Modernizing File System through In-Storage Indexing

Jinhyung Koo, Junsu Im, Jooyoung Song, and Juhyung Park, *DGIST;*
Eunji Lee, *Soongsil University;* Bryan S. Kim, *Syracuse University;*
Sungjin Lee, *DGIST*

https://www.usenix.org/conference/osdi21/presentation/koo

This paper is included in the Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation.

July 14–16, 2021

978-1-939133-22-9

Open access to the Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

Modernizing File System through In-Storage Indexing

Jinhyung Koo
DGIST

Junsu Im
DGIST

Jooyoung Song
DGIST

Juhyung Park
DGIST

Eunji Lee
Soongsil University

Bryan S. Kim
Syracuse University

Sungjin Lee
DGIST

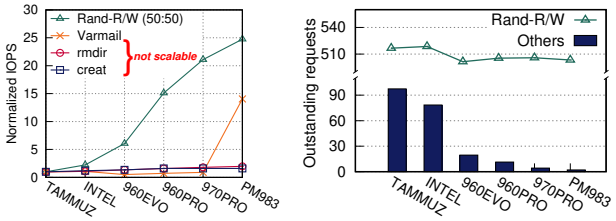
Abstract

We argue that a key-value interface between a file system and an SSD is superior to the legacy block interface by presenting KEVIN. KEVIN combines a fast, lightweight, and POSIX-compliant file system with a key-value storage device that performs in-storage indexing. We implement a variant of a log-structured merge tree in the storage device that not only indexes file objects, but also supports transactions and manages physical storage space. As a result, the design of a file system with respect to space management and crash consistency is simplified, requiring only 10.8K LOC for full functionality. We demonstrate that KEVIN reduces the amount of I/O traffic between the host and the device, and remains particularly robust as the system ages and the data become fragmented. Our approach outperforms existing file systems on a block SSD by a wide margin – 6.2× on average – for metadata-intensive benchmarks. For realistic workloads, KEVIN improves throughput by 68% on average.

1 Introduction

Files and directories are the most common way of abstracting persistent data. Traditionally, storage devices like hard disk drives simply export an array of fixed-sized logical blocks, and file systems abstract these blocks into files and directories containing user data by managing the storage space (*e.g.,* bitmaps) and the locations for the data (*e.g.,* inodes). Whenever files and directories are created or deleted, the file-system metadata, such as bitmaps and inodes, must be retrieved and updated to reflect the newly updated state of the system [3]. Since these persistent data structures must remain consistent, file systems need to employ techniques like journaling to ensure that they are atomically updated [2, 35, 38, 47]. Considering all of these responsibilities, file systems are highly intricate and performance-critical software [27, 37, 45].

However, the architecture of complex and sophisticated file systems that sits on top of storage devices with a simple array-of-blocks interface is ill-suited for today's technology trends. Before processing the actual file operations, file systems have to perform extra operations on on-disk metadata. This not only involves many extra I/Os and data transfers over the host interface, but also causes serious delays owing to I/O ordering [6, 7, 52] and journaling [26, 32]. The end of Moore's Law [50] means that the performance of file systems can no longer scale with faster CPUs. Moreover, the rise of fast storage devices like solid-state drives (SSDs) further exacerbates this problem, shifting the system bottleneck from the device to the host-side software I/O stack.



(a) EXT4 performance (b) Number of outstanding requests

Figure 1: The performance of the EXT4 file system with respect to SSD performance. With the current block interface, the file system exhibits poor performance scalability under metadata and `fsync` intensive workloads.

Figure 1 illustrates this problem by measuring the performance of the EXT4 file system as the performance of the underlying SSD increases: TAMMUZ is the slowest one, while PM983 is the fastest. We run three benchmarks: `creat` and `rmdir` as the metadata-intensive workload and `Varmail` [48] as the fsync-intensive workload. As a performance indicator for the six SSDs, we also run `Rand-R/W` that issues random reads/writes to the SSD which is directly mounted to the host without the file system. The measured throughput in Figure 1(a) is normalized to that of the slowest SSD (TAMMUZ). Under `Rand-R/W` without any metadata operations, the I/O performance increases greatly by up to 24.8× as the SSD gets faster. However, under `creat` and `rmdir`, the file system's performance increases by only 1.6× and 2.0×, respectively. Similarly, for `Varmail`, the measured throughput scales poorly from TAMMUZ to 970PRO (the second fastest SSD); the 14.0× improvement for PM983 is only possible because

the SSD ignores `fsync`[1]. Figure 1(b) shows the number of outstanding requests (measured by `iostat`) averaged across the metadata- and fsync-intensive workloads, and compares it with that of `Rand-R/W`. For `Rand-R/W`, the host system can fully utilize the performance of the underlying SSD by sending a sufficient number of I/Os. Thus, the I/O performance is mostly decided by the SSD performance. However, under the metadata- and fsync-intensive workloads, the file system fails to submit large enough I/Os to fully drive the SSD, in particular when the underlying SSD is fast, which results in much lower throughputs. These results indicate that we cannot increase the overall I/O performance just by improving the performance of the underlying SSD.

To alleviate this problem, we believe it is necessary to rethink the storage interface between the file system and the storage device; an independent improvement at either the file system or the device cannot solve the issue imposed by the legacy block interface. We are not the first to put forward this argument: many prior works have investigated extending the block interface [6, 16] or exposing a file object interface [23]. However, these either have a limited scope (*e.g.,* OPTR [6] on ordering and Janus [16] on fragmentation) or require a significant amount of resources (*e.g.,* DevFS [23] with respect to memory and CPU) that limit their effectiveness.

In this work, we argue that a key-value interface between the file system and the SSD is a better choice over the legacy interface for three primary reasons. First, it is simple and well-understood: it is widely used not only in databases (*e.g.,* key-value stores and backend storage engines for databases [12]), but also as a common programming language construct (*e.g.,* dict in Python). Second, there is great interest in the industry with the development of KV-SSD prototypes [22] and the ratification of key-value storage APIs [46]. Third, the key-value interface is more expressive than the narrow block interface and makes exposing atomicity to support transactions considerably easier. This further enhances application programmability with respect to persistence, as well as, facilitates attaining the elusive goal of syscall atomicity.

To demonstrate the effectiveness of the key-value storage interface, we design KEVIN. KEVIN consists of KEVINFS (**ke**y-**v**alue **in**terfacing **f**ile **s**ystem), which translates the user's files and their inode-equivalent metadata into key-value objects, and KEVINSSD (**ke**y-**v**alue **in**dexed **s**olid-**s**tate **d**rive), which implements a novel in-storage indexing of key-value objects in the SSD's physical address space. We observe that KEVIN has the following quantitative advantages over the traditional file system on a block SSD. First, KEVIN significantly reduces the amount of I/O transfers between the host and the device. On the other hand, a file system on a block device must access its many on-disk data structures before the user's file, incurring high I/O amplification. Second, KEVIN simplifies crash consistency without needing to employ journaling. KEVINSSD supports transactions across key-value `SET`s and `DELETE`s that make it easy to maintain a consistent and persistent state. Lastly, KEVIN is resilient to performance degradation caused by file fragmentation. As a traditional file system ages, its performance drops significantly as its data is dispersed across a fragmented block address space. In KEVIN, however, all persistent data are partially sorted and indexed through a variant of a log-structured merge (LSM) tree that prevents file fragmentation.

We implement KEVINFS in the Linux kernel v4.15 and KEVINSSD on an FPGA-based development platform. We measure our system using both microbenchmark and real-world applications, and compare it to EXT4 [49], XFS [47], BTRFS [40], and F2FS [25]. Our experiments reveal that on average, KEVIN increases system throughput by 6.2× and reduces I/O traffic by 74% for metadata-intensive workloads. These results are further accentuated when the file systems are aged and files are fragmented, highlighting the long-term effectiveness of our approach. Across eight realistic workloads (five benchmarks and three applications), KEVIN achieves 68% higher throughput on average. In summary, this paper makes the following contributions:

- We propose a novel in-storage indexing technique that eliminates the metadata management overhead of file systems by making the storage capable of indexing data.
- We prototype an SSD controller that exposes KV objects through the KV interface and optimize the LSM-tree in-storage indexing engine to efficiently service file system requests with low overhead.
- We develop a full-fledged in-kernel file system in Linux that operates over the KV interface, supporting efficient crash recovery.
- We investigate the effectiveness of KEVIN using micro and realistic benchmarks. Evaluation results show that KEVIN significantly improves I/O performance, especially under metadata-intensive scenarios.

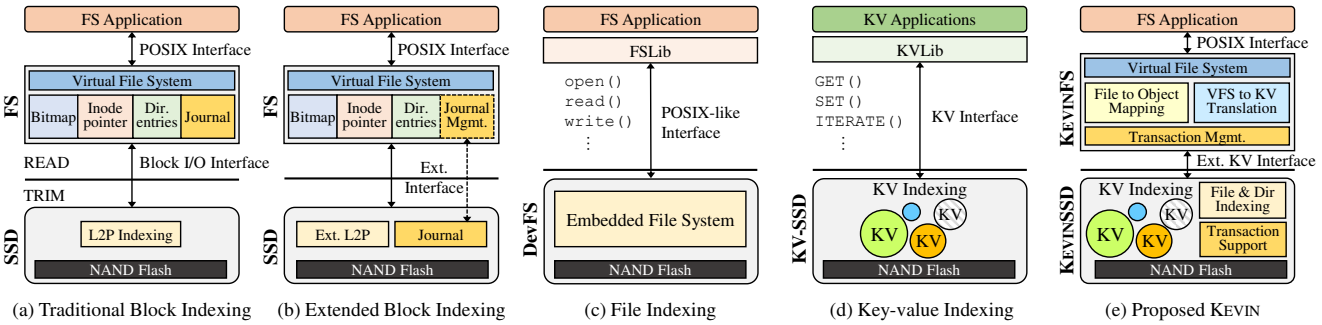
2 Background and Related Work

In this section, we review the traditional block I/O interface, and discuss how our work relates to prior studies [9, 26, 30, 55]. We then describe the basics of the LSM-tree that are fundamental to our indexing algorithm.

2.1 Traditional Block I/O Interface

Existing block storage devices expose the block I/O interface that abstracts underlying storage media as a linear array of fixed-size logical blocks (*e.g.,* 512 B or 4 KB) and provides block I/O operations. Internally, they employ a simple form of in-storage indexing to hide the unreliable and unique properties of the underlying media. HDDs maintain an indirection table to handle bad blocks [17]. Flash-based SSDs contain a flash translation layer (FTL) that maps logical blocks to physical flash pages through the logical-to-physical (L2P)

[1] A number of enterprise-grade SSDs ignore `fsync` by relying on supercapacitors to guarantee durability [52].



(a) Traditional Block Indexing (b) Extended Block Indexing (c) File Indexing (d) Key-value Indexing (e) Proposed KEVIN

Figure 2: Categories of in-storage indexing technologies

indexing table, so as to emulate over-writable media over out-of-place updatable NAND devices and to exclude bad blocks [1] (see Figure 2(a)). To virtualize files and directories over a block device, file systems maintain various on-disk data structures (*e.g.,* disk pointers, bitmaps, and directory entries). However, the management of on-disk data structures is costly, as it involves moderate extra I/O traffic, requires journaling to support consistency, and is vulnerable to fragmentation.

2.2 Review of In-Storage Indexing

Extended block I/O interface. There have been various approaches to enhancing the block I/O interface and the naive L2P-based indexing. Many have suggested custom interfaces with transactional SSDs to ensure consistency at a low cost. While specific designs differ, they commonly aim to offload a journaling mechanism to storage so that a storage controller can keep track of journaling records to avoid double-writing during journal checkpointing [9, 21, 26, 36] (see Figure 2(b)). Some have proposed an order-preserving interface and corresponding L2P indexing design to shorten I/O ordering delays for journaling [6]. Resolving fragmentation of disk pointers (*e.g.,* EXT4's extents) at the storage hardware level was presented by [16]. Those measures have alleviated specific problems (*e.g.,* journaling, ordering, and fragmentation) but have been unable to fundamentally eliminate I/O overhead associated with file-system metadata. And, since the individual strategies have specific designs, applying all of them collectively is also quite difficult.

File indexing & interface. DevFS is a local file system completely embedded within the storage hardware [23] (see Figure 2(c)), DevFS exposes the POSIX interface to a user-level application so that the application can access a file without trapping into and returning from the OS. Since all the metadata operations are performed inside the storage device, I/O stacks and communication overhead can be completely removed. However, moving the entire file system into the storage device has serious drawbacks, such as requiring costly hardware resources and providing limited file system functionalities. As discussed in [23], it is in fact difficult to run a full-fledged file system without adding large DRAM and additional CPU cores to the storage controller. This approach also limits the implementation of advanced file system features, such as snapshot and deduplication. Firmware upgrades to provide new features add maintenance costs.

Key-value indexing & interface. Kinetic HDD and KV-SSD implement parts of a key-value store engine in the storage hardware to accelerate KV clients [13] (see Figure 2(d)). KV-SSDs expose variable-size objects, each of which has a string key, and provide KV operations to manipulate objects. Samsung's KV-SSD indexes KV pairs using the hash because of its simplicity [22], but it suffers from tail latency and poor range query speed [41]. To address this, LSM-tree-based indexing is proposed [18]. Some go a step further by showing that the KV interface can be extended to support compound commands and transactions [24]. But, its FTL design was not explained in detail. Needless to say, KV-SSDs speed up KV clients by doing KV indexing on the storage side. However, since they target KV clients, existing KV interfaces and algorithms are insufficient to index files and directories. For example, Samsung's KV-SSD device based on the hash shows (*i*) slow iteration performance, (*ii*) slow sequential performance (= random), and (*iii*) slow performance on small-value KV pairs. The atomicity and durability support is limited to only a single KV object, which makes it difficult to remove file-system journaling. As a result, naively implementing file systems over KV-SSDs without fundamental design and interface changes may not promise performance improvement.

KEVIN. KEVIN is the natural extension of existing KV-SSDs. While maintaining a lean indexing architecture for a storage controller, our in-storage engine based on an LSM-tree is designed to efficiently index files and directories, together with transaction support to remove file-system journaling (see Figure 2(e)). Over such a KV storage device, we present a new POSIX-compatible file-system design that translates VFS calls and maps files and directories to KV objects. In other words, KEVIN splits the file system into the OS and the device, proposing an extended KV interface to glue the two components efficiently.

2.3 File System over Key-value Store

There have been attempts to run file systems over KV stores [19, 39]. The B^{ε}-tree and LSM-tree algorithms often used in KV stores are write-optimized, so the file system's metadata operations or small file writes are handled efficiently.

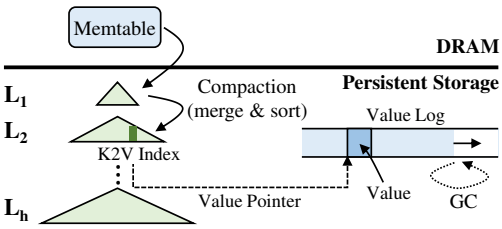


Figure 3: Overall architecture of LSM-tree

BεtrFS in particular employs full-path indexing to improve directory scanning performance. It also adopts zones [56], range deletion [56], and tree-surgery [57] techniques to improve rename and directory deletion operations. Those studies, however, still rely traditional file systems (*e.g.,* EXT4) as a data store and are based on in-kernel (*e.g.,* TokuDB [19]) or user-level (*e.g.,* LevelDB [39]) KV stores. This host-side indexing inevitably causes I/O traffic between the host and the device. The write-ahead logging (WAL) to ensure consistency with KV objects also incurs double-writes like the journaling of traditional file systems. To avoid this, BεtrFS employs late-binding journaling for sequential writes [56]. In addition, TableFS uses EXT4 as a file store to keep big files, so it suffers from fragmentation as the file system ages.

2.4 LSM-Tree Basics

We explain the basics of LSM-tree algorithms [31]. The LSM-tree, as shown in Figure 3, maintains multiple levels, L_1, L_2, ..., L_{h-1}, and L_h, where h is a tree height. Levels are organized such that L_{i+1} is T times larger than L_i. Each level contains unique KV objects sorted by the key. However, the key range of one level may overlap with those of other levels.

A KV object is first written to a DRAM-resident memtable. When the memtable becomes full, buffered KV pairs are flushed out to L_1 in persistent storage. The LSM-tree sequentially writes buffered KV objects to free space in L_1. Once L_1 becomes full, KV pairs of L_1 are flushed out to L_2 and similarly, L_i is flushed out to L_{i+1} when L_i is full. To satisfy the tree property, when flushing out L_i to L_{i+1}, the LSM-tree should perform compaction that merges and sorts the KV objects of L_i and L_{i+1}. The compaction requires many I/Os since it has to read all KV pairs from two levels, sort them by the key, and write sorted KV pairs back to the storage.

To reduce compaction costs, one suggests managing keys and values separately [28]. It appends a value of a KV object to a value log; only a key and a value pointer locating a corresponding value in the log are put into the tree. In this paper, a pair of <key, value pointer> is called a *K2V index*. Because object values do not need to be read during compaction, compaction costs can be greatly reduced, especially when a value is larger than a key. The value log contains obsolete values that must be reclaimed by garbage collection.

For retrieving a KV object, the LSM-tree may look up multiple levels, which involves extra reads due to the fact that the key ranges of the levels can overlap. If a candidate KV object fetched from L_i is not matched with a wanted one, we should move on to L_{i+1} and look up another candidate. To reduce reads for level lookups, bloom filters are used. According to [11], the number of extra reads can be reduced to around one. Once a desired KV object is found, the LSM-tree returns a value to the client because a key and its value are read together. If keys and values are separated, another read is required to retrieve its value stored in the value log.

3 Overall Architecture of KEVIN

This section explains the architecture of KEVIN, focusing particularly on its indexing schema to offload file-system metadata (inode/data bitmaps, disk pointers, and directory entries) to storage. We design two major components of KEVIN, KEVINFS and KEVINSSD, so that they have specific roles: (*i*) the mapping of files and directories to KV objects at the file-system level (§3.1); (*ii*) the indexing of KV objects in flash using the LSM-tree at the storage level (§3.2).

Before explaining the details of our system, we explain the KV interface in Table 1. KEVINSSD exports basic KV operations, `SET`, `GET`, and `ITERATE`, to read, write, and iterate over KV objects. `SET` and `GET` also support partial reads and writes that are useful for dealing with micro reads and writes on a large object. KEVINFS invokes `ITERATE` repeatedly to retrieve KV pairs whose keys are lexicographically equal to or greater than a given `pattern`. KEVINSSD supports transaction commands, `BeginTX`, `AbortTX`, and `EndTX`, exploited by KEVINFS to ensure file-system consistency. A (range) deletion command, `DELETE`, is included to support object deletion or truncation. The length of an object key is variable, but is limited to 256 B. Technically, there is no limit to a value size.

3.1 Mapping of File and Directory

KEVINFS uses only three types of KV objects: *superblock*, *meta*, and *data* objects. A superblock object keeps file system information. While a meta object stores attributes of a file or a

KV Command	Description
`GET(TID,key,off,len)`	Retrieve a value given `key`; if `off` and `len` are given, read `len` bytes of data at offset `off` from a value of `key`
`SET(TID,key,off,len,val)`	Set `key` to hold data `val`; if doesn't exist, create a new one; partially update a value given `off` and `len`
`DELETE(TID,key,off,len)`	Delete an object of `key`; truncate part of a value given `off` and `len`
`ITERATE(TID,pattern,cnt)`	Iterate over objects and return at most `cnt` objects that are lexicographically equal to or greater than `pattern`
`BeginTx(TID)`, `EndTx(TID)`, `AbortTx(TID)`	Start a new transaction with `TID`; commit the transaction; abort the transaction, discard changes (see §5.1)

Table 1: Key KV commands supported by KEVIN

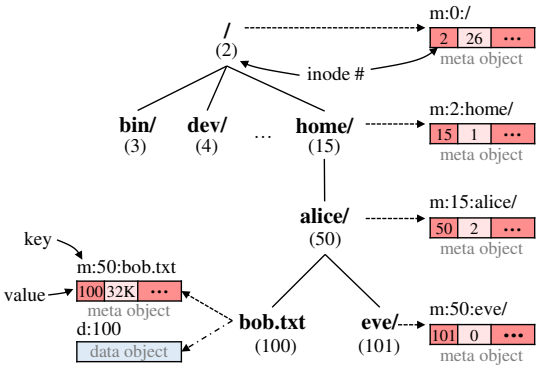


Figure 4: Meta and data objects

directory (*e.g.,* an inode number, size, and timestamps), a data object holds file data. The sizes of a superblock object and a meta object are 128 B and 256 B, respectively. Conversely, a data object can be as large as a file size.

Figure 4 illustrates how files and directories are stored as the form of KV objects. A regular file consists of a pair with one part meta object and the other data object. In contrast to a file, a directory only has a meta object to keep its attributes. For a regular file, the size field of a meta object represents a file size; for a directory, it is the total number of subdirectories and files. All objects are retrieved (`GET`), stored, or updated (`SET`) by KV commands with unique keys. Directory traversals are supported by `ITERATE` as well (explained in detail later).

For assigning a key of an object, KEVIN uses two inode-based key naming rules. **Rule #1:** a meta object key is a combination of (*i*) a prefix '`m:`', (*ii*) an inode number of a parent directory, (*iii*) a delimiter '`:`', and (*iv*) a file or directory name. **Rule #2:** a data object key is a combination of (*i*) a prefix '`d:`' and (*ii*) an inode number of a file. Our naming rules are based on [39], but is extended to deliver the semantics of KV objects so that the storage hardware can index them more efficiently (see §3.2).

Figure 4 shows an example directory tree and associated KV objects. Consider a file `bob.txt` in a directory `/home/alice/`. The inode numbers of `/home/alice/` and `bob.txt` are `50` and `100`, respectively. According to the rule #1, the meta object key is `m:50:bob.txt`. Similarly, following the rule #2, the data object key is `d:100`. As another example, consider a directory `eve/` in `/home/alice/`. A directory has a single meta object only, so a meta object whose key is `m:50:eve/` exists.

KEVINFS has no directory entries, but a list of files and directories belonging to a specific directory can be retrieved by using `ITERATE`. To list up files and directories in `/home/alice/` whose keys start with `m:50:`, KEVINFS creates a new iterator `ITERATE(m:50:,2)` and sends it to the storage, which then returns meta objects with the prefix `m:50:` (*e.g.,* `bob.txt` and `eve/` in Figure 4). To prevent too many objects from being fetched at once (which might take so long), we can specify the maximum object count `cnt` in the `ITERATE` command. In this example, `cnt` is 2, representing the number of subdirectories and files in `/home/alice/`. Traversing an entire file-system tree is easily implemented. The inode number of '`/`' is fixed to 2. KEVINFS retrieves all the files and directories in '`/`' with `ITERATE(m:2:,26)`. By repeating the above steps for directories , it builds up the entire tree.

To efficiently handle small files, KEVINFS packs attributes and data of a file in a meta object together if their size is smaller than 4 KB. This reduces I/Os since a small file can be read or written by one `GET` or `SET` to its meta object.

As an alternative to the inode-based indexing, one might suggest using the full-path indexing [19]. This improves scan performance when using a KV store based on sorted algorithms (*e.g.,* B^{ε}-trees), as it globally sorts the entire file-system hierarchy. While this is beneficial on devices with high seek time such as HDDs, on devices with fast random access like SSDs, its benefits are diminished. On the other hand, the inode-based indexing shows good performance on operations other than directory scans and offers fast directory renaming without techniques such as zones [56] or tree-surgery [57]. Especially as KEVINSSD performs more efficiently when key lengths are short (see §3.3), the inode-based indexing that has shorter key lengths is a more appropriate choice.

3.2 Indexing of KV Objects

KV objects exposed to the file system are managed by our in-storage indexing engine, KEVINSSD, which makes use of LSM-tree indexing. KEVINSSD maps KV objects to the flash, allocating and freeing flash space, and handles read and write requests on objects which are usually done by an FTL. In our system, the FTL only does simple tasks (*e.g.,* bad-block management and wear-leveling). The hardware resources (*e.g.,* CPU cycles and DRAM) saved by disabling such FTL features are used to run our indexing algorithm.

Figure 5 shows the architecture of KEVINSSD. For each level, it maintains a tiny in-memory table (48 MB DRAM for 1 TB SSD) to keep track of KV objects in the flash. Each entry of the table has <start key, end key, and pointer>, where a pointer points to the location of a flash page that holds KV objects; start and end keys are the range of keys in the page. Those key ranges can be overlapped on multiple levels. For fast search operations, all entries are sorted by start keys.

KEVINSSD manages the keys and values of meta and data objects separately. This is a reasonable choice because a key size is much smaller than its value size. This is even true for a small meta object whose value size is 256 B. According to our analysis, the average length of a meta-object key is 32 B, which is 8× smaller than its value. Since keys and values are separated, only K2V indices (*i.e.,* <key, value pointer>) for objects are stored in flash pages, called *key-index pages*, which are separated from their values in other flash pages. Meta and data objects begin with a different prefix ('`m:`' or '`d:`'), so their K2V indices are sorted in different pages.

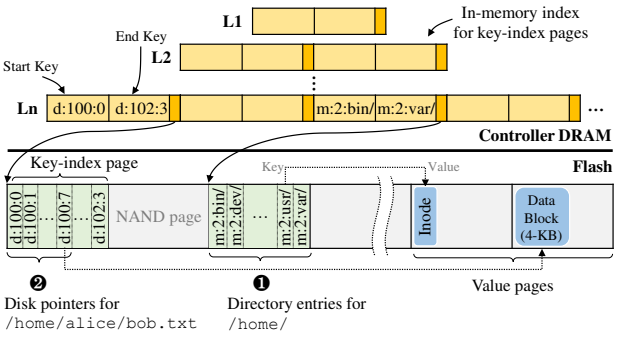


Figure 5: Layout of KV objects in KEVINSSD

Meta object indexing. A meta object key corresponds to a file or directory name in typical file systems, while meta object value is equivalent to an inode. Key-index pages for meta objects thus contain K2V indices, each of which is a pair of <meta object key, value pointer>. This is similar to a directory entry, <file or directory name, inode #>, in a directory file. According to the naming rule #1, K2V indices that belong to the same parent directory are sorted by the parent's inode number and thus are likely to be packed into the same key-index pages (see ❶ in Figure 5).

A list of files and directories belonging to a specific directory can be retrieved quickly if associated K2V indices are fully sorted. To get directory entries in `/`, for example, KEVINSSD requires one flash read (or a few if directory size is huge). However, as mentioned in §2, to reach a wanted key-index page, KEVINSSD should look up multiple levels of the tree. Moreover, if K2V indices are fragmented across multiple levels, more than one flash read is required to build up a complete directory list. We explain how we mitigate this problem in the storage (§3.3) and the file system levels (§4).

Directory entries are updated efficiently. Existing file systems read and write a 4 KB block(s) to modify a list of directory entries. In KEVINSSD, just by writing (`SET`) or removing (`DELETE`) a meta object, we can update directory entries in directories. This removes the necessity of maintaining directory files, thereby eliminating data movement costs.

Data object indexing. In contrast to a meta object, a data object can be very large. Indexing a large object (*e.g.,* 1 GB) as the form of a single KV pair incurs high I/O overhead when a small part of it is read or updated. For example, to update only 512 B of data, KEVINSSD has to read an entire object, modify it, and write it back to the flash, updating its index in the tree. To avoid this, KEVINSSD splits a data object into 4 KB subobjects with unique suffixes and manages them as if they are independent KV pairs. For `/home/alice/bob.txt` whose size is 32 KB, its data object is divided into eight 4 KB subobjects with different suffixes, '`d:100:0`', '`d:100:1`', ..., and '`d:100:7`', in storage. If a small part of a huge object is retrieved or updated, only the corresponding subobject needs to be read from or written to the flash. Please be advised that there is no additional indirection (or index) for subobjects because subobject keys are decided by file's offset.

Since subobject keys and their values are separated, key-index pages hold K2V indices, each of which is a pair of <subobject key, pointer>. As one might notice, a K2V index is like a disk pointer (or extents in EXT4) pointing to a data block in existing file systems. According to the rule #2, K2V indices are sorted by file's inode number and by suffix numbers. Therefore, K2V indices belonging to the same data object (*i.e.,* the same file) tend to be packed in the same key-index pages (see ❷ in Figure 5).

To retrieve 4 KB data from a data object, KEVINSSD should look up levels to find a desired key-index page. Once it is found, KEVINSSD can read a K2V index from the flash with one page read. Then, actual data are read by referring to its pointer. Other K2V indices read together are cached in the controller's DRAM (see §3.3). This reduces lookup costs for future requests. This indexing mechanism is similar to the management of disk pointers (*e.g.,* an extent tree in EXT4). Existing file systems maintain index blocks that contain pointers only, where each pointer points to a data block or another index block. Before reading file data, index blocks must be loaded from a disk.

In KEVINSSD, looking for a K2V index for reading data is done in storage. The update of K2V indices for a data object is done by writing or deleting a data object via `SET` and `DELETE`. Compared to typical file systems that read and write a 4 KB block(s) to retrieve and to update disk pointers, KEVINSSD does not involve any external I/Os to index file data.

3.3 Mitigating Indexing Overhead

As mentioned in §3.2, putting the LSM-tree indexing onto the storage hardware causes extra I/Os, which never happen in typical FTLs using a simple L2P indexing table (which is entirely loaded in DRAM). We introduce three main causes that create internal I/Os and explain how we solve them (see Figure 6). Note that garbage collection occurs both in KEVIN and existing SSD controllers, so it is not explained here.

Compaction cost. Compaction is an unavoidable process and may involve many reads and writes [28]. KEVINSSD manages meta and data objects in a manner that minimizes compaction I/Os by separating keys and values. Particularly, our inode-based naming policy that assigns short keys to data objects lowers the compaction cost because it enables us to pack many subobject keys into flash pages. We go one step further by compressing K2V indices for data objects. Subobject keys have regular patterns (*e.g.,* '`d:100:0`', '`d:100:1`', ...), so they are highly compressible even with naive delta-compression requiring negligible CPU cycles. This reduces the amount of data read and written during compaction. According to our analysis with write-heavy workloads, the write amplification factor (WAF) of the compaction was less than 1.19× under the steady-state condition (see §6.2).

Level lookup cost. The LSM-tree inevitably involves multiple lookups on levels until it finds a wanted KV object (see

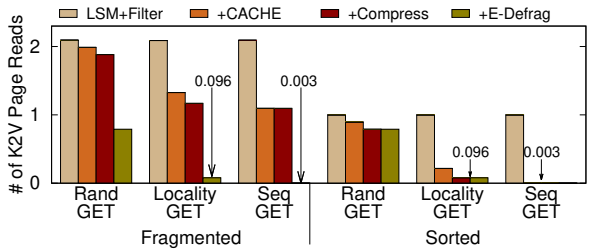


Figure 6: The number of reads per KV request to retrieve a key-index page. The LSM-tree with bloom filters is our default setting. We add each optimization technique one by one to understand their impact. The size of bloom filters is set to 6.5 MB for 40M objects. The cache size is 110 MB.

§2). To avoid useless lookups, KEVINSSD employs small bloom filters. It reduces the number of extra reads for level lookups to around one [11]. To further reduce lookup costs, it also caches popular K2V indices in DRAM. SSDs usually have a large DRAM (*e.g.,* 1 GB for 1 TB SSD) to keep an L2P table, but this large L2P mapping table is unnecessary for KEVINSSD. This enables us to use large DRAM for caching. To increase an effective DRAM size, KEVIN maintains cached K2V indices in the compressed form. To make it searchable, in between compressed indices, we add uncompressed keys sparsely which can then be used as a pivot index for binary searching. This optimization shows its strength with large files. For example, to index a 10 GB file without compression, 45 MB are required for indexing KV pairs, but with compression, only 10.8 MB memory is needed.

Fragmented tree cost. The LSM-tree allows each level to have overlapped key ranges with other levels. Therefore, K2V indices belonging to the same parent directory or file can be fragmented across multiple levels, even they have the same prefix. To retrieve a full list of directory entries or disk pointers, multiple flash pages on different levels must be read. This problem is implicitly resolved by compaction that merges and sorts K2V indices in adjacent levels. KEVIN also provides an offline user-level tool that explicitly triggers compaction in storage. Unlike traditional tools (*e.g.,* `e4defrag` [14]), this does not involve moving the entire file system's metadata and data and is thus much more efficient.

Figure 6 shows the impact of optimization techniques in reducing indexing overhead. For each KV request, we counted the number of page reads required (*i*) to find a key-index page in the LSM-tree and (*ii*) to read that page from flash. The I/O cost of reading a value was not included here. Over KV objects that were fragmented (*i.e.,* unsorted) or fully sorted, we ran three types of queries: random `GET` (= point-query), 90:10 localized `GET` (= point-query), and sequential `GET` (= range-query). Sequential `GET`s over fully-sorted KV objects required almost zero cost to read a key-index page. This is because after the first miss on a specific key-index page, following KV requests were hit by the cached indices. Caching KV indices were also useful when `GET` requests were localized. Regardless of the distribution of KV objects, random `GET`s suffered from extra reads, but even in the worst case, they required about two reads. This is because, with bloom filters, the number of page reads that happen while searching for a key-index page in the tree is theoretically limited to around one, on average [44].

Syscalls	KEVINFS	EXT4
`mkdir`	SET(MO)	W(BB + IB + I + DE)
`rmdir`	DELETE(MO)	W(BB + IB + DE)
`creat`	SET(MO)	W(IB + I + DE)
`unlink`	DELETE(MO + DO)	W(BB + IB + DE)
`setattr`	SET(MO)	W(I)
`write`	SET(DO)	W(BB + D)
`open`	GET(MO)	R(I)
`lookup`	GET(MO)	R(DE + I)
`read`	GET(DO)	R(D)
`readdir`	ITERATE(MO)	R(DE + I)

Table 2: I/O operations of KEVIN and EXT4 for basic syscalls. (MO: meta object, DO: data object, BB: block bitmap, IB: inode bitmap, I: inode, DE: directory entry, and D: data block)

Even with such optimizations, KEVINSSD exhibits slightly slower read performance than block storage devices that do not suffer from any extra I/Os for indexing. However, our entire system exhibits much higher performance than existing systems thanks to the reduction in metadata I/Os. Moreover, while metadata I/Os on existing file systems increase as it ages and gets fragmented, KEVINSSD's indexing cost is maintained constantly through regular compaction of the LSM-tree in storage and other optimizations.

4 Implementing VFS Operations

We describe how KEVINFS implements VFS operations using the KV interfaces. KEVINFS is a POSIX-compatible in-kernel file system and implements 86 out of 102 VFS operations. We summarize the types of I/O operations to handle major file syscalls in Table 2, comparing them with EXT4.

Handling write syscalls. All the write-related syscalls can be handled by two KV commands, `SET` and `DELETE`. It is clear that KEVINFS requires fewer I/O operations than EXT4. This benefit stems from the fact that KEVINFS does not need to modify on-disk metadata. Taking the example of `unlink`, KEVINFS issues two `DELETE` commands to remove a meta object and a data object (which are associated with the file to be deleted) from storage. On the other hand, EXT4 has to update data and inode bitmaps to return a data block as well as an inode. EXT4 needs to update directory entries to exclude the deleted file from the directory.

KEVINSSD does not involve many internal I/Os for `SET` and `DELETE`. `SET` first buffers a KV object in the memtable and then appends to the flash later, leaving the old version if it exists. `DELETE` internally involves a small write to leave a tombstone (4 B) in the tree. Outdated objects (overwritten by `SET`) and deleted objects are persistently removed during compaction, which is not expensive in our design as we manage keys and values separately.

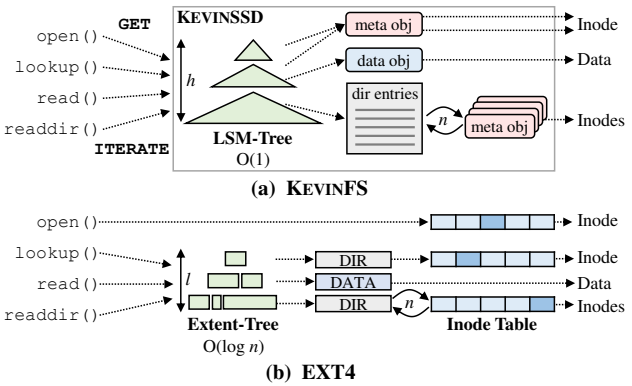


Figure 7: Handling of read syscalls of KEVINFS and EXT4

Handling read syscalls. Read-related syscalls can be implemented by two KV commands, `GET` and `ITERATE`. Regardless of the type of data being accessed, KEVINFS needs to send `GET` or `ITERATE` to a designated meta or data object as shown in Figure 7(a). `open`, which retrieves an inode of a file, can be implemented as `GET` to a meta object. `lookup` is the same as `open` in that, given a full path name (*e.g.,* `/home/alice/`), it retrieves inodes of directory components (*e.g.,* '`/`', '`home/`', and '`alice/`') by sending `GET`s to meta objects. Reading data from a file is also translated into `GET` to a data object. Finally, `readdir` corresponds to `ITERATE`, which retrieves a set of meta objects (*i.e.,* inodes) that belong to the same parent directory.

While the LSM-tree is used as a unified indexing data structure to service all the read-related syscalls in KEVIN, EXT4 relies on several on-disk data structures: an inode table, an extent tree that indexes disk pointers, and a directory file that holds directory entries and their inode numbers (see Figure 7(b)). KEVIN and EXT4 should be comparatively analyzed further because the two systems operate dissimilarly over different data structures. But, KEVINFS benefits from its in-storage indexing; all the I/Os associated with the LSM-tree are performed in storage without any external data transfers.

When opening a file, EXT4 fetches an inode from the inode table by using its inode number as an index. EXT4 requires a 4 KB block read and is faster than KEVINFS that has to look up the LSM-tree before reading an inode.

For `lookup` and `read`, KEVINFS needs to look up the LSM-tree to get locations of meta or data objects (*i.e.,* key-index pages). Similarly, EXT4 needs to search the extent tree to find disk pointers that locate disk blocks for a directory or regular file. Both cases may involve extra reads from the disk. To skip the tree search step for small files, EXT4 embeds a few disk pointers in an inode. KEVINFS cannot avoid the tree search. However, it does not require reading a directory file during `lookup`, and the data of a small file is preloaded when its meta object is read. Thus, for `lookup` and small `read`, the two systems exhibit similar performance.

If a file or a directory is huge, the tree search cost could be high. In KEVINFS, the worst-case I/O cost of looking up the LSM-tree is $O(h)$, where h is the tree height. However, as shown in Figure 6, even under random I/Os, the average I/O cost is not as high as two reads thanks to bloom filters [11]. In EXT4, the worst-case I/O cost of the extent tree is $O(l)$, where l is the height of the tree ($l = 5$ by default). The average I/O cost is $O(log\ n)$, where n is the number of extents for a file which actually decides the tree height. If a file is not fragmented, n is close to 1, and thus the tree search requires less than two reads. However, if it is severely fragmented, the I/O cost could be more than two reads.

Besides the tree lookup cost, KEVINFS has another benefit in that it is never logically fragmented. In EXT4, once a file is fragmented, many pieces of file data are scattered across non-continuous logical blocks. In this case, even when the file is sequentially read, EXT4 has to issue many read requests to the disk [10]. As reported by [16], it badly affects I/O throughput. In KEVINFS, no logical fragmentation happens because each file is represented as an object, not a set of logical blocks. Hence, KEVINFS can always perform sequential reads in big granularity. Also, as explained in §3.3, KEVINSSD shows high sequential I/O performance on subobjects with index caching and compression. As a result, EXT4 generally provides good performance when a file is continuously allocated, but KEVINFS is more resistant to fragmentation.

Finally, `readdir` requires retrieving a full list of directory entries to read the associated inodes. EXT4 offers different performance depending on the degree of inode table fragmentation. If the inodes are allocated together and thus are stored in the same blocks, only few block I/Os are needed to retrieve them. However, if they are highly fragmented, EXT4 suffers from high I/O overhead. In KEVINFS, the inodes (*i.e.*, *values* of meta objects) pointed to by the directory entries are scattered across multiple pages (see ❶ in Figure 5). This inevitably degrades `readdir` performance. To mitigate this, KEVINFS uses a simple tweak that rewrites meta objects to the disk. When meta objects retrieved by `ITERATE` are evicted from the page cache, KEVINFS rewrites them to the disk even if some of them are clean. All of the meta objects that are evicted together are likely to be written to the same flash pages so that the next time KEVINFS can retrieve them quickly without multiple page reads. We plan to study a way to sort meta objects inside KEVINSSD without explicitly rewriting.

5 Crash Consistency

We describe how KEVIN implements transactions to maintain consistency. KEVINFS issues fine-grained transactions by tracking dependency among KV objects so that they are updated atomically (see §5.1), and KEVINSSD supports transaction commands exploited by KEVINFS (see §5.2).

5.1 Maintaining Consistency in KEVINFS

Although an ideal file system would immediately persist data upon a write without any consistency problems, current file

systems follow a compromised model for better performance. That is, file systems provide an explicit interface to the users (*i.e.,* `fsync`) by which users can request a barrier across updates or immediate durability enforcement whenever needed. In addition, file systems such as EXT4 maintain a global transaction comprising all associated blocks with write requests during a time window, and the flush daemon atomically persists them to storage through the journaling. This mechanism prevents the out-of-execution and/or buffering of write that may lead to an inconsistent state for the file system.

KEVIN makes data durable through both user-initiated `fsync` and the flush daemon, but without the overhead associated with the journaling mechanism. This is achieved by KEVINSSD supporting fine-grained transactions.

KEVINFS only builds a transaction associated with dependent KV objects and simply transfers the information to the underlying storage. KEVINSSD then materializes given transactions to the physical medium with an SSD-internal technique (see §5.2). To this end, we extend the KV interface to support three transaction commands: `BeginTX`, `EndTX`, and `AbortTX`. Below is an example of a transaction that manages the KV objects associated with `unlink` in Table 2. KEVINFS can instruct the storage to remove meta and data objects atomically by wrapping KV commands in the same transaction:

```
BeginTX(TID);
DELETE(TID,m:4:c.txt); /* data object */
DELETE(TID,d:5); /* meta object */
EndTX(TID);
```

KEVINSSD guarantees atomicity and durability for a transaction. To ensure file-system consistency, KEVINFS preserves the order between dependent transactions. As a result, KEVINFS ensures the same level of reliability as other journaling file systems, and also offers the following desired properties.

Transaction disentanglement. The performance of a file system suffers from a phenomenon known as transaction entanglement that flushes the entire global transaction when `fsync` is requested for only a part of the buffered data. This not only increases the `fsync` latency, but lowers the effect of write buffering. Some attempted to resolve this issue by splitting a transaction into the smaller ones by files or by subtrees [29, 32]. However, it could not be effective in practice because the transaction disentanglement is impossible when data is shared across transactions. Typical file systems engrave small metadata within a fixed-sized block (*e.g.,* inode/bitmap blocks), and thus chances are high that the metadata updates in a different context happen to the same block.

In contrast, KEVINFS does not maintain any on-disk metadata shared by different files unless they are adjacent in the file system tree (*e.g.,* a parent directory and a file). This nature makes transaction disentanglement easy and likely more effective. KEVINFS basically maintains a single running transaction containing all pending KV commands and sends them at once through periodic flush daemon (a default period is 5s). However, upon `fsync`, KEVINFS forks a small transaction

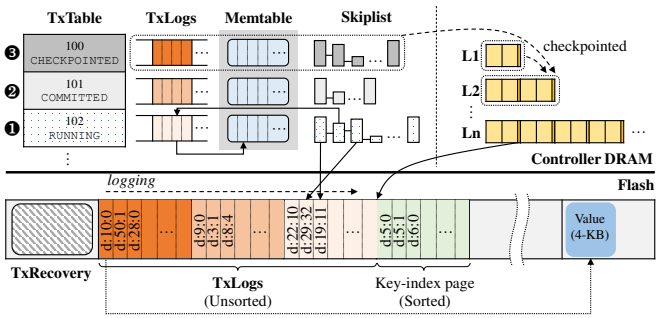


Figure 8: Transaction management of KEVINSSD

that only includes the KV objects associated with the fsynced file, thereby achieving short latency.

Syscall atomicity guarantee. Current journaling file systems do not ensure the atomicity of a syscall. Because the transaction size is limited by the remaining journal size, even a single syscall can be split into multiple transactions in some cases [51]. This is rare but possible and thus the user-level applications should employ another technique (*e.g.,* user-level journaling) to ensure an atomic write over the file system. KEVINFS has no such limitation and thus enforces the atomicity for each syscall by assuring all of the associated KV objects reside in the same transaction.

5.2 Transaction Processing in KEVINSSD

We now explain how KEVINSSD supports transaction commands. Our design is essentially based on journaling but we further optimize it to perform well with KEVINSSD.

Transaction management. Figure 8 shows the transaction management in KEVINSSD. We employ three data structures: a transaction table (*TxTable*), transaction logs (*TxLogs*), and a recovery log (*TxRecovery*). The TxTable keeps the information of transactions, while the TxLogs keep K2V indices of transaction objects. The TxLogs are stored either in the DRAM or in the flash. They are also used to keep track of K2V indices committed to L_1 in the tree. The TxRecovery is used to recover or abort transactions during the recovery.

When `BeginTx(TID)` comes, KEVINSSD creates a new entry in the TxTable, where each entry keeps a `TID`, its status, and locations of K2V indices associated with the transaction. Many transactions can be activated simultaneously as there exist multiple entries in the table. Initially, the status of the transaction is `RUNNING`, which means that it can be aborted in the event of a crash (see ❶ in Figure 8). When subsequent commands belonging to the transaction arrive, KEVINSSD keeps KV indices in the DRAM-resident TxLogs and buffers associated values in the memtable. Once the TxLogs or the memtable becomes full, KV indices or values are logged into the in-flash TxLogs or the flash. All of them are not applied to the LSM-tree yet as they can be aborted. When `EndTx(TID)` is received, the associated transaction is committed, and its status is changed to `COMMITTED` (see ❷). KEVINSSD then notifies KEVINFS that the transaction is committed. Even

though some KV indices and values can still be buffered in DRAM (*i.e.,* the in-memory TxLogs and the memtable), KEVINSSD ensures persistence by using a capacitor. Finally, committed KV indices should be reflected to the permanent data structure, the LSM-tree, through a checkpoint process.

Unfortunately, the checkpoint cost is high because committed KV indices should be inserted into L_1 in the LSM-tree. Recall that some of KV indices are stored in the in-flash TxLogs and are unsorted because they are logged in their arrival order. Thus, the checkpoint process involves extra I/Os and sorting overhead. We relieve this cost by treating committed TxLogs as part of L_1 and delaying the writing of their KV indices to the tree until the compaction between L_1 and L_2 happens. When the compaction is triggered, KV indices in the TxLogs and L_1 are flushed out to L_2 together. In this way, we can skip writing KV indices to L_1. To quickly look up KV indices in the TxLogs (which are unsorted), KEVINSSD temporarily builds a small skiplist to index K2V pairs in the TxLogs. The sorted nature of the skiplist also makes it easy to apply KV indices into the tree during the compaction.

Later, once associated KV indices are checkpointed to L_2 through the compaction, the transaction status is changed into `CHECKPOINTED` (see ❸), and the associated TxLogs and the TxTable entry occupied by them are reclaimed.

Recovery. The TxTable, buffered K2V indices, and values must be materialized to the flash regularly or when a certain event happens. In our design, KEVINSSD materializes them when a sudden crash is detected. While power is being supplied by a capacitor, it flushes out buffered K2V indices to the in-flash TxLogs and buffered values to the flash. The TxTable is updated to point to the in-flash TxLogs and is then appended to the TxRecovery. Two specific flash blocks (*e.g.,* blocks #2 and #3) are reserved for the TxRecovery and are treated as a circular log. When a system reboots, KEVINSSD scans the TxRecovery, finds the up-to-date TxTable, and checks the status of each transaction. If a transaction was already committed, it means that KEVINSSD persistently wrote KV objects to the flash before. Associated K2V indices are thus pushed into the skiplist to be searchable. The `RUNNING` transactions are aborted, and associated resources are reclaimed.

KEVINSSD supports the same level of crash consistency as EXT4 with the ordered mode, but requires much smaller I/Os by avoiding double writes. Moreover, by leveraging capacitor-backed DRAM in the controller, it further reduces the overhead of flushing out KV objects and lowers the delay of marking journal commits. The DRAM-resident TxLogs is 2 MB in size in our default setup, so a large capacitor is not required.

6 Experiments

We present experimental results on KEVIN. We seek to answer the following questions: (*i*) Does KEVIN provide high performance under various workloads, in particular, metadata intensive ones? (*ii*) How much data movement between the host and the SSD can be reduced? (*iii*) Does KEVIN provide high resistance to fragmentation? (*vi*) Does KEVIN provide benefits over existing file systems based on KV stores?

6.1 Experimental Setup

All the experiments are performed on a server machine equipped with Intel's i9-10920X CPU (12 cores running at 4.6 GHz) and 16 GB DRAM. The Linux kernel v4.15.18 is used as the operating system. Our SSD platform is based on a Xilinx VCU108 [53] equipped with a custom flash card providing 2.4 GB/s and 860 MB/s throughputs for reads and writes, respectively. The total SSD capacity is set to 128 GB. The FPGA contains controller logic to manage NAND chips and to provide the PCIe interface to interact with the server.

Our SSD platform does not have a CPU and has a similar architecture to an open-channel SSD [5]: it runs the FTL software on the host system. To emulate the limited resources of an SSD controller on an x86 host, we implement FTLs in the guest Linux OS in QEMU/KVM, completely isolated from the host that runs file systems. We assign 4 cores to the guest. 128 MB DRAM (0.1% of the SSD capacity [42, 43]) is assigned to the guest, while the rest is used by the host. The interface throughput between the host and the guest is about 8 GB/s, which is similar to that of PCIe 4.0 x4. Our default setup is biased towards existing systems considering its high interface throughput. Our system setting has a limitation. LSM-tree's compaction requires high computation power, but can easily be accelerated by FPGA or ASIC [18, 58]. Owing to the lack of those accelerators on our host system, we use the Intel i9 CPU as a sorting accelerator.

We implement two FTL schemes, the page-level FTL that uses the simple L2P indexing and the proposed KEVINSSD. The page-level FTL uses 128 MB DRAM for an L2P mapping table. KEVINSSD uses 6.5 MB of DRAM for bloom filters, 2 MB for TxLogs, 1 MB for memtables, 6 MB for in-memory index, and 112.5 MB for caching popular entries. We compare KEVINFS with four kernel file systems, EXT4 with the ordered mode, XFS, BTRFS, and F2FS.

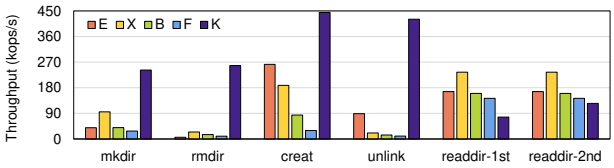
6.2 Experimental Results

We evaluate KEVIN using micro-benchmarks in §6.2.1 and carry out experiments with realistic workloads in §6.2.2. Performance analysis of aged file systems is presented in §6.2.3. The benefits of in-storage indexing are analyzed more deeply in §6.2.4. In graphs, EXT4, XFS, BTRFS, F2FS, and KEVIN are abbreviated as 'E', 'X', 'B', 'F'. and 'K', respectively.

6.2.1 Results with Micro-benchmarks

We conduct a set of experiments using three types of micro-benchmarks: metadata-only, small-file, and data-only workloads, all of which have different file/directory access patterns.

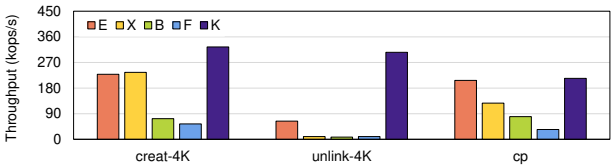
Metadata-only workloads. They create and delete a large number of empty files and directories. We use `creat`, `mkdir`, `unlink`, and `rmdir` from Filebench [48] that perform intensive updates of on-disk metadata, but do not involve any I/Os



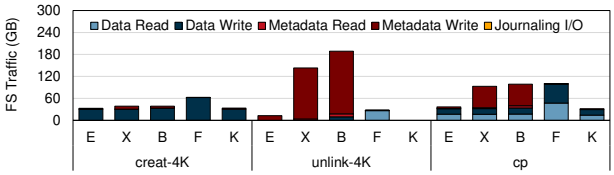
(a) Throughput

(b) File system traffic

Figure 9: Metadata intensive workloads



(a) Throughput



(b) File system traffic

Figure 11: Data & metadata workloads

on file data. A total of 8M files and directories are created and deleted. This is almost the maximum number of files and directories that can be created with EXT4's default configuration on a 128 GB disk. We also run `readdir` that iterates over a large number of directories. 4M files are stored in 800K directories. To generate sufficient I/Os, we run 16 Filebench instances in parallel. All of the file systems are initially empty.

Figure 9(a) shows results. KEVIN outperforms other systems by 6.2× on average. Aside from `readdir`, KEVIN achieves up to 43.8× better I/O throughput than EXT4. This is because KEVIN eliminates almost all of the metadata I/O traffic. Figure 9(b) depicts the amount of data moved between the host and the SSD. Compared to the existing file systems, KEVIN involves tiny data movements because it only needs to deliver small-sized KV commands.

For `readdir`, KEVIN performs poorly for the first run because it requires many reads to retrieve values (*i.e.,* inodes) from the flash pages. For its second run, KEVIN shows improved performance (from 75kops/s to 120kops/s). As explained in §4, KEVINFS rewrites a group of meta objects fetched by `ITERATE` to store them in the same flash pages, hoping that it reduces in-storage reads in the future. This optimization can increase the eviction cost slightly, but it is imperceptible to users as the I/O traffic incurred by rewrites is low and is handled in the background. Note that before the second run, we empty the inode and dentry caches to get rid of the impact of cached metadata.

KEVIN incurs internal I/Os to manage LSM-tree indices in storage, which can be categorized into three types: compaction I/Os to merge and sort K2V indices, tree lookup reads to find key-index pages, and garbage collection (GC) I/Os

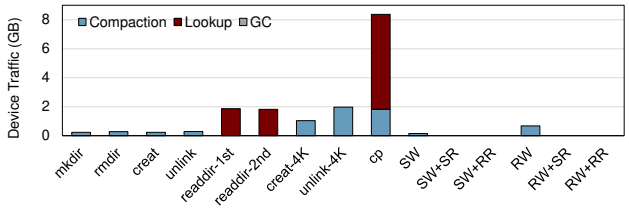
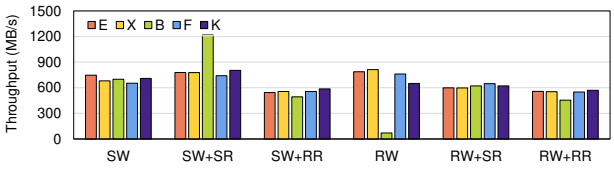


Figure 10: KEVIN I/O overheads on micro-benchmarks

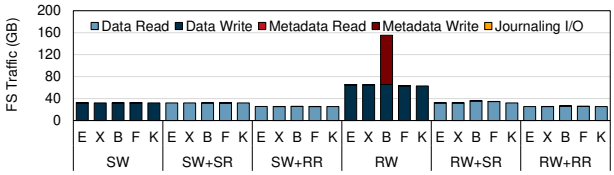
to reclaim free space. As illustrated in Figure 10, extra I/Os to manage the LSM-tree are negligible in comparison with metadata overhead in other file systems (see Figure 9(b)). Since our experiments are conducted under clean file systems, compaction and GC I/Os are almost zero. We analyze their impacts on performance in §6.2.3.

Small-file workloads. They include three scenarios: the creation (`creat-4K`) and deletion (`unlink-4K`) of 4 KB small files, as well as a copy (`cp`) of many small files. All of them create lots of data traffic on both metadata and file data. We create and delete 8M files and copy 4M files. Figure 11 shows experimental results. KEVIN exhibits the best performance when creating and removing small files thanks to its low metadata overhead. However, for `cp`, KEVINFS shows similar performance to EXT4. We observe that KEVIN shows high write throughput for small files, but the throughput of reading small files to copy is slow and becomes a bottleneck. This is owing to the relatively high tree lookup overheads. For our experiments, we run 16 Filebench instances in parallel, which cause random file reads. This eventually results in random meta object lookups on the KEVINSSD side. Moreover, as a small 4 KB file contains one subobject, KEVINSSD's compression optimizations are not effective. This is the reason why the number of reads to find key-index pages (tree lookup) is relatively high for `cp` in Figure 10. Even worse, while 4 KB file data is slightly large to be embedded in a meta object in KEVINSSD, EXT4 can directly locate a disk block where data is stored by referring to disk pointers in inodes, thereby incurring no extents lookup.

Data-only workloads. To assess how efficiently KEVIN handles a large file, we create a 32 GB file and run various I/O patterns using the FIO tool [4] on it. We first measure sequential and random write throughputs. For measuring sequential write (`SW`) throughput, we run a single FIO instance that sequentially writes 32 GB of data on a single file. For the measurement of random write (`RW`) throughput, we run 16 FIO instances that randomly write 4 KB data on a file. Over each created file (`RW` or `SW`), we run FIO instances that read data sequentially (`+SR`) or randomly (`+RR`) to measure their respective read throughputs.



(a) Throughput



(b) File system traffic

Figure 12: Data intensive workloads

Overall, KEVIN shows similar performance as the other file systems as depicted in Figure 12(a). However, for random write (`RW`), it shows slightly low throughput because of extra I/Os for compaction (see Figure 10). An interesting observation is that KEVIN exhibits excellent performance even for random read workloads (`SW+RR` and `RW+RR`). This is because KEVIN benefits from the highly compressible key format of data objects. This property enables us to cache almost all KV indices in DRAM for the 32 GB file, making it possible to achieve a sufficiently high hit ratio.

6.2.2 Results with Realistic Workloads

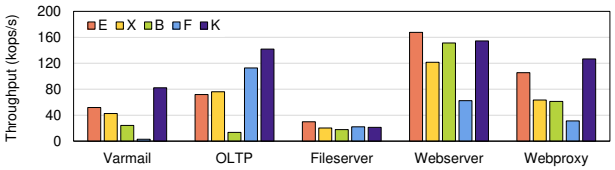
To understand the performance of KEVINFS under realistic workloads, we conduct experiments using five Filebench workloads (`Varmail`, `OLTP`, `Fileserver`, `Webserver`, and `Webproxy`), and four real applications (`TPC-C`, `clone`, `rsync`, and kernel compilation). Filebench mimics I/O behaviors of a target application that are modeled by parameters listed in Table 3. In our experiment, we use default parameters preset by Filebench, except for the number of operations.

Figures 13 and 14 show our results. For `Varmail`, KEVINFS exhibits 37% higher throughput than EXT4. `Varmail` emulates a mail server, so it performs I/Os on many small files. Metadata-intensive syscalls, `creat` and `unlink`, are frequently invoked to create and remove files. To persist user emails immediately, it calls `fsync` every time after `write`, incurring many I/Os to the journaling area.

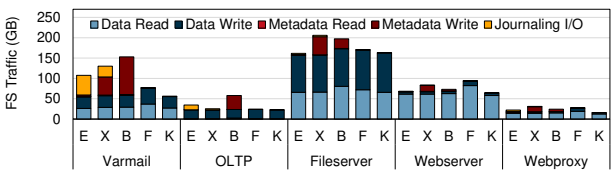
`OLTP` is a write-intensive workload in which more than 100 threads create files and append data to the files. It also frequently invokes `fdatasync`, which results in many I/Os sent to metadata and the journaling area. As a result, KEVIN exhibits 26% higher throughput than F2FS.

Table 3: Filebench parameters. C/U/R/W represents the ratio of `creat`, `unlink`, `read`, and `write` operations.

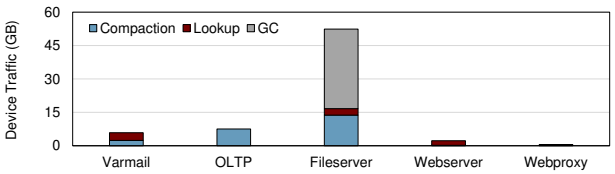
	Avg. file size	# of files	Threads	# of operations	C/U/R/W ratio
`Varmail`	16 KB	3.2 M	16	12.8 M	1:1:2:2
`OLTP`	10 MB	3.2 K	211	10 M	0:0:1:10
`Fileserver`	128 KB	800 K	50	8 M	1:1:1:2
`Webserver`	16 KB	1.6 M	100	12.8 M	0:0:10:1
`Webproxy`	16 KB	2 M	100	4 M	1:1:5:1



(a) Throughput



(b) File system traffic



(c) KEVIN I/O overheads

Figure 13: Realistic workloads from Filebench

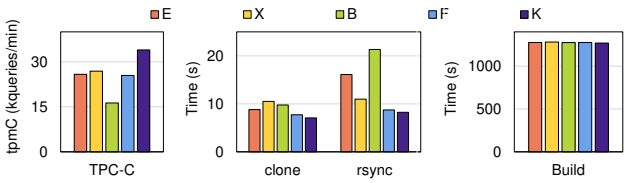


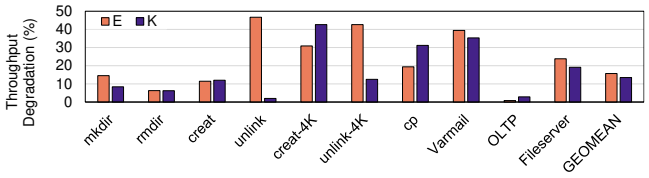
Figure 14: Application workloads

KEVINFS shows 23% lower throughput than EXT4 in `Fileserver`. `Fileserver` is a data-intensive workload that reads and writes a relatively large size of files (128 KB). It does not invoke `fsync`, so metadata updates and journaling I/Os occur only occasionally. Owing to the large amounts of data written to the disk, KEVINFS suffers from compaction overhead which slows down its performance over EXT4.

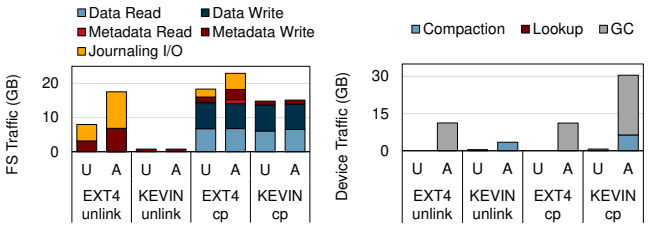
`Webserver` is a read-dominant workload issuing many reads to small files (16 KB) with few writes. Syscalls that update metadata are not invoked in `Webserver`. Although it is not preferable to KEVIN, KEVIN shows a slightly slower performance (10%) than EXT4. `Webserver` exhibits high locality in accessing files. Thanks to a high cache hit ratio (98.3%) in KEVINSSD, the number of page reads to get key-index pages is relatively small.

`Webproxy` is also a read-dominant workload, but KEVINFS exhibits higher throughput compared to EXT4. Our close examination reveals that this is owing to the high directory management overhead. `Webproxy` contains a large number of files (*e.g.,* 1M files) per directory. Whenever files are created and removed, it is necessary to update directory entries, which is costly. KEVINFS does not maintain directory entries, so its performance is not affected by directory updates.

Finally, we carry out experiments using real applica-



(a) Effect of fragmentation on performance



(b) File system traffic (c) I/O overhead of device

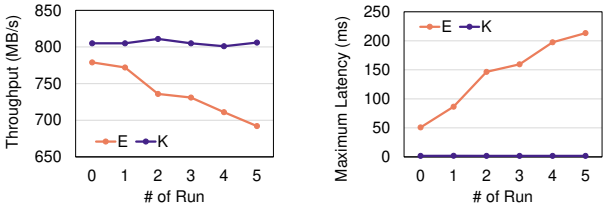
Figure 15: Write performance on aged file systems. In (b) and (c), 'U'/'A' shows unaged/aged file system performance.

tions, including `TPC-C`, `clone`, `rsync`, and kernel compilation (`build`). For `TPC-C`, MySQL is used as the DBMS engine. We create 50 data warehouses run by 100 clients. The DB size is 14 GB. The overall behavior of `TPC-C` is similar to `OLTP` in that it is write-intensive and frequently invokes `fsync`. Thus, KEVIN provides about 31% better throughput (tpmC) than EXT4. A local 3.1 GB Linux kernel repository is used as the source for both `clone` and `rsync`. They involve the creation of many small files (as `creat-4K` in §6.2.1), so KEVIN offers the best performance. A Linux kernel compilation process requires many small file reads and writes, along with directory traversals. Our results, however, reveal that the bottleneck of the kernel compilation is CPU not I/O. Consequently, all of the file systems provide similar compilation times.

6.2.3 Results under Aged File Systems

We analyze the performance of the file systems when they are aged. To age the file systems, we use Geriatrix [20] and Filebench to write more than 800 GB of data. For performance measurement, we run the same benchmarks that we use in §6.2.1. Since the file system space utilization is about 60%, we reduce the number of files and directories created by half.

Figure 15(a) shows the extent to which the file-system performance degrades after the aging process. We observe that KEVIN shows smaller performance reductions compared to EXT4 across almost all of the benchmarks. EXT4 is affected by high metadata and journaling overhead, which are exacerbated by file-system fragmentation. In the case of `unlink` in Figure 15(a), metadata and journaling I/Os increase by up to 2.2× after aging. On the other hand, there are no significant changes in file-system level I/O traffic in KEVIN. After aging, the compaction I/Os in KEVINSSD increase to 7.7×. Due in part to its very small portion in total I/Os, its negative impact on I/O performance is not huge. This confirms that KEVIN is more resistant to fragmentation. Unfortunately,



(a) FIO (b) `readdir`

Figure 16: Read performance on aged file systems

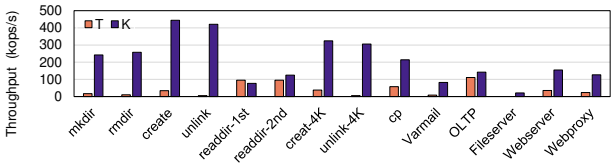
KEVIN suffers from increased compaction and GC overhead in data-intensive workloads, `creat-4K` and `cp`. Our LSM-tree indexing algorithm requires more flash space (3~10%) than the typical FTL, owing to obsolete objects staying in the tree before getting reclaimed by compaction. Thus, GC invocations occur more frequently.

To understand the impact of fragmentation on user-perceived performance, we measure read throughput and latency while varying the degree of fragmentation. The degree of fragmentation is controlled by the number of fragmentation tool runs. For each run, 128 GB data are written to the file system. We first measure sequential read throughput on a 32 GB large file (see Figure 16(a)). The read throughput of EXT4 gradually degrades as the run repeats. When the file system is clean (*i.e.,* the run 0), the file has only one extent. However, the number of extents increases to 3,798 at run 4. As explained in §4, this increases not only the tree search cost but the number of I/O requests to the disk. On the other hand, KEVIN exhibits consistent read throughput, achieving 16% higher throughput than EXT4.

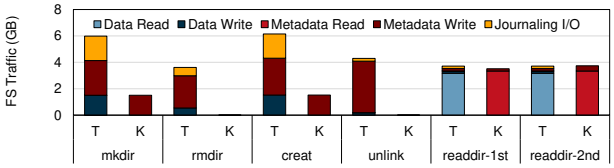
We also measure the latency of `readdir` (see Figure 16(b)). The performance of `readdir` is decided by the number of block reads to fetch inodes from the inode table. As the run repeats, the inode table is severely fragmented, and thus EXT4 involves more disk accesses to retrieve inodes. This results in an increase in latency of `readdir`. On average, KEVIN shows slower speed for `readdir` than EXT4, as in Figure 9(a). However, it is not affected by the fragmentation of the inode table and can remove data transfers to the host by fetching inodes internally. Moreover, by reading multiple meta objects at the same time through SSD's internal parallelism, it exhibits much shorter latency when the file system is aged.

6.2.4 Analyzing Effects of In-storage Indexing

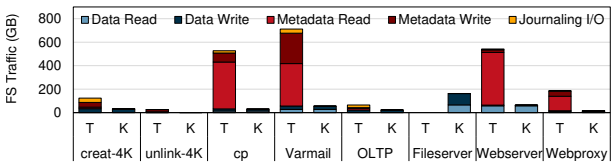
Finally, we evaluate the benefits of performing indexing operations in storage. The best way of doing this would be to move KEVINSSD's internal indexing engine to KEVINFS. Since our LSM-tree engine is currently designed to run as the flash firmware, porting this back to the kernel is a non-trivial job. As an alternative, we use TokuDB from previous KV store based file system studies [19, 56, 57]. TokuDB uses the B$^\epsilon$-tree as an internal indexing algorithm which has low computational complexity and asymptotically performs better than LSM-trees. We used the TokuDB version included in the B*e*trFS's git repository [33]. To bridge TokuDB with KEV-



(a) Throughput



(b) File system traffic for metadata workloads



(c) File system traffic for small-file and realistic workloads

Figure 17: Effects of in-storage indexing

INFS, we port the code from v3.11.10 [19] to v4.15.18 kernel. For a fair I/O traffic comparison, we turn off TokuDB's value compression feature and set the internal cache size to 4 GB. This in-kernel TokuDB operates on a block SSD formatted to EXT4 [19]. The block SSD uses a page-level FTL.

We conduct experiments with Filebench used from §6.2.1 and §6.2.2. Figure 17(a) shows I/O throughputs. In the graph, 'T' represents a setting where KEVINFS uses TokuDB as its in-kernel indexing engine in the host, while 'K' denotes the proposed KEVIN that uses the in-storage indexing engine. KEVIN shows an improvement of 7.4× on average even with `readdir` which is relatively slow.

To understand why KEVIN performs much better than KEVINFS+TokuDB, we analyze I/O traffics from the two settings, which are presented in Figures 17(b) and (c). KEVINFS+TokuDB numbers are taken from TokuDB's statistics. In KEVINFS+TokuDB, 'Data Read/Write' represents the traffic from reading and writing KV objects and 'Metadata Read/Write' is the extra indexing I/O traffic from the B^ε-tree to manage KV objects. 'Journaling I/O' includes the traffic from TokuDB's WAL logic. `Fileserver` fails to run on KEVINFS+TokuDB owing to the space overhead [19] caused by the B^ε-tree algorithm that consumes all disk space.

In the case of the write-intensive workloads, traffic differences are substantial. This is because TokuDB incurs many extra I/Os. As mentioned in §2.3, the WAL policy [13, 15, 34] has to write all KV objects to logs before materializing them to the data area. Note that this overhead can be mitigated if the late-binding journaling is used [56] which is not implemented yet in this work. In workloads such as `Varmail` that have many `fsync` calls, TokuDB has to flush the logs, increasing `fsync`'s latency. KEVIN shows 33.8× shorter latency compared to KEVINFS+TokuDB. The B^ε-tree also incurs more traffic because of its inherent behavior that transfers data from the internal node buffer to the leaf node. Operations involving many point queries such as `cp`, `Webserver`, and `Webproxy` show lower read traffic in KEVIN than KEVINFS+TokuDB. KEVIN performs indexing with the key-index page caching and compression from the storage device itself, and thus it offers fast indexing performance without any external I/Os. `readdir` shows worse performance on KEVIN. TokuDB manages all KV pairs in a sorted manner without key-value separation, and thus `ITERATE` performs quickly akin to sequential I/Os. However, when KEVIN rewrites the meta object before the second run, it shows higher performance, as it does not need to read multiple value pages. In this case, we expect the performance will be further improved if KEVIN adopts full-path indexing that globally sorts the file-system hierarchy.

7 Conclusion

In this paper, we proposed KEVIN, which improved file system performance by offloading indexing capability to the storage hardware. KEVINSSD exposed the KV interface and supported transaction commands. On top of this, we built KEVINFS, a new file system that translated VFS calls into KV objects and exploited storage capabilities to remove metadata and journaling overhead. Our results showed that, on average, KEVIN improved I/O throughput by 6.2× and reduced the I/O traffic by 74% for metadata-intensive workloads.

The idea of KEVIN can be extended in two directions. First, we focused on porting file systems over the KV device in this study. However, the proposed KV interface can be extended to support a broader range of applications, ranging from block-interface applications [8] to SQL applications [12], giving us the potential to replace the existing block I/O interface. Applications running directly over the KV device are expected to enjoy the same benefits (*e.g.,* small metadata overheads) as KEVIN. Second, KEVINFS can be implemented in the form of a user-level file system. KEVINSSD can export KV-APIs to the user-space (*e.g.,* via SPDK [54]), and KEVINFS accesses a storage device without going through the deep kernel stack. The user-level KEVINFS would be faster than existing user-level file systems because it not only has a lighter-weight architecture (*e.g.,* free from metadata management and journaling), but is also less affected by fragmentation.

Acknowledgments

We would like to thank our shepherd, Dr. Donald E. Porter, and five anonymous reviewers for all their helpful comments. This work was supported by Samsung Electronics Co., Ltd. and the National Research Foundation (NRF) of Korea (NRF-2018R1A5A1060031 and NRF-2019R1A2C1090337). We thank Samsung Electronics for providing KV-SSD prototypes. (Corresponding author: Sungjin Lee)

References

[1] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D Davis, Mark S Manasse, and Rina Panigrahy. Design tradeoffs for SSD performance. In *Proceedings of the USENIX Annual Technical Conference*, pages 57–70, 2008.

[2] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces, Crash Consistency: FSCK and Journaling*. Arpaci-Dusseau Books, 1.01 edition, 2019.

[3] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces, File System Implementation*. Arpaci-Dusseau Books, 1.01 edition, 2019.

[4] Jens Axboe. FIO: Flexible I/O Tester Synthetic Benchmark. https://github.com/axboe/fio.

[5] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. Lightnvm: The linux open-channel SSD subsystem. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 359–374, 2017.

[6] Yun-Sheng Chang and Ren-Shuo Liu. OPTR: Order-Preserving Translation and Recovery Design for SSDs with a Standard Block Device Interface. In *Proceedings of the USENIX Annual Technical Conference*, pages 1009–1024, 2019.

[7] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Consistency without Ordering. In *Proceedings of the USENIX Conference on File and Storage Technologies*, page 9, 2012.

[8] Chanwoo Chung, Jinhyung Koo, Junsu Im, Arvind, and Sungjin Lee. LightStore: Software-defined Network-attached Key-value Drives. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 939–953, 2019.

[9] Joel Coburn, Trevor Bunker, Meir Schwarz, Rajesh Gupta, and Steven Swanson. From ARIES to MARS: Transaction Support for next-Generation, Solid-State Drives. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 197–212, 2013.

[10] Alex Conway, Eric Knorr, Yizheng Jiao, Michael A. Bender, William Jannen, Rob Johnson, Donald E. Porter, and Martin Farach-Colton. Filesystem aging: It's more usage than fullness. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems*, 2019.

[11] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal Navigable Key-value Store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 79–94, 2017.

[12] Facebook, Inc. MyRocks: A RocksDB storage engine with MySQL. http://myrocks.io.

[13] Facebook, Inc. RocksDB: A Persistent Key-value Store for Fast Storage Environments. https://rocksdb.org.

[14] Akira Fujita and Takashi Sato. e4defrag - Online Defragmenter for Ext4 Filesystem. https://man7.org/linux/man-pages/man8/e4defrag.8.html.

[15] Google, Inc. LevelDB. https://github.com/google/leveldb.

[16] Sangwook Shane Hahn, Sungjin Lee, Cheng Ji, Li-Pin Chang, Inhyuk Yee, Liang Shi, Chun Jason Xue, and Jihong Kim. Improving File System Performance of Mobile Storage Systems Using a Decoupled Defragmenter. In *Proceedings of the USENIX Annual Technical Conference*, pages 759–771, 2017.

[17] Xiao He, Zhongxia Wang, Jingsheng Zhang, and Chengzi Ji. Research on security of hard disk firmware. In *Proceedings of International Conference on Computer Science and Network Technology*, pages 690–693, 2011.

[18] Junsu Im, Jinwook Bae, Chanwoo Chung, Arvind, and Sungjin Lee. PinK: High-speed In-storage Key-value Store with Bounded Tails. In *Proceedings of the USENIX Annual Technical Conference*, pages 173–187, 2020.

[19] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. BεtrFS: A Right-Optimized Write-Optimized File System. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 301–315, 2015.

[20] Saurabh Kadekodi, Vaishnavh Nagarajan, and Gregory R. Ganger. Geriatrix: Aging What You See and What You Don't See. A File System Aging Approach for Modern Storage Systems. In *Proceedings of the USENIX Annual Technical Conference*, pages 691–704, 2018.

[21] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Gi-Hwan Oh, and Changwoo Min. X-FTL: Transactional FTL for SQLite Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 97–108, 2013.

[22] Yangwook Kang, Rekha Pitchumani, Pratik Mishra, Yang-suk Kee, Francisco Londono, Sangyoon Oh, Jongyeol Lee, and Daniel D. G. Lee. Towards Building a High-performance, Scale-in Key-value Storage System. In *Proceedings of the ACM International Conference on Systems and Storage*, pages 144–154, 2019.

[23] Sudarsun Kannan, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Yuangang Wang, Jun Xu, and Gopinath Palani. Designing a True Direct-Access File System with DevFS. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 241–256, 2018.

[24] Sang-Hoon Kim, Jinhong Kim, Kisik Jeong, and Jin-Soo Kim. Transaction Support using Compound Commands in Key-Value SSDs. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems*, July 2019.

[25] Changman Lee, Dongho Sim, Joo Young Hwang, and Sangyeun Cho. F2FS: A new file system for flash storage. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 273–286, 2015.

[26] Seung-Ho Lim, Hyun Jin Choi, and Kyu Ho Park. Journal Remap-based FTL for Journaling File System with Flash Memory. In *Proceedings of the International Conference on High Performance Computing and Communications*, pages 192–203, 2007.

[27] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A study of Linux file system evolution. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 31–44, 2013.

[28] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 133–148, 2016.

[29] Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Physical disentanglement in a container-based file system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 81–96, 2014.

[30] Changwoo Min, Woon-Hak Kang, Taesoo Kim, Sang-Won Lee, and Young Ik Eom. Lightweight application-level crash consistency on transactional flash storage. In *Proceedings of the USENIX Annual Technical Conference*, pages 221–234, 2015.

[31] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The Log-structured Merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[32] Daejun Park and Dongkun Shin. iJournaling: Fine-Grained Journaling for Improving the Latency of Fsync System Call. In *Proceedings of the USENIX Annual Technical Conference*, pages 787–798, 2017.

[33] Percona, Inc. BetrFS Repository. https://github.com/oscarlab/betrfs.

[34] Percona, Inc. Percona TokuDB. https://www.percona.com/software/mysql-database/percona-tokudb.

[35] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and evolution of journaling file systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, pages 8–8. USENIX Association, 2005.

[36] Vijayan Prabhakaran, Thomas L. Rodeheffer, and Lidong Zhou. Transactional Flash. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, pages 147–160, USA, 2008.

[37] Anthony Rebello, Yuvraj Patel, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Can Applications Recover from fsync Failures? In *Proceedings of the USENIX Annual Technical Conference*, pages 753–767, 2020.

[38] Reiser, H. ReiserFS. http://www.namesys.com, 2004.

[39] Kai Ren and Garth Gibson. TABLEFS: Enhancing Metadata Efficiency in the Local File System. In *Proceedings of the USENIX Annual Technical Conference*, pages 145–156, 2013.

[40] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-Tree Filesystem. *Trans. Storage*, 9:9:1–9:32, 2013.

[41] Samsung Electronics. Samsung Key Value SSD enables High Performance Scaling. https://www.samsung.com/semiconductor/global.semi.static/Samsung_Key_Value_SSD_enables_High_Performance_Scaling-0.pdf, 2017.

[42] Samsung Electronics. 860EVO SSD Specification. https://www.samsung.com/semiconductor/global.semi.static/Samsung_SSD_860_EVO_Data_Sheet_Rev1.pdf, 2018.

[43] Samsung Electronics. 960PRO SSD Specification. https://www.samsung.com/semiconductor/minisite/ssd/product/consumer/ssd960/, 2019.

[44] Russell Sears and Raghu Ramakrishnan. bLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 217–228, 2012.

[45] Muthian Sivathanu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Somesh Jha. A Logic of File Systems. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 1–1, 2005.

[46] SNIA. Key Value Storage API Specification Version 1.0. https://www.snia.org/tech_activities/standards/curr_standards/kvsapi.

[47] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference*, pages 1–1, 1996.

[48] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A Flexible Framework for File System Benchmarking. *The USENIX Magazine*, 41, 2016.

[49] The Linux Foundation. Ext4 Filesystem documentation. https://www.kernel.org/doc/Documentation/filesystems/ext4.txt.

[50] Thomas N Theis and H-S Philip Wong. The end of moore's law: A new beginning for information technology. *Computing in Science & Engineering*, 19(2):41–50, 2017.

[51] Rajat Verma, Anton Ajay Mendez, Stan Park, Sandya Srivilliputtur Mannarswamy, Terence P. Kelly, and Charles B. Morrey III. Failure-atomic updates of application data in a linux file system. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 203–211, 2015.

[52] Youjip Won, Jaemin Jung, Gyeongyeol Choi, Joontaek Oh, Seongbae Son, Jooyoung Hwang, and Sangyeun Cho. Barrier-Enabled IO Stack for Flash Storage. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 211–226, 2018.

[53] Xilinxm, Inc. Xilinx Virtex UltraScale FPGA VCU108 Evaluation Kit. https://www.xilinx.com/products/boards-and-kits/ek-u1-vcu108-g.html#hardware.

[54] 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.

[55] Jeseong Yeon, Minseong Jeong, Sungjin Lee, and Eunji Lee. RFLUSH: Rethink the Flush. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 201–210, 2018.

[56] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Optimizing every operation in a write-optimized file system. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 1–14, 2016.

[57] Yang Zhan, Alex Conway, Yizheng Jiao, Eric Knorr, Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Donald E. Porter, and Jun Yuan. The full path to full-path indexing. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 123–138, 2018.

[58] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, et al. FPGA-Accelerated Compactions for LSM-based Key-Value Store. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 225–237, 2020.

A Artifact Appendix

Abstract

KEVIN is composed of two main elements: KEVINSSD (providing key-value interface with in-storage indexing) and KEVINFS (providing file system abstraction). The artifact is consisted of multiple Git repositories including KEVINSSD, KEVINFS and others used for evaluation for KEVIN. Please refer to the README file from https://github.com/dgist-datalab/kevin.

Scope

The artifact includes all the necessary source code required to run KEVIN as well as the benchmarks used in this paper. As it takes several weeks to run all the benchmarks used in this paper, we also provide exemplary benchmark suite with tuned parameters.

Contents

We provide four Git repositories related to KEVIN. First, KEVINFS provides abstraction of files and directories (see §3.1, §4 and §5.1). Second, KEVINSSD is the storage engine optimized for in-storage indexing using LSM-tree (see §3.2, §3.3 and §5.2). Third, BLOCKSSD is another storage engine used for comparison. Its FTL firmware uses page-level mapping and provides the block interface to the host. BLOCKSSD is used for comparison of traditional file systems in §6. Lastly, we provide the kernel source used in this paper. It is based on Linux kernel v4.15.18 and is further customized to run KEVIN+TokuDB (see §6.2.4). KEVINFS also runs on this kernel. Additionally, the DOI for the artifact includes a detailed screencast of the tool along with results with example workloads to prove the functionality of KEVIN.

Hosting

We provide the public Git URLs and commit hashes for each repository used during the artifact evaluation.

- KEVINFS

  https://github.com/dgist-datalab/kevin

  `1bd8566c580f8190364008f1a355fe337fcb6309`

- KEVINSSD

  https://github.com/dgist-datalab/KevinSSD

  `026e2a9bd274989b1324bdb9d008f2044e6d145d`

- BLOCKSSD

  https://github.com/dgist-datalab/BlockSSD

  `f944a94455f56a42ee3a888431b71d7e555b7671`

- Kernel source used with KEVIN

  https://github.com/dgist-datalab/linux/tree/kevin-4.15

  Branch: `kevin-4.15`

  `c2b106a1de494c293f33c5f130435c2eaee02dcf`

- The DOI for the artifact

  `10.5281/zenodo.4659803`

  https://zenodo.org/record/4659803

Requirements

We use the Xilinx Virtex® UltraScale™ FPGA VCU108 platform and customized NAND flash modules. The customized NAND flash modules used in this paper are not publicly or commercially available. Therefore, you may need your own NAND modules compatible with VCU108 and adequate modifications to the hardware backend (KEVINSSD and BLOCKSSD) to replicate this work.