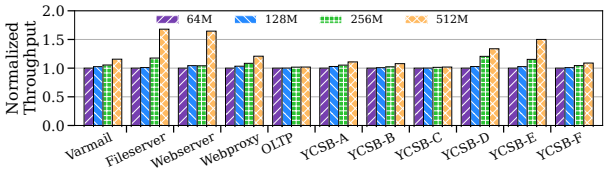


Figure 14. Performance impact of the log region size (normalized to 64MB).

5.5 Recovery Time of ByteFS

To measure the recovery time of ByteFS after a system crash, we intentionally power off the system after running YCSB workloads. ByteFS can recover the system in 4.2 seconds on average after reboot, with the recovery process described in §4.7. It takes 0.9 seconds to load the entire SSD DRAM content and 2.7 seconds to scan the global log region and TxLog to flush all committed entries into the flash media.

5.6 Sensitivity Analysis

Vary M-SSD Access Latency. We vary the emulator configuration with various NAND flash read/write latencies from low-end to high-end SSDs [2, 16, 46]. As shown in Figure 13, ByteFS outperforms F2FS and NOVA regardless of flash access latencies. We also emulate a CXL-based SSD with less cache-line access latency (175 ns [39]) and high-end flash media (marked as 3/80*). The benefit of ByteFS is larger with higher flash write latency, as ByteFS hides the flash write latency by flushing the in-device write log in the background. With CXL, both ByteFS and NOVA have better performance in some workloads as the CXL protocol reduces the access latency of the byte interface. However, NOVA is still slower than ByteFS since it does not optimize for the high flash access latency.

Vary SSD DRAM Log Size. As shown in Figure 14, ByteFS can scale its performance with a larger SSD DRAM. As we increase the write log size, ByteFS can cache and coalesce more updated entries inside the DRAM before flushing. While a larger log size might lead to a longer flush time, many workloads still benefit from a larger log size, as the log flushing is done in the background. For some workloads (e.g., OLTP) the benefits are marginal as they already have good write locality.

6 Related Work

Storage Class Memory. The hardware community has been focused on developing scalable memory technologies such as

PCM [36], ReRAM [11], and FeRAM [20] to increase the memory capacity. However, many of them are not available in real products. Intel released the Optane persistent memory [23] in 2019, but it was shut down in 2022. This motivates the community to investigate alternatives. Most recently, research demonstrated the feasibility of memory-semantic SSDs [1, 12]. ByteFS targets this new type of storage device and enables the system software support to exploit its unique properties.

Memory-Semantic SSDs. To exploit the byte-addressability of SSDs, FlatFlash [10] used the SSD as main memory by mapping the SSD into the host virtual memory. With the new CXL protocols, Jung et al. developed an FPGA-based emulator [26] for mapping the SSD as a memory extension. However, few of them investigated how the M-SSD should be managed at the systems software side. ByteFS provides efficient system software support for M-SSDs. It rethinks the SSD firmware design to address the fundamental mismatch of data access granularity between the SSD DRAM and flash chips.

Persistent Memory Systems. To support byte-addressable persistent memory, prior studies have developed a variety of file systems [17, 19, 27, 45]. For example, BPFS [17] developed an optimized file system in OS kernel and optimized shadow paging technique for ensuring crash consistency. PMFS [19] optimized the data access to persistent memory via direct access. NOVA [45] implemented a per-inode log-structured file system. SplitFs [27] handled data operations entirely in the user space and processed the metadata operations in Ext4. Unlike these prior studies, ByteFS develops a file system for M-SSDs that have fundamentally different device characteristics.

7 Conclusion

We present ByteFS, a new kernel-level file system for memory-semantic SSDs with dual byte/block interface. It significantly reduces I/O amplification across the entire storage stack. We implement ByteFS on both a programmable SSD FPGA board and an SSD emulator to demonstrate its efficiency.

Acknowledgments

We thank the anonymous reviewers and our shepherd Haris Volos for their insightful comments and feedback. We thank the members in the Systems Platform Research Group at University of Illinois Urbana-Champaign for constructive discussions. We also thank Yuqi Xue for proofreading the paper. This work was partially supported by NSF grants CCF-2107470 and CCF-1919044.

References

[1] Memory semantic ssd. https://samsungmsl.com/ms-ssd/.

[2] intel Optane SSD 905P 1TB Performance Testing. https://www.tomshardware.com/reviews/intel-optane-ssd-905p,5600-2.html, 2021.

[3] Intel® Optane™ Persistent Memory 200 Series Brief. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-persistent-memory-200-series-brief.html, 2021.

[4] Passmark - SK Hynix HMA82GS7CJR8N-VK 16GB - Price Performance Comparison. https://www.memorybenchmark.net/ram.php?ram=SK+Hynix+HMA82GS7CJR8N-VK+16GB&id=12578, 2021.

[5] RocksDB. https://github.com/facebook/rocksdb, 2021.

[6] Samsung SSD 970 Pro. https://www.samsung.com/semiconductor/minisite/ssd/product/consumer/970pro/, 2021.

[7] Why Intel killed its Optane memory business. https://www.theregister.com/2022/07/29/intel_optane_memory_dead/, 2022.

[8] Intel® Instruction Set Extensions Technology. https://www.intel.com/content/www/us/en/support/articles/000005779/processors.html, 2024.

[9] What is storage class memory. https://www.purestorage.com/knowledge/what-is-storage-class-memory.html, 2024.

[10] Ahmed Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-mei Hwu. Flatflash: Exploiting the byte-accessibility of ssds within a unified memory-storage hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 971–985, New York, NY, USA, 2019. Association for Computing Machinery.

[11] H. Akinaga and H. Shima. Resistive random access memory (reram) based on metal oxides. *Proceedings of the IEEE*, 98(12):2237–2251, Dec 2010.

[12] Duck-Ho Bae, Insoon Jo, Youra Adel Choi, Joo-Young Hwang, Sangyeun Cho, Dong-Gi Lee, and Jaeheon Jeong. 2b-ssd: The case for dual, byte- and block-addressable solid-state drives. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 425–438, 2018.

[13] Roberto Bez and Agostino Pirovano. Non-volatile memory technologies: emerging concepts and new materials. *Materials Science in Semiconductor Processing*, 7(4-6):349–355, 2004.

[14] Ravi Budruk, Don Anderson, and Tom Shanley. *PCI express system architecture.* Addison-Wesley Professional, 2004.

[15] An Chen. A review of emerging non-volatile memory (nvm) technologies and applications. *Solid-State Electronics*, 125:25–38, 2016.

[16] Wooseong Cheong, Chanho Yoon, Seonghoon Woo, Kyuwook Han, Daehyun Kim, Chulseung Lee, Youra Choi, Shine Kim, Dongku Kang, Geunyeong Yu, et al. A flash memory controller for 15μs ultra-low-latency ssd using high-speed 3d nand flash with 3μs read time. In *2018 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 338–340. IEEE, 2018.

[17] 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*, SOSP '09, pages 133–146, Big Sky, MT, 2009.

[18] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *PVLDB*, 7(4):277–288, 2013.

[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 9th European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, Amsterdam, The Netherlands, 2014.

[20] Ferroelectric RAM. https://en.wikipedia.org/wiki/Ferroelectric_RAM.

[21] Sujan K. Gonugondla, Mingu Kang, Yongjune Kim, Mark Helm, Sean Eilert, and Naresh Shanbhag. Energy-efficient deep in-memory architecture for nand flash memories. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2018.

[22] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Compute express link (cxl), 2017.

[23] Intel. 3D XPoint: A Breakthrough in Non-Volatile Memory Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html, 2018.

[24] Intel. Intel® optane™ persistent memory. http://www.intel.com/optanedcpersistentmemory/, 2021.

[25] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[26] Myoungsoo Jung. Hello bytes, bye blocks: Pcie storage meets compute express link for memory expansion (cxl-ssd). In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '22, page 45–51, New York, NY, USA, 2022. Association for Computing Machinery.

[27] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. Splitfs: Reducing software overhead in file systems for persistent memory. SOSP '19, page 494–508, New York, NY, USA, 2019. Association for Computing Machinery.

[28] The kernel development community. Journal (jbd2) - The Linux Kernel documentation. https://www.kernel.org/doc/html/latest/filesystems/ext4/journal.html, 2024.

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

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

[31] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S. Gunawi. The CASE of FEMU: Cheap, accurate, scalable and extensible flash emulator. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 83–90, Oakland, CA, February 2018. USENIX Association.

[32] Linux. Linux Page Cache. https://tldp.org/LDP/lki/lki-4.html.

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

[34] Jayashree Mohan, Rohan Kadekodi, and Vijay Chidambaram. Analyzing IO amplification in linux file systems. *CoRR*, abs/1707.08514, 2017.

[35] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA*, pages 24–33, 2009.

[36] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y. C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S. H. Chen, H. L. Lung, and C. H. Lam. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4.5):465–479, July 2008.

[37] Pekka Enberg Richard Gooch. Overview of the Linux Virtual File System - The Linux Kernel documentation. https://www.kernel.org/doc/html/latest/filesystems/vfs.html, 2024.

[38] Samsung Semiconductor. Cmm-h (cxl memory module - hybrid): Samsung's cxl-based ssd for the memory-centric computing era. https://semiconductor.samsung.com/us/news-events/tech-blog/webinar-memory-semantic-ssd/, 2023.

[39] Debendra Das Sharma. Compute express link®: An open industry-standard interconnect enabling heterogeneous data-centric computing. In *2022 IEEE Symposium on High-Performance Interconnects (HOTI)*, pages 5–12, 2022.

[40] Chia-Che Tsai, Yang Zhan, Jayashree Reddy, Yizheng Jiao, Tao Zhang, and Donald E Porter. How to get more value from your file system directory cache. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 441–456, 2015.

[41] Vasily Tarasov, *et al.* Filebench: A flexible framework for file system benchmarking. *The USENIX Magazine*, 41(1), 2016.

[42] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. Characterizing and modeling non-volatile memory systems. In *Proceedings of 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'20)*, 2020.

[43] Matthew Wilcox. XArray - The Linux Kernel documentation. https://www.kernel.org/doc/html/latest/core-api/xarray.html, 2024.

[44] Fangnuo Wu, Mingkai Dong, Gequan Mo, and Haibo Chen. Treesls: A whole-system persistent microkernel with tree-structured state checkpoint on nvm. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP'23)*, Koblenz, Germany, 2023.

[45] 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, Santa Clara, CA, February 2016. USENIX Association.

[46] Shao-Peng Yang, Minjae Kim, Sanghyun Nam, Juhyung Park, Jin yong Choi, Eyee Hyun Nam, Eunji Lee, Sungjin Lee, and Bryan S. Kim. Overcoming the memory wall with CXL-Enabled SSDs. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 601–617, Boston, MA, July 2023. USENIX Association.

[47] 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 (SOSP'23)*, Koblenz, Germany, 2023.