F2FS: A New File System for Flash Storage

Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho

S/W Development Team
Memory Business
Samsung Electronics Co., Ltd.

Abstract

F2FS is a Linux file system designed 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 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). It reduces elapsed time of several realistic workloads by up to 40%. On a server system, F2FS is shown to perform better than EXT4 by up to 2.5× (SATA SSD) and 1.8× (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. [21] 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. [12] 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 the random 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 [27] and/or the copy-on-write strategy. For example, one might anticipate file systems like BTRFS [26] and NILFS2 [15] would perform well on NAND flash SSDs; unfortunately, they do not consider the characteristics of flash storage devices and are inevitably suboptimal 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 present the design and implementation of 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 its design and implementation.

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.

• **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, making performance degradation due to multiple logging (vs. single-segment logging) insignificant.

• **Adaptive logging (Section 2.6).** F2FS builds basically on append-only logging to turn random writes into sequential ones. 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.

In a nutshell, F2FS builds on the concept of LFS but deviates significantly from the original LFS proposal with 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.

In the remainder of this paper, Section 2 first describes 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.

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

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 fixed-size *segments*. The segment is a basic unit of management in F2FS and is used to determine the initial file system metadata layout.

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 are further discussed in Section 2.4 and 2.5.

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—a recovery point after a sudden power-off event (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 (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 blocks in the "Main" area (see below). The SIT information is retrieved to select victim segments and identify valid blocks in them during the cleaning process (Section 2.5).

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

• **Segment Summary Area (SSA)** stores summary entries representing the owner information of all blocks in the Main area, such as parent inode number and its

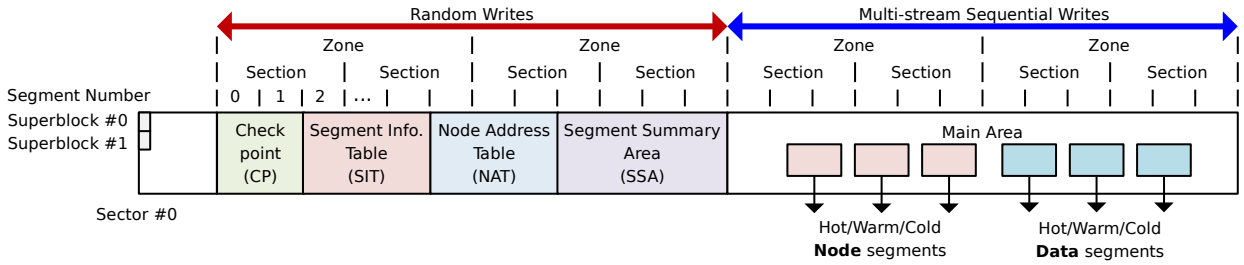


Figure 1: On-disk layout of F2FS.

node/data offsets. The SSA entries identify parent node blocks before migrating valid blocks during cleaning.

• **Main Area** is filled with 4KB blocks. Each block is allocated and typed to be *node* or *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. 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 data can be retrieved from the Main area, with indices obtained via the corresponding file structure.

2.2 File Structure

The original LFS introduced *inode map* to translate an inode number to an on-disk location. In comparison, F2FS utilizes the "node" structure that extends the inode map to locate more indexing blocks. Each node block has a unique identification number, "node ID". By using node ID as an index, NAT serves the physical locations of all node blocks. A node block represents one of three types: inode, direct and indirect node. An inode block contains a 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 eliminate update propagation (i.e., "wandering tree" problem [27]). In the traditional LFS design, if a leaf data is updated, its direct and indirect pointer blocks are updated

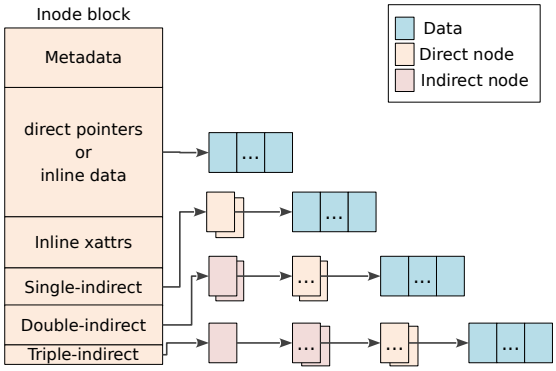


Figure 2: File structure of F2FS.

recursively. F2FS, however, only updates one direct node block and its NAT entry, effectively addressing the wandering tree problem. For example, when a 4KB data is appended to a file of 8MB to 4GB, the LFS updates two pointer blocks recursively while F2FS updates only one direct node block (not considering cache effects). For files larger than 4GB, the LFS updates one more pointer block (three total) while F2FS still updates only one.

An inode block contains direct pointers to the file's data blocks, two single-indirect pointers, two double-indirect pointers and one triple-indirect pointer. F2FS supports *inline data* and *inline extended attributes*, which embed small-sized data or extended attributes in the inode block itself. Inlining reduces space requirements and improve I/O performance. Note that many systems have small 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

In F2FS, 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 carries a hash value, inode number, length of a file name and file type (e.g., normal file, directory and sym-

bolic link). A directory file constructs multi-level hash tables to manage a large number of dentries efficiently.

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 an *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 large directories are preferred (e.g., in a server environment), users can configure F2FS to initially allocate space for many dentries. With a larger hash table at low levels, F2FS reaches 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 hotter than indirect node blocks since they are updated much more frequently. 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:

• **Data blocks moved by cleaning** (see Section 2.5). Since they have remained valid for an extended period of time, we expect they will remain so in the near future.
• **Data blocks labeled "cold" by the user**. 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 registered file extensions.

By default, F2FS activates six logs open for writing. The user may adjust the number of write streams to two or four at 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. With only two logs, F2FS allocates 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 an FTL, with a view to mitigating the garbage collection (GC) overheads.[2] FTL algorithms are largely classified into three groups (block-associative, set-associative and fully-associative) according to 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 [18]. The log flash block can be used exclusively for a single data flash block (block-associative) [13], for all data flash blocks (fully-associative) [17], or for a set of contiguous data flash blocks (set-associative) [24]. Modern FTLs 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 strategy is expected to be effective for set-associative FTLs. Multi-head logging is also a natural match with the recently proposed "multi-streaming" interface [10].

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	Directory entry blocks
	Warm	Data blocks made by users
	Cold	Data blocks moved by cleaning; Cold data blocks specified by users; Multimedia file data

2.5 Cleaning

Cleaning is a process to reclaim scattered and invalidated blocks, and secures 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 sustained performance of F2FS (and any LFS in general). In F2FS, cleaning is done in the unit of a section.

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 three steps:

[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.

(1) Victim selection. The cleaning process starts first to identify a victim section among non-empty sections. There are two well-known policies for victim selection during LFS cleaning—*greedy* and *cost-benefit* [11, 27]. The greedy policy selects a section with the smallest number of valid blocks. Intuitively, this policy controls overheads of migrating valid blocks. F2FS adopts the greedy policy for its foreground cleaning to minimize the latency visible to applications. Moreover, F2FS reserves a small unused capacity (5% of the storage space by default) so that the cleaning process has room for adequate operation at high storage utilization levels. Section 3.2.4 studies the impact of utilization levels on cleaning cost.

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 its utilization but also its "age". F2FS infers the age of a section by averaging the age of segments in the section, which, in turn, can be obtained from their last modification time recorded in SIT. With the cost-benefit policy, F2FS gets another chance to separate hot and cold data.

(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 parent node blocks containing their indices from the SSA information. If 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. Background cleaning does not kick in when normal I/O or foreground cleaning is in progress.

(3) Post-cleaning process. After all valid blocks are migrated, a victim section is registered as a candidate to become a new free section (called a "pre-free" section in F2FS). After a checkpoint is made, the section finally becomes a free section, to be reallocated. We do this because if a pre-free section is reused before checkpointing, the file system may lose the data referenced by a previous checkpoint when unexpected power outage 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 high cleaning overheads, resulting in a serious performance drop (quantified to be over 90% under harsh conditions, see Section 3.2.5). On the other hand, threaded logging writes blocks to *holes* (invalidated, obsolete space) in existing dirty segments. This policy requires no cleaning operations, but triggers random writes and may degrade performance as a result.

F2FS implements both policies and switches between them dynamically according to the file system status. Specifically, if there are more than k clean sections, where k is a pre-defined threshold, normal logging is initiated. Otherwise, threaded logging is activated. k is set to 5% of total sections by default and can be configured.

There is a chance that 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, since all holes in a dirty segment are filled first before F2FS searches for more in other dirty segments. Lee et al. [16] demonstrate that flash storage devices show better random write performance with strong spatial locality. F2FS gracefully gives up normal logging and turns to threaded logging for higher sustained performance, as will be shown in Section 3.2.5.

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 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** are written at the beginning and the end of the pack, respectively. F2FS maintains in the header and footer a version number that is incremented on creating a checkpoint. The version number discriminates the latest stable pack between two recorded packs during the mount time;

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

• **NAT and SIT journals** contain a small number of re-

cently modified entries of NAT and SIT to avoid frequent NAT and SIT updates;

• **Summary blocks of active segments** consist of in-memory 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 (e.g., this can happen when two processes open a common file and one process deletes it), 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 is dropped.

Likewise, F2FS also manages two sets of NAT and SIT blocks, distinguished by the NAT and SIT bitmaps in each checkpoint pack. When it writes 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 implements a *NAT and SIT journal* within the checkpoint pack. 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 selecting 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 file and conduct `fsync` to guarantee durability. A naïve approach to supporting `fsync` would be to trigger checkpointing and recover data with the roll-back model. However, this approach leads to poor performance, as checkpointing involves writing all node and dentry blocks unrelated to the database file.

Table 2: Platforms used in experimentation. Numbers in parentheses 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 (NVMe) SSD 960GB: *(1,295, 922, 41, 254)*

F2FS implements an efficient roll-forward recovery mechanism to enhance `fsync` performance. The key idea is to write data blocks and their direct node blocks only, excluding 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 direct node blocks.

F2FS performs 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. **n** refers to the number of blocks updated since the last checkpoint. (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.

3 Evaluation

3.1 Experimental Setup

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

For the target systems, we back-ported F2FS from the 3.15-rc1 main-line kernel to the 3.4.5 and 3.14 kernel, respectively. In the mobile system, F2FS runs on a state-of-the-art eMMC storage. In the case of the server system, we harness a SATA SSD and a (higher-speed) PCIe

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-app	Random writes with frequent fsync	579	852KB	1	1/99	Y
	Twitter-app	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

SSD. Note that the values in the parentheses denoted under each storage device indicate the basic sequential read/write and random read/write bandwidth in MB/s. We measured the bandwidth through a simple single-thread application that triggers 512KB sequential I/Os and 4KB random I/Os with O_DIRECT.

We compare F2FS with EXT4 [18], BTRFS [26] and NILFS2 [15]. EXT4 is a widely used update-in-place file system. BTRFS is a copy-on-write file system, and NILFS2 is an 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 are subject to costly random writes with frequent fsync calls, we run *mobibench* [8], a macro benchmark, to measure the SQLite performance. We also replay two system call traces collected from the "Facebook" and "Twitter" application (each dubbed "Facebook-app" and "Twitter-app") under a realistic usage scenario [8].

For the server workloads, we make use of a synthetic benchmark called Filebench [20]. It emulates various file system workloads and allows for fast intuitive system performance evaluation. We use four pre-defined workloads in the benchmark—videoserver, fileserver, varmail and oltp. They differ in I/O pattern and fsync usage.

Videoserver issues mostly sequential reads and writes. Fileserver pre-allocates 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 pre-allocates ten large files and updates their data randomly with fsync with 200 threads in parallel.

3.2 Results

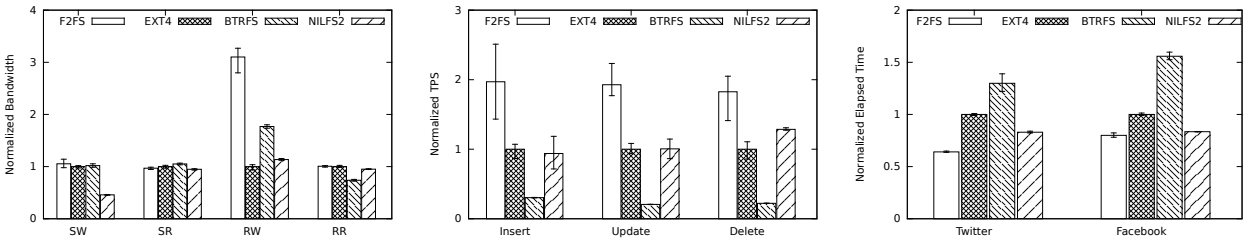
This section gives the performance results and insights obtained from deep block trace level analysis. We examined various I/O patterns (i.e., read, write, fsync and discard[3]), amount of I/Os and request size distribution. For intuitive and consistent comparison, we normalize performance results against EXT4 performance. We note that 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, CPU efficiency with instruction execution overheads and lock contention become an additional critical factor.

3.2.1 Performance on the Mobile System

Figure 3(a) shows the iozone results of sequential read/write (SR/SW) and random read/write (RR/RW) bandwidth 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 turns over 90% of 4KB random writes into 512KB sequential writes (not directly shown in the plot). BTRFS also performs well (1.8×) as it produces sequential writes through the copy-on-write policy. While NILFS2 transforms random writes to sequential writes, it gains only 10% improvement due to costly synchronous writes. Furthermore, it issues up to 30% more write requests than other file systems. For RR, all file systems show comparable performance. BTRFS shows slightly lower performance due to its tree indexing overheads.

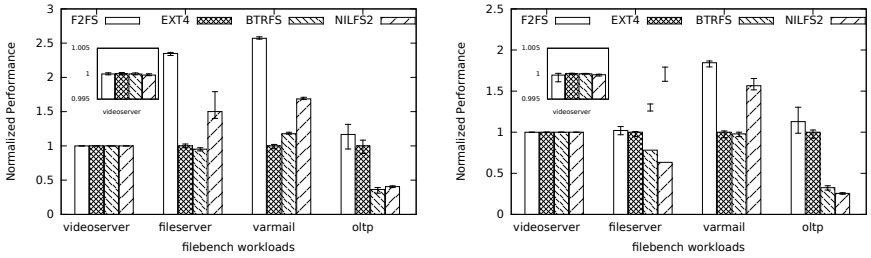
Figure 3(b) gives SQLite performance measured in transactions per second (TPS), normalized against that of EXT4. We measure three types of transactions—

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



(a) iozone (b) SQLite (c) Facebook-app and Twitter-app

Figure 3: Performance results on the mobile system.



(a) SATA SSD (b) PCIe SSD

Figure 4: Performance results on the server system.

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 this workload, the roll-forward recovery policy of F2FS produces huge benefits. In fact, F2FS reduces the amount of data writes by about 46% over EXT4 in all examined cases. Due to heavy indexing overheads, BTRFS writes 3× more data than EXT4, resulting in performance degradation of nearly 80%. NILFS2 achieves similar performance with a nearly identical amount of data writes compared to EXT4.

Figure 3(c) shows normalized elapsed times to complete replaying the Facebook-app and Twitter-app traces. They resort to SQLite for storing data, and F2FS reduces the elapsed time by 20% (Facebook-app) and 40% (Twitter-app) compared to EXT4.

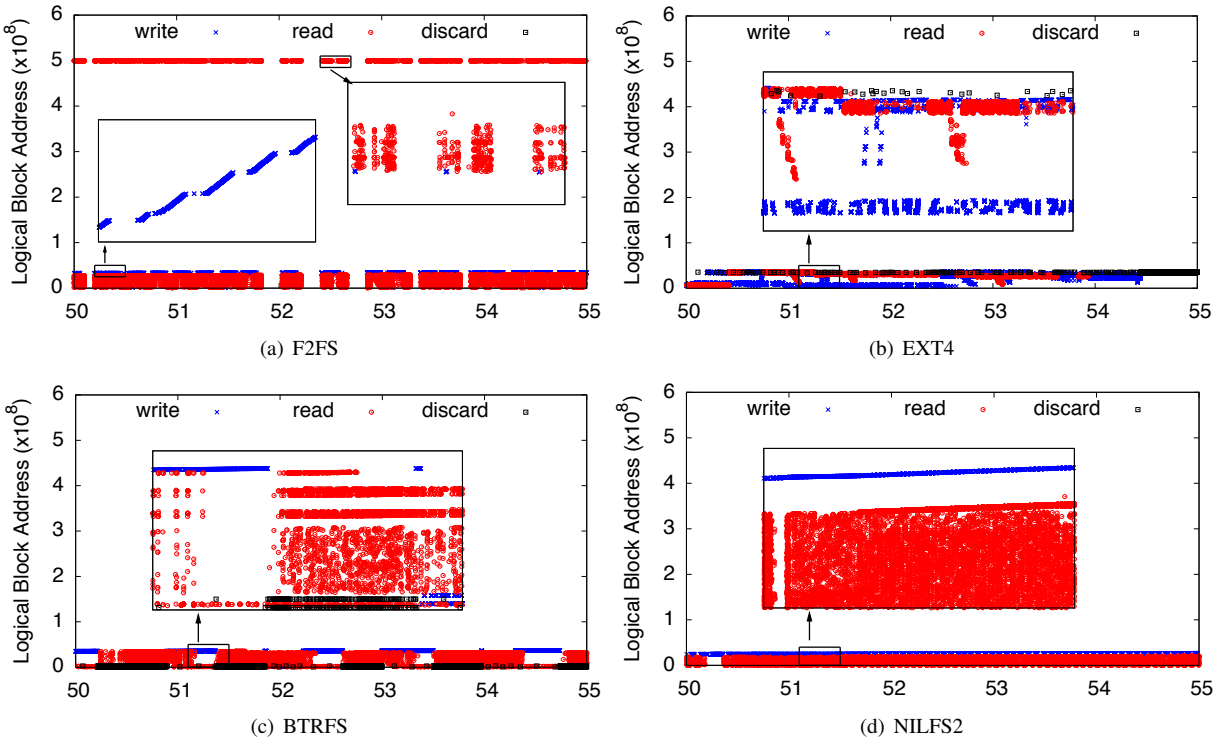
3.2.2 Performance on the Server System

Figure 4 plots performance of the studied file systems using SATA and PCIe SSDs. Each bar indicates normalized performance (i.e., performance improvement if the bar has a value larger than 1).

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

Fileserver has different I/O patterns; Figure 5 compares block traces obtained from all file systems on the SATA SSD. A closer examination finds that only 0.9% of all write requests generated by EXT4 are for 512KB, while F2FS has 6.9% (not directly shown in the plot). Another finding is that EXT4 issues many small discard commands and causes visible command processing overheads, especially on the SATA drive; it trims two thirds of all 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 a 2.4× performance gain (Figure 4(a)).

On the other hand, BTRFS degrades performance by 8%, since it issues 512KB data writes in only 3.8% of all write requests. In addition, it trims 47% of block address space with small discard commands (corresponding to 75% of all discard commands) during the read service time as shown in Figure 5(c). In the case of NILFS2, as many as 78% of its write requests are for 512KB (Figure 5(d)). However, its periodic synchronous data flushes limited the performance gain over EXT4 to 1.8×. On the PCIe SSD, all file systems perform rather similarly. This is because the PCIe SSD used in the study performs concurrent buffered writes well.



(a) F2FS (b) EXT4

(c) BTRFS (d) NILFS2

Figure 5: Block traces of the fileserver workload according to the running time in seconds.

In the varmail case, F2FS outperforms EXT4 by 2.5× on the SATA SSD and 1.8× 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. BTRFS performance was on par with that of EXT4 and NILFS2 performed relatively well on the PCIe SSD.

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 measurable performance advantages over EXT4—16% on the SATA SSD and 13% on the PCIe SSD. On the other hand, both BTRFS and NILFS2 performed rather poorly on the PCIe drive. Fast command processing and efficient random writes on the PCIe drive appear to move performance bottleneck points, and BTRFS and NILFS2 do not show robust performance.

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 F2FS logging and cleaning policies.

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 the maximum number of valid blocks in a segment (512 under the default configuration). An aged dirty segment would carry zero valid blocks in it if all (hot) data stored in the segment have been invalidated. By comparison, a dirty segment full of valid blocks is likely keeping cold data.

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 written. 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 two to six.

Figure 6 gives 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, not shown) and node segments (15%), the effectiveness of multi-head logging is fairly limited. Adding two more logs changes the picture

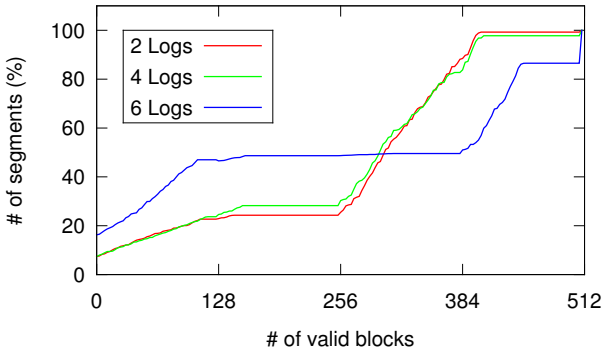


Figure 6: Dirty segment distribution according to the number of valid blocks in segments.

somewhat; it increases the number of segments having fewer than 256 valid blocks. It also slightly increases the number of nearly full segments.

Lastly, with six logs, we clearly see the benefits of hot and cold data separation; the number of pre-free segments having zero valid blocks and the number of full segments increase significantly. Moreover, there are more segments having relatively few valid blocks (128 or fewer) and segments with many valid blocks (384 or more). An obvious impact of this bimodal distribution is improved cleaning efficiency (as cleaning costs depend 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 different. Second, since the number of valid blocks in dirty segments will gradually decrease over time, the left-most knee of the curves in Figure 6 will move upward (at a different speed according to the chosen logging configuration). Hence, if we age the file system, we expect that multi-head logging benefits will become more visible. Fully studying these observations is beyond the scope of this paper.

3.2.4 Cleaning Cost

We quantify the impact of cleaning in F2FS in this section. In order to focus on file system level cleaning cost, we ensure that SSD level GC does not occur during experiments by intentionally leaving ample free space in the SSD. To do so, we format a 250GB SSD and obtain a partition of (only) 120GB.

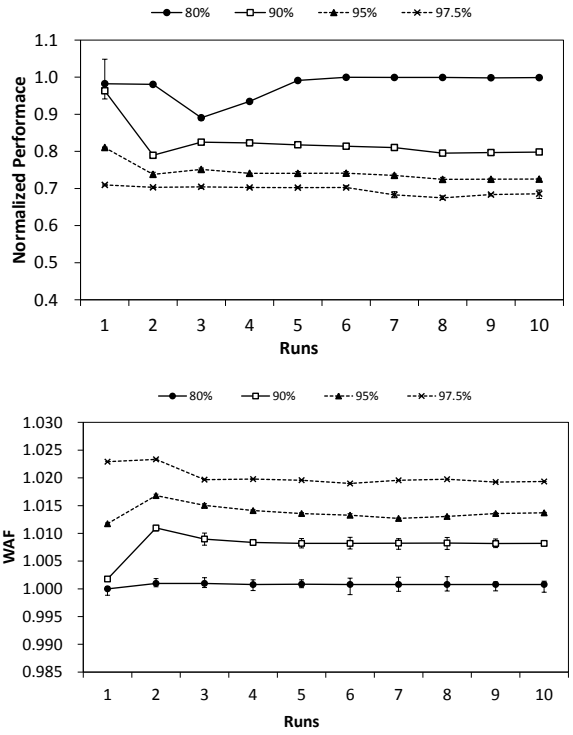
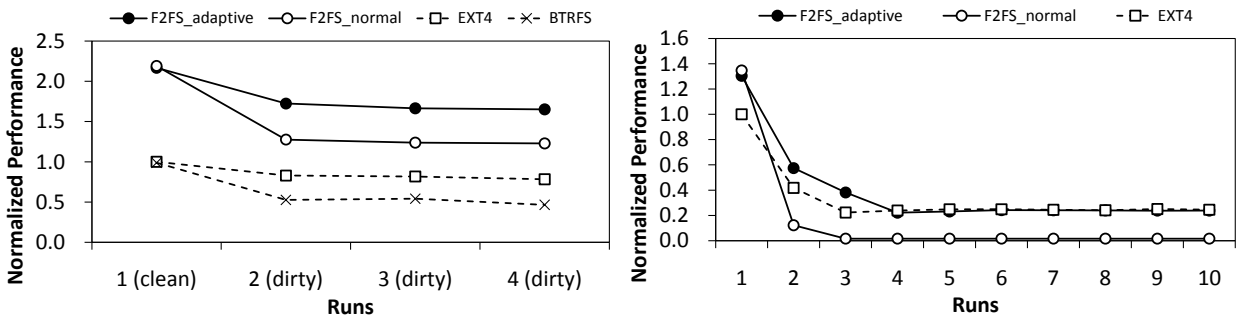


Figure 7: Relative performance (upper) and write amplification factor (lower) of the first ten runs. Four lines capture results for different file system utilization levels.

After reserving 5% of the space for overprovisioning (Section 2.5), we divide remaining capacity into "cold" and "hot" regions. We build four configurations that reflect different file system utilization levels by filling up the two regions as follows: 80% (60 (cold):20 (hot)), 90% (60:30), 95% (60:35) and 97.5% (60:37.5). Then, we iterate ten runs of experiments where each run randomly writes 20GB of data in 4KB to the hot region.

Figure 7 plots results of the first ten runs in two metrics: performance (throughput) and write amplification factor (WAF).[4] They are relative to results obtained on a clean SSD. We make two main observations. First, higher file system utilization leads to larger WAF and reduced performance. At 80%, performance degradation and WAF increase were rather minor. On the third run, the file system ran out of free segments and there was a performance dip. During this run, it switched to threaded logging from normal logging, and as the result, performance stabilized. (We revisit the effects of adaptive, threaded logging in Section 3.2.5.) After the third run, nearly all data were written via threaded logging, in place. In this case, cleaning is needed not for data, but

[4]Iterating 100 runs would not reveal further performance drops.



(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.

for recording nodes. As we raised utilization level from 80% to 97.5%, the amount of GC increased and the performance degradation became more visible. At 97.5%, the performance loss was about 30% and WAF 1.02.

The second observation is that F2FS does not dramatically increase WAF at high utilization levels; adaptive logging plays an important role of keeping WAF down. Note that threaded logging incurs random writes whereas normal logging issues sequential writes. While random writes are relatively expensive and motivates append-only logging as a preferred mode of operation in many file systems, our design choice (of switching to threaded logging) is justified because: cleaning could render very costly due to a high WAF when the file system is fragmented, and SSDs have high random write performance. Results in this section show that F2FS successfully controls the cost of cleaning at high utilization levels.

Showing the positive impact of background cleaning is not straightforward because background cleaning is suppressed during busy periods. Still, We measured over 10% performance improvement at a 90% utilization level when we insert an idle time of ten minutes or more between runs.

3.2.5 Adaptive Logging Performance

This section delves into 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 below 5% of total 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. 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 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 (not shown in the plot) and successfully limited performance degradation 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 to perform 4KB random writes on the sixteen 4GB files. The aggregate write volume amounts to 512MB per file. 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. On the other hand, a log-structured file system like F2FS may suffer high cleaning costs. Also note that this workload fragments the data in the storage device, and the storage performance would suffer as the workload triggers repeated

device-internal GC operations.

Under EXT4, the performance degradation was about 75% (Figure 8(b)). In the case of F2FS_normal, as expected, the performance drops 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 not severe) and shows performance very similar to that of EXT4 as the experiment advances with more random writes.

The two experiments in this section reveal that adaptive logging is critical for F2FS to sustain its performance at high storage utilization levels. The adaptive logging policy is also shown to effectively limit the performance degradation of F2FS due to fragmentation.

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. [27]. 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 [30]. 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 [19]. Oh et al. [23] demonstrated that threaded logging provides better performance in a highly utilized volume. F2FS has been tuned on the basis of prior work and real-world workloads and devices.

A number of studies focus on separating hot and cold data. Wang and Hu [28] 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 [29] 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 [21] 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 [28], 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 measured hotness.

Unlike the hot/cold data separation methods that resort to run-time monitoring of access patterns [21, 28], 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.

NVMFS is an experimental file system assuming two distinct storage media: NVRAM and NAND flash SSD [25]. The fast byte-addressable storage capacity from NVRAM is used to store hot and meta data. Moreover, writes to the SSD are sequentialized as in F2FS.

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 [1, 3, 6, 14, 31]. 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 more commonplace.

Josephson 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 [13, 17, 24]. 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 tradeoffs lead to performance 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.

Acknowledgment

Authors appreciate constructive comments of the reviewers and our shepherd, Ted Ts'o, which helped improve the quality of this paper. This work is a result of long and dedicated team efforts; without numerous contributions of prior F2FS team members (especially Jaegeuk Kim, Chul Lee, ByoungGeun Kim, Sehwan Lee, Seokyoung Ko, Dongbin Park and Sunghoon Park), this work would not have been possible.

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] J.-U. Kang, J. Hyun, H. Maeng, and S. Cho. The multi-streamed solid-state drive. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, Philadelphia, PA, 2014. USENIX Association.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

[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] S. Qiu and A. L. N. Reddy. NVMFS: A hybrid file system for improving random write in nand-flash SSD. In *IEEE 29th Symposium on Mass Storage Systems and Technologies, MSST 2013, May 6-10, 2013, Long Beach, CA, USA*, pages 1–5, 2013.

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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