F2FS: A New File System for Flash Storage

Paper # 31

Abstract

F2FS is a Linux file system designed from scratch to perform well on modern flash storage devices. The file system builds on append-only logging and its key design decisions were made with the characteristics of flash storage in mind. This paper describes in detail the main design ideas, data structures, algorithms and the resulting performance of F2FS.

Experimental results highlight the desirable performance of F2FS; on a state-of-the-art mobile system, it outperforms EXT4 under synthetic workloads by up to 3.1× (iozone) and 2× (SQLite). Under realistic workloads, it reduces the elapsed time by up to 40% over EXT4. On a server system, F2FS is shown to perform better than EXT4 by up to 2.5× using a SATA SSD and 8× with a PCIe SSD.

1 Introduction

NAND flash memory has been used widely in various mobile devices like smartphones, tablets and MP3 players. Furthermore, server systems started utilizing flash devices as their primary storage. Despite its broad use, flash memory has several limitations, like erase-before-write requirement, the need to write on erased blocks sequentially and limited write cycles per erase block.

In early days, many consumer electronic devices directly utilized "bare" NAND flash memory put on a platform. With the growth of storage needs, however, it is increasingly common to use a "solution" that has multiple flash chips connected through a dedicated controller. The firmware running on the controller, commonly called FTL (flash translation layer), addresses the NAND flash memory's limitations and provides a generic block device abstraction. Examples of such a flash storage solution include eMMC (embedded multimedia card), UFS (universal flash storage) and SSD (solid-state drive). Typically, these modern flash storage devices show much lower access latency than a hard disk drive (HDD), their mechanical counterpart. When it comes to random I/O, SSDs perform orders of magnitude better than HDDs.

However, under certain usage conditions of flash storage devices, the idiosyncrasy of the NAND flash media manifests. For example, Min et al. [20] observe that frequent random writes to an SSD would incur internal fragmentation of the underlying media and degrade the sustained SSD performance. Studies indicate that random write patterns are quite common and even more taxing to resource-constrained flash solutions on mobile devices. Kim et al. [11] quantified that the facebook mobile application issues 150% and WebBench register 70% more random writes than sequential writes. Furthermore, over 80% of total I/Os are random and more than 70% of there-in writes are triggered with `fsync` by applications such as facebook and twitter [8]. This specific I/O pattern comes from the dominant use of SQLite [2] in those applications. Unless handled carefully, frequent random writes and flush operations in modern workloads can seriously increase a flash device's I/O latency and reduce the device lifetime.

The detrimental effects of random writes could be reduced by the log-structured file system (LFS) approach [26] and/or the copy-on-write strategy. For example, one might anticipate file systems like BTRFS [25] and NILFS2 [14] would perform well on NAND flash SSDs; unfortunately, they do not consider the characteristics of flash storage devices and are inevitably sub-optimal in terms of performance and device lifetime. We argue that traditional file system design strategies for HDDs—albeit beneficial—fall short of fully leveraging and optimizing the usage of the NAND flash media.

In this paper, we propose F2FS, a new file system optimized for modern flash storage devices. As far as we know, F2FS is the first publicly and widely available file system that is designed from scratch to optimize performance and lifetime of flash devices with a generic block interface.[1] This paper describes the design and implementation of F2FS.

Listed in the following are the main considerations for the design of F2FS:

• **Flash-friendly on-disk layout (Section 2.1).** F2FS employs three configurable units: *segment*, *section* and *zone*. It allocates storage blocks in the unit of segments from a number of individual zones. It performs "cleaning" in the unit of section. These units are introduced to align with the underlying FTL's operational units to avoid unnecessary (yet costly) data copying and increase device lifetime accordingly.

• **Cost-effective index structure (Section 2.2).** LFS writes data and index blocks to newly allocated free space. If a leaf data block is updated (and written to somewhere), its direct index block should be updated, too. Once the direct index block is written, again its indirect index block should be updated. Such recursive updates result in a chain of writes, creating the "wandering tree" problem [4]. In order to attack this problem, we propose a novel index table called *node address table*.

• **Multi-head logging (Section 2.4).** We devise an effective hot/cold data separation scheme applied during logging time (i.e., block allocation time). It runs multiple active "log segments" concurrently and appends data and metadata to separate log segments based on their anticipated update frequency. Since the flash storage devices exploit media parallelism, multiple active segments can run simultaneously without frequent management operations, which makes potential performance degradation due to multiple logging (compared to single-segment logging) insignificant, unlike HDDs.

• **Adaptive logging (Section 2.6).** F2FS builds basically on the append-only logging concept to turn random writes into sequential ones on the fly. At high storage utilization, however, it changes the logging strategy to threaded logging [23] to avoid long write latency. In essence, threaded logging writes new data to free space in a dirty segment without cleaning it in the foreground. This strategy works well on modern flash devices but may not do so on HDDs.

• `fsync` **acceleration with roll-forward recovery (Section 2.7).** F2FS optimizes small synchronous writes to reduce the latency of `fsync` requests, by minimizing required metadata writes and recovering synchronized data with an efficient roll-forward mechanism.

[1]F2FS has been available in the Linux kernel since version 3.8 and has been adopted in commercial products.

In a nutshell, F2FS builds on the concept of LFS but deviates significantly from the original LFS proposal with many new design considerations. We have implemented F2FS as a Linux file system and compare it with two state-of-the-art Linux file systems—EXT4 and BTRFS. We also evaluate NILFS2, an alternative implementation of LFS in Linux. Our evaluation considers two generally categorized target systems: mobile system and server system. In the case of the server system, we study the file systems on a SATA SSD and a PCIe SSD. The results we obtain and present in this work highlight the overall desirable performance characteristics of F2FS.

The remainder of this paper is organized as follows. Section 2 first describes in detail the design and implementation of F2FS. Section 3 provides performance results and discussions. We describe related work in Section 4 and conclude in Section 5.

2 Design and Implementation of F2FS

2.1 On-Disk Layout

The on-disk data structures of F2FS are carefully laid out to match how underlying NAND flash memory is organized and managed. As illustrated in Figure 1, F2FS divides the whole volume into *segments*, each of which has a fixed size. The segment is a basic unit of management hard-coded in F2FS, which is used to determine the initial file system metadata layout. Given that the erase block size in many flash storage devices is 2MB, we match the segment size to be 2MB by default.

A *section* is comprised of consecutive segments, and a *zone* consists of a series of sections. These units are important during *logging* and *cleaning*, which will be further discussed in Section 2.4 and 2.5. By default, both section and zone are sized to match a segment, but users can easily modify the sizes with `mkfs.f2fs`—the F2FS format tool.

F2FS splits the entire volume into six areas:

• **Superblock (SB)** has the basic partition information and default parameters of F2FS, which are given at the format time and not changeable.

• **Checkpoint (CP)** keeps the file system status, bitmaps for valid NAT/SIT sets (see below), orphan inode lists and summary entries of currently active segments. A successful "checkpoint pack" should store a consistent F2FS status at a given point of time, which is used as a recovery point after a sudden power-off event, as described in Section 2.7. The CP area stores two checkpoint packs across the two segments (#0 and #1): one for the last stable version and the other for the intermediate

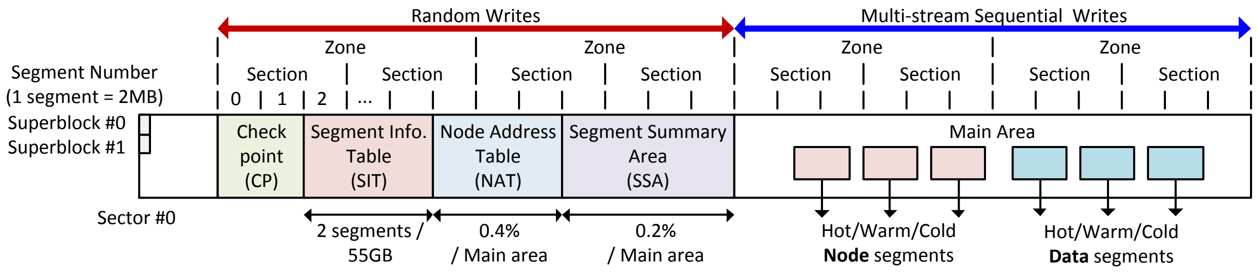


Figure 1: On-disk layout of F2FS.

or obsolete version, alternatively.

• **Segment Information Table (SIT)** contains per-segment information such as the number of valid blocks and the bitmap for the validity of all the blocks in the "Main" area (see below). The SIT information is retrieved to select victim segments and identify there-in valid blocks during the cleaning process, as will be explained in Section 2.5. The SIT occupies two segments per 55GB of the Main area capacity.

• **Node Address Table (NAT)** is a block address table to locate all the "node blocks" stored in the Main area. It occupies 0.4% of the Main area size.

• **Segment Summary Area (SSA)** stores summary entries representing the owner information of all the blocks in the Main area, such as parent inode number and its node/data offsets. The SSA entries are used to load parent node blocks for migrating valid blocks during the cleaning operation. It occupies 0.2% of the Main area size.

• **Main Area** is dynamically filled with 4KB-sized blocks. Each block is allocated and used as one of two types, *node* and *data*. A node block contains inode or indices of data blocks, while a data block contains either directory or user file data. Note that a section does not store data and node blocks simultaneously.

Given the above on-disk data structures, let us illustrate how a file look-up operation is done in F2FS. Assuming a file "`/dir/file`", F2FS performs the following steps: (1) It obtains the root inode by reading a block whose location is obtained from NAT; (2) In the root inode block, it searches for a directory entry named `dir` from its data blocks and obtains its inode number; (3) It translates the retrieved inode number to a physical location through NAT; (4) It obtains the inode named `dir` by reading the corresponding block; and (5) In the `dir` inode, it identifies the directory entry named `file`, and finally, obtains the file inode by repeating steps (3) and (4) for `file`. The actual `file` data can be retrieved from the Main area, with indices obtained via the corresponding file structure.

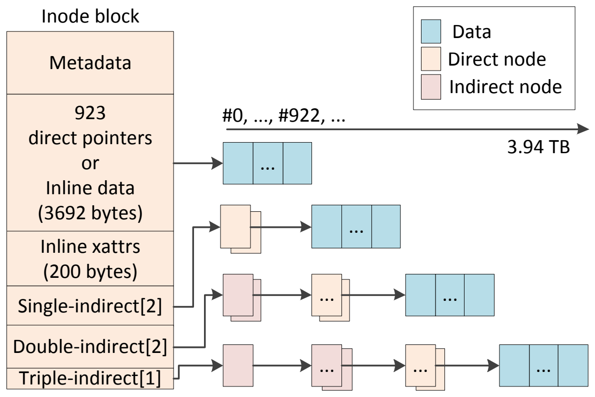


Figure 2: File structure of F2FS.

2.2 File Structure

The original LFS introduced *inode map* to translate an inode number to an on-disk location. In a different manner, F2FS proposes *node structure* that extends the inode map to locate more indexing blocks. Each node block has a unique identification number, "node ID". By using the node ID as an index, NAT serves the physical locations of all the node blocks. A node block represents one of three types: inode, direct, and indirect node. An inode block contains the file's metadata, such as file name, inode number, file size, atime, and dtime. A direct node block contains block addresses of data and an indirect node block has node IDs locating another node blocks.

As illustrated in Figure 2, F2FS uses pointer-based file indexing with direct and indirect node blocks to mitigate excessive update propagation (i.e., the wandering tree problem [26]). In the traditional LFS design, if a leaf data is updated, its direct and indirect pointer blocks are updated recursively. F2FS, however, only has to update the one direct node block and its NAT entry, effectively addressing the wandering tree problem. For example, when a 4KB-sized data is appended to a file of 8MB to 4GB, the LFS needs to update two pointer blocks recursively while F2FS updates only one direct node block (not considering cache effects). For files larger than

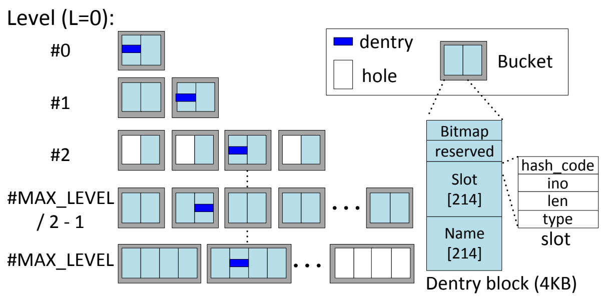


Figure 3: Directory structure of F2FS.

4GB, the LFS needs to update one more pointer block (three blocks total) while F2FS still updates only one.

An inode block contains four-byte direct pointers to the file's first 923 data blocks, two single-indirect pointers, two double-indirect pointers and one triple-indirect pointer. A single-indirect pointer points to a direct node that has 1,018 direct pointers. A double-indirect pointer points to an indirect node that has 1,018 single-indirect pointers, and a triple-indirect pointer points to a double indirect node containing 1,018 double-indirect pointers. An inode can address up to 1,057,053,439 data blocks, corresponding to about 3.94TB.

F2FS supports *inline data* and *inline extended attributes*, which embed small-sized data or extended attributes into the inode block itself. These inline features reduce space overheads and improve I/O performance. Note that many embedded systems have small-sized files and a small number of extended attributes. By default, F2FS activates inlining of data if a file size is smaller than 3,692 bytes. F2FS reserves 200 bytes in an inode block for storing extended attributes.

2.3 Directory Structure

Figure 3 shows that a 4KB directory entry ("dentry") block is composed of a bitmap and two arrays of slots and names in pairs. The bitmap tells whether each slot is valid or not. A slot occupies 11 bytes and carries a hash value (four bytes), inode number (four bytes), length of a file name (two bytes), and file type (one byte) (e.g., normal file, directory, and symbolic link). The name entry is an eight-byte character string. A single dentry block contains 214 pairs of slots and names, and a bitmap for 214 bits. Accordingly, a dentry consumes one to 32 consecutive slots, which supports file names of up to 255 bytes.

In F2FS, a directory file constructs *multi-level hash tables* to manage a large number of dentries efficiently. In each N^{th} level, a hash table is configured by the number of buckets, B, and there-in dentry blocks, D, determined by:

$$\mathbf{B} = \begin{cases} 2^{(n+L)} & N < MAX\_LEVEL/2 \\ 2^{(MAX/2+L-1)} & N \geq MAX\_LEVEL/2 \end{cases}$$

$$\mathbf{D} = \begin{cases} 2 & N < MAX\_LEVEL/2 \\ 4 & N \geq MAX\_LEVEL/2 \end{cases}$$

where *MAX_LEVEL* and L are set to 63 and 0, respectively, by default.

When F2FS looks up a given file name in a directory, it first calculates the hash value of the file name. Then, it traverses the constructed hash tables incrementally from level 0 to the maximum allocated level recorded in the inode. In each level, it scans one bucket of two or four dentry blocks, resulting in *O(log(# of dentries))* complexity. To find a dentry more quickly, it compares the bitmap, the hash value, and the file name in order.

When creating a file, F2FS traverses the existing hash tables likewise the above lookup procedure, until it finds contiguous free slots covering the given file name. If it fails, F2FS expands the directory level and allocates an empty dentry block to secure more space. When deleting a file, F2FS finds the target dentry, and then deactivates a series of bits representing the dentry in the dentry block's bitmap. If the entire bitmap is clear, F2FS deallocates the dentry block, leaving a punched hole.

When large directories are preferred (e.g., in a server environment), users can configure F2FS through the `sysfs` mechanism [21], to initially allocate space for many dentries. If the initial directory level is 4, for example, F2FS builds a level 0 hash table with 2^4 buckets as the first step, which covers up to 6,848 dentries. With a larger hash table at the lowest level, F2FS can reach to a target dentry more quickly.

2.4 Multi-head Logging

Unlike the LFS that has one large log area, F2FS maintains six major log areas to maximize the effect of hot and cold data separation. F2FS statically defines three levels of temperature—hot, warm and cold—for node and data blocks, as summarized in Table 1.

Direct node blocks are considered hot, since they are updated much more frequently than indirect node blocks. Indirect node blocks contain node IDs and are written only when a dedicated node block is added or removed. Direct node blocks and data blocks for directories are considered hot, since they have obviously different write patterns compared to blocks for regular files. Data blocks satisfying one of the following three conditions are considered cold:

Table 1: Separation of objects in multiple active segments.

Type	Temp.	Objects
Node	Hot	Direct node blocks for directories
	Warm	Direct node blocks for regular files
	Cold	Indirect node blocks
Data	Hot	Dentry blocks
	Warm	Data blocks made by users
	Cold	Data blocks moved by cleaning Cold data blocks specified by users Multimedia file data

• **Data blocks moved by cleaning** (see Section 2.5). Since they have remained valid for an extended period of time, we expect they will not be invalidated in the near future.
• **Data blocks specifically labeled "cold" by users**. F2FS supports an extended attribute operation to this end.
• **Multimedia file data**. They likely show write-once and read-only patterns. F2FS identifies them by matching a file's extension against the file extensions registered at the format time.

By default, F2FS activates six logs open for writing. The user may adjust the number of write streams to two or four at the mount time if doing so is believed to yield better results on a given storage device and platform. If six logs are used, each logging segment corresponds directly to a temperature level listed in Table 1. In the case of four logs, F2FS combines the cold and warm logs in each of node and data types. Finally, with only two logs, F2FS uses one for node and the other for data types. Section 3.2.3 examines how the number of logging heads affects the effectiveness of data separation.

F2FS introduces configurable zones to be compatible with different FTLs, with a view to mitigating the garbage collection (GC) overheads.[2] FTL algorithms can be largely classified into three groups based on the associativity between data and "log flash blocks" [24]. Once a data flash block is assigned to store initial data, log flash blocks assimilate data updates as much as possible, like the journal in EXT4 [17]. The log flash block can be used exclusively for a single data flash block ("block-associative") [12], for all data flash blocks ("fully-associative") [16], and for a set of contiguous data flash blocks ("set-associative") [24]. Modern FTLs tend to adopt a fully-associative or set-associative method, to be able to properly handle random writes. Note that F2FS writes node and data blocks in parallel using multi-head logging and an associative FTL would mix the separated blocks (in the file system level) into the same flash block. In order to avoid such misalignment, F2FS maps active logs to different zones to separate them in the FTL. This mapping strategy is expected to be effective for set-associative FTLs.

[2]Conducted by FTL, GC involves copying valid flash pages and erasing flash blocks for further data writes. GC overheads depend partly on how well file system operations align to the given FTL mapping algorithm.

2.5 Cleaning

Cleaning is a process to reclaim scattered and invalidated blocks, and secure free segments for further logging. Because cleaning occurs constantly once the underlying storage capacity has been filled up, limiting the costs related with cleaning is extremely important for the performance of F2FS (and any LFS in general). In F2FS, cleaning is done in the unit of a section—analogous to the FTL's flash block during GC.

F2FS performs cleaning in two distinct manners, *foreground* and *background*. Foreground cleaning is triggered only when there are not enough free sections, while a kernel thread wakes up periodically to conduct cleaning in background. A cleaning process takes the following three steps.

(1) Victim selection. The cleaning process starts first to identify the best victim section among non-empty sections. There are two well-known policies for victim selection during LFS cleaning—"greedy" and "cost-benefit" [10, 26]. The greedy policy selects a section with the smallest number of valid blocks that SIT keeps track of. Intuitively, this policy mitigates overheads incurred while migrating valid blocks. F2FS adopts the greedy policy for its foreground cleaning to minimize the latency directly visible to applications.

On the other hand, the cost-benefit policy is practiced in the background cleaning process of F2FS. This policy selects a victim section not only based on the utilization but also its "age". F2FS infers the age of a section from the average ages of the segments in the section, which, in turn, can be obtained from their last modification time recorded in SIT. As the cost-benefit policy is used by the background cleaning, F2FS gives another chance to separate hot and cold data adaptively according to the user behavior at run time.

(2) Valid block identification and migration. After selecting a victim section, F2FS must identify valid blocks in the section quickly. To this end, F2FS maintains a validity bitmap per segment in SIT. Once having identified all valid blocks by scanning the bitmaps, F2FS retrieves the parent node blocks containing their indices from the SSA information. If it turns out that the blocks are valid, F2FS migrates them to other free logs.

For background cleaning, F2FS does not issue actual I/Os to migrate valid blocks. Instead, F2FS loads the blocks into page cache and marks them as dirty. Then, F2FS just leaves them in the page cache for the kernel worker thread to flush them to the storage later. This *lazy migration* not only alleviates the performance impact on foreground I/O activities, but also allows small writes to be combined into larger ones.

(3) Post-cleaning process. After all the valid blocks are migrated, the victim section is registered as a candidate to become a new free section (called "pre-free" section in F2FS). After a checkpoint is made, the section finally becomes a free section that can be reallocated. We do this because if a pre-free section is reused before checkpointing, the file system may lose the data referenced by the previous checkpoint when an unexpected power outage event occurs.

2.6 Adaptive Logging

The original LFS introduced two logging policies, *normal logging* and *threaded logging*. In the normal logging, blocks are written to clean segments, yielding strictly sequential writes. Even if users submit many random write requests, this process transforms them to sequential writes as long as there exists enough free logging space. As the free space shrinks to nil, however, this policy starts to suffer from high cleaning overheads, resulting in a serious performance drop (quantified to be over 90% under harsh conditions, see Section 3.2.4). On the other hand, the threaded logging writes blocks to *holes* (invalidated, obsolete space) in existing dirty segments. This policy requires no cleaning operations, but triggers random writes and potentially degrades performance as a result.

F2FS implements both the policies and switches between them dynamically according to the file system status. More concretely, if there are more than *k* clean sections, where *k* is a predefined threshold, the normal logging is initiated. Otherwise, the threaded logging is activated to avoid cleaning. *k* is set to 5% of total sections by default and can be configured by the user. Note that if the underlying SSD's FTL reserves a number of flash blocks for better random write performance (commonly called "overprovisioned capacity"), it would be better to have a high *k* in order to activate the threaded logging earlier.

There is a chance that the threaded logging incurs undesirable random writes when there are scattered holes. Nevertheless, such random writes typically show better spatial locality than those in update-in-place file systems like EXT4, since all the holes in a dirty segment are filled first before F2FS searches for more in other dirty segments. Note that Lee et al. [15] demonstrated that flash storage devices show better performance for random writes with strong spatial locality. F2FS gracefully gives up the normal logging and turns to the threaded logging for better sustainable performance, as will be demonstrated in Section 3.2.4.

2.7 Checkpointing and Recovery

F2FS implements *checkpointing* to provide a consistent recovery point from a sudden power failure or system crash. Whenever it needs to remain a consistent state across events like `sync`, `umount`, and foreground cleaning, F2FS triggers a checkpoint procedure as follows: (1) All the dirty node and dentry blocks in the page cache are flushed; (2) It suspends ordinary writing activities including system calls such as `create`, `unlink`, and `mkdir`; (3) The file system metadata, NAT, SIT, and SSA, are written to their dedicated areas on the disk; and (4) Finally, F2FS writes a *checkpoint pack*, consisting of the following information, to the CP area:

• **Header and footer** is written at the beginning and the end of the pack, respectively. A nominated there-in information is the version number that is incremented whenever a checkpoint is made. The version number is used to discriminate the latest stable pack between two packs during the mount time;

• **NAT and SIT bitmaps** indicate the set of NAT and SIT blocks comprising the current pack;

• **NAT and SIT journal** contain a small number of recently modified entries of NAT and SIT to avoid frequent NAT and SIT updates;

• **Summary blocks of active segments** consist of under-constructing SSA blocks that will be flushed to the SSA area in the future; and

• **Orphan blocks** keep orphan inode information. If an inode is deleted before it is closed yet, it should be registered as an orphan inode, so that F2FS can recover it after a sudden power-off.

2.7.1 Roll-Back Recovery

After a sudden power-off, F2FS rolls back to the latest consistent checkpoint. In order to keep at least one stable checkpoint pack while creating a new pack, F2FS maintains two checkpoint packs. If a checkpoint pack has identical contents in the header and footer, F2FS considers it valid. Otherwise, it should be the intermediate pack that we have to drop.

Likewise, F2FS also manages two sets of NAT and SIT blocks, distinguished by the NAT and SIT bitmaps in each checkpoint pack. Whenever it needs to write updated NAT or SIT blocks during checkpointing, F2FS writes them to one of the two sets alternatively, and then mark the bitmap to point to its new set.

If a small number of NAT or SIT entries are updated frequently, F2FS would write many 4KB-sized NAT or SIT blocks. To mitigate this overhead, F2FS proposes a *NAT and SIT journal* within the checkpoint pack. In our current design, the journal can store up to 38 NAT entries and 6 SIT entries. This technique reduces the number of I/Os, and accordingly, the checkpointing latency as well.

During the recovery procedure at mount time, F2FS searches valid checkpoint packs by inspecting headers and footers. If both checkpoint packs are valid, F2FS picks the latest one by comparing their version numbers. Once choosing the latest valid checkpoint pack, it checks whether orphan inode blocks exist or not. If so, it truncates all the data blocks referenced by them and lastly frees the orphan inodes, too. Then, F2FS starts file system services with a consistent set of NAT and SIT blocks referenced by their bitmaps, after the roll-forward recovery procedure is done successfully, as is explained below.

2.7.2 Roll-Forward Recovery

Applications like database (e.g., SQLite) frequently write small data to a database file and conduct `fsync` to guarantee reliability and durability. A naïve approach to supporting `fsync` would be to trigger checkpointing and recover data with the roll-back model. This approach, however, suffers from poor performance, since checkpointing involves writing all node and dentry blocks that are unrelated to the database file.

F2FS implements an efficient roll-forward recovery mechanism to enhance the `fsync` performance. The key idea is to write data blocks and their direct node blocks only, excluding the other node or F2FS metadata blocks. In order to find the data blocks selectively after rolling back to the stable checkpoint, F2FS remains a special flag inside the direct node blocks.

F2FS performs the roll-forward recovery as follows. If we denote the log position of the last stable checkpoint as **N**, (1) F2FS collects the direct node blocks having the special flag located in **N+n**, while constructing a list of their node information. (2) By using the node information in the list, it loads the most recently written node blocks, named **N-n**, into the page cache. (3) Then, it compares the data indices in between **N-n** and **N+n**. (4) If it detects different data indices, then it refreshes the cached node blocks with the new indices stored in **N+n**, and finally marks them as dirty. Once completing the roll-forward recovery, F2FS performs checkpointing to store the whole in-memory changes to the disk.

Table 2: Platforms used in experimentation. (Numbers in parenthesis are basic sequential and random performance *(Seq-R, Seq-W, Rand-R, Rand-W)* in MB/s.)

Target	**System**	**Storage Devices**
Mobile	CPU: Exynos 5410 Memory: 2GB OS: Linux 3.4.5 Android: JB 4.2.2	eMMC 16GB: 2GB partition: *(114, 72, 12, 12)*
Server	CPU: Intel i7-3770 Memory: 4GB OS: Linux 3.14 Ubuntu 12.10 server	SATA SSD 250GB: *(486, 471, 40, 140)* PCIe SSD 250GB: *(666, 616, 37, 140)*

3 Evaluation

3.1 Experimental Setup

We evaluate F2FS on two broadly categorized target systems, mobile system and server system. We employ Galaxy S4, a commercial platform, to represent the mobile system and an x86 platform for the server system. Specifications of the platforms are summarized in Table 2.

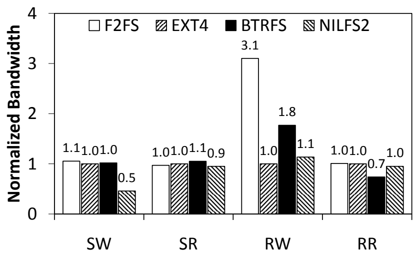
For mobile and server systems, we back-ported F2FS from the 3.15-rc1 main-line kernel to 3.4.5 and 3.14 kernels respectively. In the mobile platform, F2FS runs on a state-of-the-art eMMC flash storage. In the case of the server system, we harness a SATA SSD and a PCIe SSD to measure performance gains according to their different storage speeds. Note that the values in the parenthesis denoted under each storage device indicate the basic sequential read/write and random read/write bandwidths in MB/s. We measured the bandwidths through a simple single-threaded application that triggers 512KB sequential I/Os and 4KB random I/Os to the device with `O_DIRECT`.

We compare F2FS with EXT4 [17], BTRFS [25] and NILFS2 [14]. EXT4 is a conventional update-in-place file system used in the default Android middleware. BTRFS is a copy-on-write file system, and NILFS2 is an alternative implementation of the original LFS.

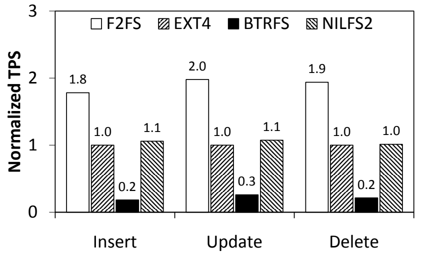
Table 3 summarizes our benchmarks and their characteristics in terms of generated I/O patterns, the number of touched files and their maximum size, the number of working threads, the ratio of reads and writes (R/W) and whether there are `fsync` system calls. For the mobile system, we execute and show the results of iozone [22], to study basic file I/O performance. Because mobile systems suffer from costly random writes with frequent

Table 3: Summary of benchmarks.

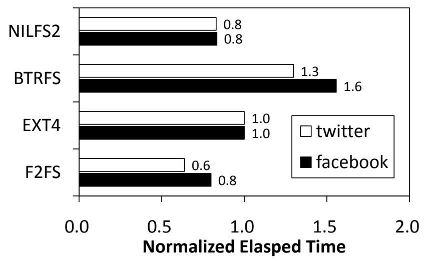
Target	Name	Workload	Files	File size	Threads	R/W	**`fsync`**
Mobile	iozone	sequential and random read/write	1	1G	1	50/50	N
	SQLite	random writes with frequent `fsync`	2	3.3MB	1	0/100	Y
	facebook	random writes with frequent `fsync`	579	852KB	1	1/99	Y
	twitter	generated by the given system call traces	177	3.3MB	1	1/99	Y
Server	videoserver	mostly sequential reads and writes	64	1GB	48	20/80	N
	fileserver	many large files with random writes	80,000	128KB	50	70/30	N
	varmail	many small files with frequent `fsync`	8,000	16KB	16	50/50	Y
	oltp	large files with random writes and `fsync`	10	800MB	211	1/99	Y



(a) iozone



(b) SQLite



(c) facebook and twitter

Figure 4: Performance results on the mobile system.

`fsync` calls, we run *mobibench* [8], a macro benchmark, to measure the SQLite performance. We also run two system call traces collected from real workloads, facebook and twitter.

For the server workloads, we make use of a synthetic benchmark called Filebench [19]. It emulates various file system workloads and allows for fast and intuitive system performance evaluation. We use four predefined workloads in the benchmark—videoserver, fileserver, varmail and oltp. These workloads differ from each other in terms of I/O randomness and `fsync` usage.

We first select videoserver—which issues mostly sequential reads and writes—to demonstrate that F2FS performs relatively well under such a naïve workload compared with other conventional file systems. Fileserver preallocates 80,000 files with 128KB data and subsequently starts 50 threads, each of which creates and deletes files randomly as well as reads and appends small data to randomly chosen files. This workload, thus, represents a scenario having many large files touched by buffered random writes and no `fsync`. Varmail creates and deletes a number of small files with `fsync`, while oltp preallocates ten large files and updates their data randomly with `fsync` with 200 threads in parallel.

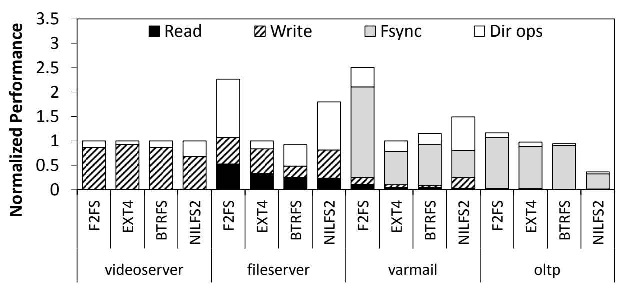
3.2 Results

This section gives the performance results and insights obtained from deep block trace level analysis. We examined various I/O patterns (i.e., read, write, `fsync` and discard[3]), amount of I/Os and request size distribution. For intuitive and consistent comparison, we normalize performance results against the EXT4 performance. Before presenting results, we note that the performance depends basically on the speed gap between sequential and random I/Os. In the case of the mobile system that has low computing power and a slow storage, I/O pattern and its quantity are the major performance factors. For the server system, with a relatively abundant main memory pool, high computing power, and a fast storage device, CPU efficiency with the instruction execution overheads and lock contention become an additional critical factor along with the I/O characteristics.

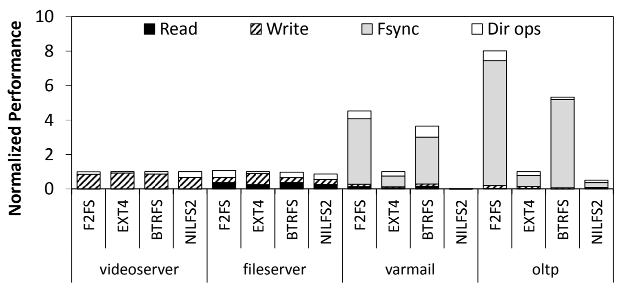
3.2.1 Performance on the Mobile System

Figure 4(a) shows the iozone results of sequential read/write (SR/SW) and random read/write (RR/RW) bandwidths on a single 1GB file. In the SW case, NILFS2 shows performance degradation of nearly 50% over EXT4 since it triggers expensive synchronous writes periodically, according to its own data flush policy. In the RW case, F2FS performs 3.1× better than EXT4, since it successfully turns over 90% of 4KB random writes into 512KB sequential writes. BTRFS also performs well (1.8×) as it produces sequential writes through the copy-on-write policy. While NILFS2 trans-

[3]A discard command gives a hint to the underlying flash storage device that the specified address range has no valid data. This command is sometimes called “trim” or “unmap”.



(a) SATA SSD



(b) PCIe SSD

Figure 5: Performance results on the server system.

forms random writes to sequential writes, it gains only 10% improvement due to the costly synchronous writes. Furthermore, it issues up to 30% more write requests than other file systems. In the RR case, all file systems show comparable performance. BTRFS shows slightly lower performance due to its tree indexing overheads.

Figure 4(b) gives the SQLite performance measured in transactions per second (TPS), normalized against that of EXT4. We measure three types of transactions—insert, update and delete—on a DB comprised of 1,000 records under the write ahead logging (WAL) journal mode. This journal mode is considered the fastest in SQLite. F2FS shows significantly better performance than other file systems and outperforms EXT4 by up to 2×. For the SQLite workload with frequent `fsync` operations, the roll-forward recovery policy of F2FS is shown to produce huge benefits. In fact, F2FS reduces the amount of data writes by about 46% over EXT4 in all the examined cases. Due to the heavy indexing overheads, BTRFS writes 3× more data than EXT4, resulting in performance degradation of nearly 80%. NILFS2 achieves similar performance with nearly identical amount of data writes compared to EXT4.

Figure 4(c) shows the normalized elapsed times to complete replaying the facebook and twitter traces on our mobile system. The traces are a series of system calls collected while running the facebook and twitter applications under a realistic usage scenario [8]. Because these applications resort to SQLite for storing and managing data, F2FS reduces the elapsed time by as much as 40% (twitter) and 20% (facebook) over EXT4.

3.2.2 Performance on the Server System

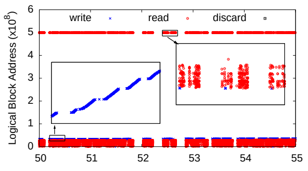
For the results we report in this subsection, we run Filebench with the predefined scripts for 300 seconds each. In the experiments, the performance is measured in operations per second (ops/s). The benchmark also reveals the CPU time and the latency of each system call consumed during the test.

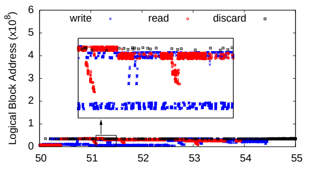
Figure 5 plots the performance of the studied file systems using the SATA and PCIe SSDs. Each bar indicates the normalized ops/s (i.e., relative performance or performance improvement if the bar has a value larger than 1). We break down the performance into contributions made by read, write, `fsync` and directory operations; for example, the varmail and oltp results are tightly coupled with the `fsync` performance, while the videoserver results depend highly on the write performance.

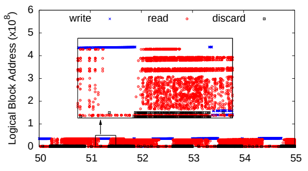
As mentioned above, videoserver generates mostly sequential reads and writes, and all the results, regardless of the device used, expose no performance gaps among the file systems. This demonstrates that F2FS has no performance regression for normal sequential I/Os.

Fileserver has different I/O patterns; Figure 6 compares the block traces obtained from all the file systems on the SATA SSD. As shown, this workload produces a number of random reads and buffered random writes while serving directory operations, as well as random appends. A closer examination finds that only 0.9% of all write requests generated by EXT4 are for 512KB, while F2FS has 6.9%. Another finding is that EXT4 issues many small discard commands, resulting in visible command processing overheads; it trims two thirds of all the block addresses covered by data writes and nearly 60% of all discard commands were for an address space smaller than 256KB in size. In contrast, F2FS discards obsolete spaces in the unit of segments only when checkpointing is triggered; it trims 38% of block address space with no small discard commands. These differences lead to the 2.4× performance gain of F2FS over EXT4, as shown in Figure 5(a).

On the other hand, BTRFS degrades the performance by 8% over EXT4, since it issues 512KB data writes in only 3.8% of all the write requests. In addition, it trims out 47% of block address space with small discard commands (corresponding to 75% of all discard commands)


(a) F2FS


(b) EXT4


(c) BTRFS

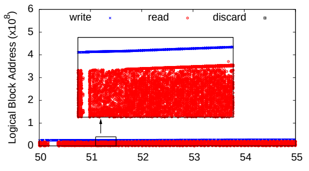

(d) NILFS2

Figure 6: Block traces of the fileserver workload according to the running time in seconds.

during the read service time as shown in Figure 6(c). In the case of NILFS2, as many as 78% of its write requests are for 512KB as illustrated in Figure 6(d). However, its periodic synchronous data flushes limited the performance gain over EXT4 to 1.8×. On the PCIe SSD, all the file systems perform rather similarly, and F2FS gained only 8% over EXT4. This is because the PCIe SSD performs concurrent buffered writes well.

In the varmail case, F2FS outperforms EXT4 by up to 2.5× on the SATA SSD and 4.5× on the PCIe SSD, respectively. Since varmail generates many small writes with concurrent `fsync`, the result again underscores the efficiency of `fsync` processing in F2FS. An interesting result is that BTRFS shows a higher performance gain on the PCIe SSD—3.6× vs. 1.5× (on the SATA SSD). This is due to the different bottleneck points in effect for the SSDs; since the PCIe SSD serves I/O quickly, the bottleneck moves to the cache flush commands triggered by `fsync`. BTRFS merges and skips redundant cache flush commands at the file system level, which improved the performance more dramatically on the PCIe SSD. (F2FS also implements cache flush merging.) Lastly, NILFS2 shows the worst performance due to its heavy data flush and `fsync` implementation.

The oltp workload generates a large number of random writes and `fsync` calls on a single 800MB database file (unlike varmail, which touches many small files). F2FS shows significant performance advantages over EXT4—1.2× on the SATA SSD and 8× on the PCIe SSD. Like it did for varmail, BTRFS greatly outperforms EXT4 on the PCIe SSD, by 5.3×, thanks to cache flush merging.

Our results so far have clearly demonstrated the relative effectiveness of the overall design and implementation of F2FS. We will now examine the impact of two design choices on logging.

3.2.3 Multi-head Logging Effect

This section studies the effectiveness of the multi-head logging policy of F2FS. Rather than presenting extensive evaluation results that span many different workloads, we focus on an experiment that captures the intuitions of our design. The metric used in this section is the *number of valid blocks* in a given dirty segment before cleaning. If hot and cold data separation is done perfectly, a dirty segment would have either zero valid blocks or 512 valid blocks (i.e., the maximum number of valid blocks in a segment under the default configuration). An aged dirty segment would carry zero valid blocks in it if all the (hot) data stored in the segment have been invalidated. By comparison, a dirty segment full of valid blocks is likely keeping cold data.

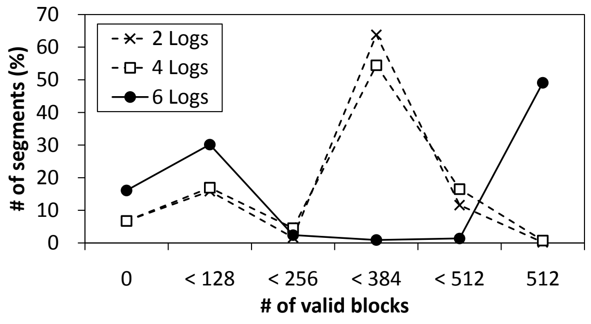
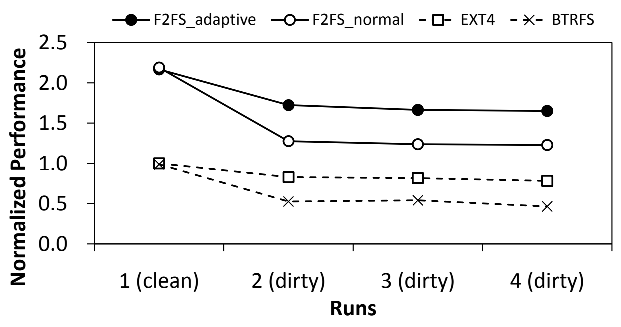


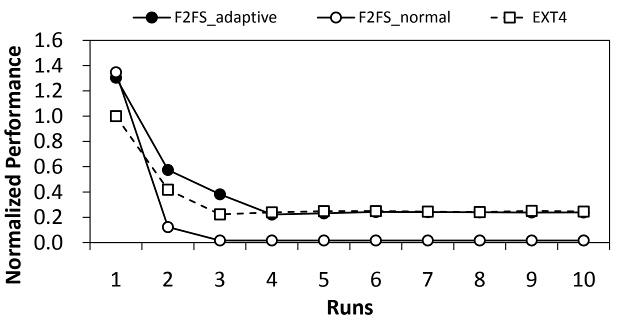
Figure 7: Segment distribution according to the number of valid blocks in segments.

In our experiment, we run two workloads simultaneously: varmail and copying of jpeg files. Varmail employs 10,000 files in total in 100 directories and writes 6.5GB of data. We copy 5,000 jpeg files of roughly 500KB each, hence resulting in 2.5GB of data writes. Note that F2FS statically classifies jpeg files as cold data. After these workloads finish, we count the number of valid blocks in all dirty segments. We repeat the experiment as we vary the number of logs from 2 to 6.

Figure 7 plots the result. With two logs, over 75% of all segments have more than 256 valid blocks while "full segments" with 512 valid blocks are very few. Because the two-log configuration splits only data segments (85% of all dirty segments) and node segments (15%), the effectiveness of multi-head logging is fairly limited. Adding two more logs changes the picture somewhat; it reduces the number of segments having fewer than 384 valid blocks. Instead, relatively full segments increase in number. Lastly, with six logs, we clearly see the benefits of hot and cold data separation; the number of prefree segments having zero valid blocks and the number of full segments increase significantly. In turn, the number



(a) fileserver (random operations) on a device filled up 94%.



(b) iozone (random updates) on a device filled up 100%.

Figure 8: Worst-case performance drop ratio under file system aging.

of partially full segments is reduced to a low level. An obvious impact of this bimodal (or bathtub-shaped) dirty segment distribution is improved efficiency of cleaning (because the cost of cleaning depends on the number of valid blocks in a victim segment).

We make several observations before we close this section. First, the result shows that more logs, allowing finer separation of data temperature, generally bring more benefits. However, in the particular experiment we performed, the benefit of four logs over two logs was rather insignificant. If we separate cold data from hot and warm data (as defined in Table 1) rather than hot data from warm and cold data (default), the result would look quite different. Second, because the number of valid blocks in dirty segments will gradually decrease over time, high points of 2-log and 4-log configurations in Figure 7 (segments with many valid blocks) will shift to left (at a different speed according to the chosen logging configuration). Hence, if we age the file system, the benefit of multi-head logging will become more visible. Fully studying the effect of multi-head logging based on these observations is beyond the scope of this paper.

3.2.4 Adaptive Logging Performance

This section answers the question: *How effective is the F2FS adaptive logging policy with threaded logging?* By default, F2FS switches to threaded logging from normal logging when the number of free sections falls down below 5% of the total number of sections. We compare this default configuration ("F2FS_adaptive") with "F2FS_normal", which sticks to the normal logging policy all the time. For experiments, we design and perform the following two intuitive tests on the SATA SSD.

• **fileserver test.** This test first fills up the target storage partition 94%, with hundreds of 1GB files. The test then runs the fileserver workload four times and measures the performance trends (Figure 8(a)). As we repeat experiments, the underlying flash storage device as well as the file system get fragmented. Accordingly, the performance of the workload is supposed to drop, as is clear in the plot. Note that we were unable to perform this test with NILFS2 as it stopped with a "no space" error report.

EXT4 showed the mildest performance hit—17% between the first and the second round. By comparison, BTRFS and F2FS (especially F2FS_normal) saw a more severe performance drop of 22% and 48% each, as they do not find enough sequential space. On the other hand, F2FS_adaptive serves 51% of total writes with threaded logging and successfully limited the performance hit in the second round to 22% (comparable to BTRFS and not too far from EXT4). As the result, F2FS maintained the performance improvement ratio of two or more over EXT4 across the board. All the file systems were shown to sustain performance beyond the second round.

Further examination reveals that F2FS_normal writes 27% more data than F2FS_adaptive due to foreground cleaning. The large performance hit on BTRFS is due partly to the heavy usage of small discard commands.

• **iozone test.** This test first creates sixteen 4GB files and additional 1GB files until it fills up the device capacity (∼100%). Then it runs iozone on the 4GB files, which measures the random write performance for 512MB random updates in each of the sixteen files. We repeat this step ten times, which turns out to be quite harsh, as both BTRFS and NILFS2 failed to complete with a "no space" error. Note that from the theoretical viewpoint, EXT4—an update-in-place file system—would perform the best in this test because EXT4 issues random writes without creating additional file system metadata. Also note that this workload fragments the data in the storage device and the storage performance would suffer as the workload triggers repeated internal GC operations.

Under EXT4, the performance drop was about 75%. In the case of F2FS_normal, as expected, the performance drops down to a very low level (of less than 5% of

EXT4 from round 3) as both the file system and the storage device keep busy cleaning fragmented capacity to reclaim new space for logging. F2FS_adaptive is shown to handle the situation much more gracefully; it performs better than EXT4 in the first few rounds (when fragmentation was relatively insignificant) and shows performance very similar to that of EXT4 as the experiment advances with more and more random writes.

4 Related Work

This section discusses prior work related to ours in three categories—log-structured file systems, file systems targeting flash memory, and optimizations specific to FTL.

4.1 Log-Structured File Systems (LFS)

Much work has been done on log-structured file systems (for HDDs), beginning with the original LFS proposal by Rosenblum et al. [26]. Wilkes et al. proposed a hole plugging method in which valid blocks of a victim segment are moved to *holes*, i.e., invalid blocks in other dirty segment [29]. Matthews et al. proposed an adaptive cleaning policy where they choose between a normal logging policy and a hole-plugging policy based on cost-benefit evaluation [18]. Oh et al. [23] demonstrated that threaded logging provides better performance in a highly utilized volume. F2FS has been tuned on the basis of the prior work and real-world workloads and devices.

A number of studies focus on separating hot and cold data. Wang and Hu [27] proposed to distinguish active and inactive data in the buffer cache, instead of writing them to a single log and separating them during cleaning. They determine which data is active by monitoring access patterns. Hylog [28] adopts a hybrid approach; it uses logging for hot pages to achieve high random write performance, and overwriting for cold pages to reduce cleaning cost.

SFS [20] is a file system for SSDs implemented based on NILFS2. Like F2FS, SFS uses logging to eliminate random writes. To reduce the cost of cleaning, they separate hot and cold data in the buffer cache, like [27], based on the "update likelihood" (or hotness) measured by tracking write counts and age per block. They use iterative quantization to partition segments into groups based on the measured hotness.

Unlike the hot/cold data separation methods that resort to run-time monitoring of access patterns [20, 27], F2FS estimates update likelihood using information readily available, such as file operation (append or overwrite), file type (directory or regular file), and file extensions. While our experimental results show that the simple approach we take is fairly effective, more sophisticated run-time monitoring approaches can be incorporated in F2FS to fine-track data temperature.

4.2 Flash Memory File Systems

A number of file systems have been proposed and implemented for embedded systems that use raw NAND flash memories as storage [30, 3, 1, 6, 13]. These file systems directly access NAND flash memories while addressing all the chip-level issues such as wear-leveling and bad block management. Unlike these systems, F2FS targets flash storage devices that come with a dedicated controller and firmware (FTL) to handle low-level tasks. Such flash storage devices are becoming more commonplace in recent systems.

William et al. proposed the direct file system (DFS) [9], which leverages special support from host-run FTL, including atomic update interface and very large logical address space, to simplify the file system design. DFS is however limited to specific flash devices and system configurations and is not open source.

4.3 FTL Optimizations

There has been much work aiming at improving random write performance at the FTL level, sharing some design strategies with F2FS. Most FTLs use a log-structured update approach to overcome the no-overwrite limitation of flash memory. DAC [5] provides a page-mapping FTL that clusters data based on update frequency by monitoring accesses at run time. To reduce the overheads of large page mapping tables, DFTL [7] dynamically loads a portion of the page map into working memory on demand and offers the random-write benefits of page mapping for devices with limited RAM.

Hybrid mapping (or log block mapping) is an extension of block mapping to improve random writes [16, 24, 12]. It has a smaller mapping table than page mapping while its performance can be as good as page mapping for workloads with substantial access locality.

5 Concluding Remarks

F2FS is a full-fledged Linux file system designed for modern flash storage devices and is slated for wider adoption in the industry. This paper describes key design and implementation details of F2FS. Our evaluation results underscore how our design decisions and trade-offs lead to performance and lifetime advantages, over other existing file systems. F2FS is fairly young—it was incorporated in Linux kernel 3.8 in late 2012. We expect new optimizations and features will be continuously added to the file system.

References

[1] Unsorted block image file system. `http://www.linux-mtd.infradead.org/doc/ubifs.html`.

[2] Using databases in android: SQLite. `http://developer.android.com/guide/topics/data/data-storage.html#db`.

[3] Yet another flash file system. `http://www.yaffs.net/`.

[4] A. B. Bityutskiy. JFFS3 design issues. `http://www.linux-mtd.infradead.org`, 2005.

[5] M.-L. Chiang, P. C. Lee, and R.-C. Chang. Using data clustering to improve cleaning performance for flash memory. *Software-Practice and Experience*, 29(3):267–290, 1999.

[6] J. Engel and R. Mertens. LogFS-finally a scalable flash file system. In *Proceedings of the International Linux System Technology Conference*, 2005.

[7] A. Gupta, Y. Kim, and B. Urgaonkar. *DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings*, volume 44. ACM, 2009.

[8] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won. I/O stack optimization for smartphones. In *Proceedings of the USENIX Anual Technical Conference (ATC)*, pages 309–320, 2013.

[9] W. K. Josephson, L. A. Bongo, K. Li, and D. Flynn. DFS: A file system for virtualized flash storage. *ACM Transactions on Storage (TOS)*, 6(3):14:1–14:25, 2010.

[10] A. Kawaguchi, S. Nishioka, and H. Motoda. A flash-memory based file system. In *Proceedings of the USENIX Anual Technical Conference (ATC)*, pages 155–164, 1995.

[11] H. Kim, N. Agrawal, and C. Ungureanu. Revisiting storage for smartphones. *ACM Transactions on Storage (TOS)*, 8(4):14, 2012.

[12] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho. A space-efficient flash translation layer for compactflash systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, 2002.

[13] J. Kim, H. Shim, S.-Y. Park, S. Maeng, and J.-S. Kim. Flashlight: A lightweight flash file system for embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 11(1):18, 2012.

[14] R. Konishi, Y. Amagai, K. Sato, H. Hifumi, S. Kihara, and S. Moriai. The Linux implementation of a log-structured file system. *ACM SIGOPS Operating Systems Review*, 40(3):102–107, 2006.

[15] S. Lee, D. Shin, Y.-J. Kim, and J. Kim. LAST: locality-aware sector translation for NAND flash memory-based storage systems. *ACM SIGOPS Operating Systems Review*, 42(6):36–42, 2008.

[16] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(3):18, 2007.

[17] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux Symposium*, volume 2, pages 21–33. Citeseer, 2007.

[18] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson. Improving the performance of log-structured file systems with adaptive methods. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 238–251, 1997.

[19] R. McDougall, J. Crase, and S. Debnath. Filebench: File system microbenchmarks. `http://www.opensolaris.org`, 2006.

[20] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom. SFS: Random write considered harmful in solid state drives. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 139–154, 2012.

[21] P. Mochel. The sysfs filesystem. In *Proceedings of the Linux Symposium*, page 313, 2005.

[22] W. D. Norcott and D. Capps. Iozone filesystem benchmark. *URL: www.iozone.org*, 55, 2003.

[23] Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh. Optimizations of LFS with slack space recycling and lazy indirect block update. In *Proceedings of the Annual Haifa Experimental Systems Conference*, page 2, 2010.

[24] C. Park, W. Cheon, J. Kang, K. Roh, W. Cho, and J.-S. Kim. A reconfigurable FTL (flash translation layer) architecture for NAND flash-based applications. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(4):38, 2008.

[25] O. Rodeh, J. Bacik, and C. Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):9, 2013.

[26] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[27] J. Wang and Y. Hu. WOLF: A novel reordering write buffer to boost the performance of log-structured file systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 47–60, 2002.

[28] W. Wang, Y. Zhao, and R. Bunt. Hylog: A high performance approach to managing disk layout. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 144–158, 2004.

[29] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems (TOCS)*, 14(1):108–136, 1996.

[30] D. Woodhouse. JFFS: The journaling flash file system. In *Proceedings of the Ottawa Linux Symposium*, 2001.