Simplifying and Accelerating NOR Flash I/O Stack for RAM-Restricted Microcontrollers

Abstract

NOR flash has been increasingly popular for RAM-restricted microcontrollers due to its small package, high reliability, etc. To satisfy RAM restrictions, existing NOR flash file systems migrate their functionalities, i.e., block-level data organization and wear leveling (WL), from RAM to NOR flash. However, such fine-grained block-level management introduces frequent index updates and NOR flash scanning, leading to severe I/O amplification, which further deteriorates as they are decoupled in existing NOR flash file systems.

To address the problem, we propose NF2FS, a NOR flash-friendly file system. Our key insight is that applications running on NOR flash usually have (1) small file sizes, therefore block-based data organization can be converted to flat file layout (for fast file/dir scanning); (2) deterministic I/O patterns, thereby WL can be achieved through coarse file swapping. As a result, NF2FS relaxes data organization and WL to a coarse-grained file level, which are then cooperated within the file system. We implement NF2FS in FreeRTOS using a range of techniques, including the *all-logging layout*, along with efficient layout management approaches such as *dual bitmap space allocator* and *soft-update-like crash consistency*. Experiments suggest that NF2FS significantly outperforms existing works and can prevent quick NOR flash wear-out.

1 Introduction

With the emergence of the Internet of Things (IoT), the demand for NOR flash has increased dramatically [8, 9, 29]. Compared to traditional storage devices (e.g., NAND flash), NOR flash is small (MB level), byte-addressable, and deployed with RAM-restricted microcontroller units (MCU). Similar to NAND flash, NOR flash faces the erase-before-write problem. Consequently, these characteristics emphasize two primary goals for building a NOR flash file system: (1) to minimize RAM usage and (2) to prolong NOR flash lifespans.

Unfortunately, the widely used block-based file systems that serve for SSD or UFS [22] are unsuitable for NOR flash. The reason is that they rely on the hardware controller (e.g., FTL for WL and L2P mapping [21]) to provide a unified disk-like interface, as shown in Figure 1(a), while NOR flash has neither controller nor integrated RAM due to its low cost and density. Meanwhile, raw-NAND file systems, which aim to manage NAND flash without a controller, still fail to suit NOR flash. As shown in Figure 1(b), they aim to emulate hardware controllers using software (i.e., *soft controller*). Specifically, they build upon a multi-layer I/O stack with separated functionalities: the file system leverages block indexes to organize data, while the *soft controller* provides WL to balance device wear [3]. However, the two layers both deploy complex data structures (e.g., B-tree) in host RAM, contradicting NOR flash's RAM restriction.

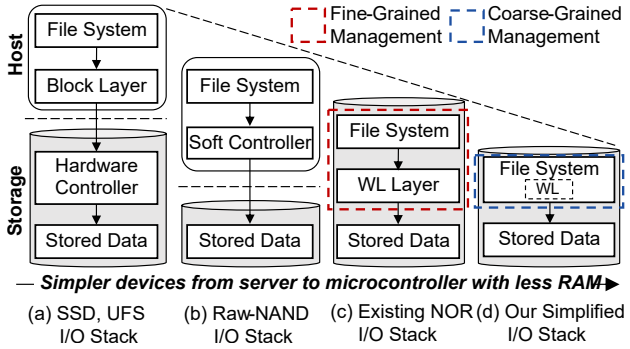


Figure 1. File Systems in Flash-based I/O Stacks. *Generally, file systems use indexes to organize data, while hardware controllers perform WL. Without hardware controllers, NAND file systems perform data organization and WL in host RAM. In contrast, NOR file systems incorporate the two functionalities into storage due to RAM limits.*

To address the problems, existing NOR flash file systems migrate their functionalities from host RAM to storage (by maintaining all data structures in NOR flash while leveraging a small portion of host RAMs for I/O), as shown in Figure 1(c). To accommodate in-place updating data structures in the file system and WL layer[1], they either arrange updates in an out-of-place manner or store WL metadata across blocks (i.e., a decentralized approach). For example, LittleFS [17] leverages copy-on-write (CoW) to update indexes, while SPIFFS [16] distributes program/erase (P/E) information to each block. Nevertheless, these approaches introduce severe I/O amplification, reducing both performance and lifespan. Specifically, (1) updating indexes leads to cascade pointer updating; (2) decentralizing P/E information forces block selection to scan all NOR flash blocks. Our experiment suggests that in write-intensive real-world workloads, LittleFS [17] and SPIFFS [16] spend up to 28.43× and 4.51× time compared to the *native file system* that directly operates raw NOR flash.

We observe that the key problems of existing NOR flash file systems lie in two aspects: (1) they manage NOR in a fine-grained block level even on the RAM-restricted platforms, causing I/O amplification; (2) they seperate WL functionality from the file system design, with both fine-grained block-level management, further deteriorating I/O amplification.

[1]WL layer and *soft controller* are interchangeable for NOR flash file systems as the *soft controller* in NOR flash file systems only has WL functionality.

We thus propose NF2FS, a NOR Flash-Friendly File System. As shown in Figure 1(d), NF2FS relaxes the management granularity of data organization and WL from a block to a file, and thus incorporates WL within the file system layer. This is based on our key insight that applications running on NOR flash usually have small file sizes and deterministic I/O patterns (i.e., similar and simple-to-predictable I/O patterns) [20, 27, 33]. Particularly, for MCU applications, configuration and log files are frequently updated [27, 33], while larger binaries are read-dominated [20]. This motivates us to convert block-based data organization to a flat file layout and achieve coarse WL through file swapping. As a result, NF2FS enables data organization to cooperate with WL and thus removes the separate WL layer.

To build NF2FS in NOR flash, we further propose a NOR-friendly *all-logging layout*, which serves all file system entities (e.g., files, directories, and file system metadata) as logically contiguous areas (i.e., *log areas*). These areas are updated in a logging manner to avoid cascade pointer modifications, which can be scanned for data retrieving (without block-level indexing) and swapped for WL. However, efficiently managing the all-logging layout is non-trivial. We further propose three techniques: (1) fast log area allocation with *dual bitmap* that exploits NOR-specific P/E characteristics; (2) efficient WL with *long-term sorted P/E array* that absorbs decentralized P/E information into a centralized and sorted array; (3) low-overhead crash consistency with *soft update-like metadata updates* that synchronously enforces NOR flash I/O order with atomic single-bit operations.

We implement NF2FS in FreeRTOS [25]. We build NORbench to emulate a range of real-world NOR flash workloads. Experiments based on NORbench suggest that, compared with existing NOR flash file systems, NF2FS can reduce up to 74.06% and 99.49% runtime in write-intensive real-world workloads, can decrease 75.96%-98.70% P/E cycles (with balanced wear) for workloads with many small writes.

In summary, this paper makes three contributions:

- We reveal that existing NOR flash file systems' block-level data organization and WL can exacerbate I/O amplification and reduce the lifespan of NOR flash.
- We propose NF2FS, which employs an all-logging layout as the core technique to organize data and WL in a file granularity. NF2FS also introduces three techniques for efficient space allocation, WL, and crash consistency, with full consideration of NOR flash's P/E characteristics.
- Experiments show that NF2FS consistently outperforms existing NOR flash file systems (e.g., SPIFFS and LittleFS), while effectively preventing quick NOR flash wear-out.

2 Background

2.1 Demystifing NOR Flash

NOR Flash Market. NOR flash has gained increasing popularity with the rise of the Internet of Things (IoT) [14, 29, 40] and tinyML [35, 39]. According to IC Insights, the sales of NOR flash devices in 2021 have increased by 63% compared to 2020 [9]. Moreover, Yole forecasts that the NOR flash market will keep growing with a Compound Annual Growth Rate from 2021–2027 of 6%, similar to the NAND flash [8].

Table 1. NOR vs NAND. *Compared to NAND, NOR flash is popular for MCUs due to the following (first five) properties.*

Properties	NOR Flash [15, 19]	NAND Flash [13]
I/O Granularity	Byte	Page
Power Consumption	Low	High
eXecute-In-Place	Support	Not support
Reliability	High (No ECC[a])	Low (Need ECC)
Overall Price	Low	High[b]
Capacity	MB-level	GB-level
Lifespan (P/E Cycle)	Short (10K - 100K)	Long (100K - 1M)

[a]: Error correction code (e.g., CRC, the cyclic redundancy check).
[b]: Although NAND has a lower price per bit, it is more costly than NOR since even the smallest NAND is significantly larger than NOR [6].

NOR Flash Structure. NOR flash is a byte-addressable device that requires erase-before-write in the block (or sector) granularity. Compared to NAND flash, NOR flash has a lower density (e.g., 2–32 MB [15, 19]) and limited P/E cycles [6]. However, it is indispensable in IoT scenarios as it has (1) *low power consumption* thanks to its simple NOR gate [18], which is important in energy-sensitive IoT scenarios; (2) *eXecute-in-Place (XIP)* that allows NOR flash to be mapped into the memory space, enabling direct binary execution without loading; (3) *reliable data retention* that can retain data much longer than NAND without error correction code (ECC) [16, 29]; (4) *low overall price* that is suitable for small IoT devices without large storage requirements [6]. Table 1 summarizes the main differences between NOR and NAND.

NOR Flash P/E Characteristics. NOR gate introduces interesting P/E characteristics, referred to as *unidirectional bit updating*. Specifically, data stored in NOR flash are charge-dependent, where the bit value of 1 indicates no charge in the cell and vice versa. Therefore, changing the bit value of 1 to 0 requires charge injection (i.e., program or write), while turning the bit value of 0 to 1 requires eviction (i.e., erase). However, writing a bit value of 1 to a cell will not change its original value. Therefore, the in-place update is possible if all changed bit values are from 1 to 0, and we write bit values of 1 to bits that should remain unchanged. Table 2 summarizes the unique P/E characteristics using four examples.

2.2 NOR Flash with MCU Restrictions

Due to its small density and package size, NOR flash is typically deployed with a microcontroller that has extremely limited RAM resources (e.g., 32 KB [4]). Generally, MCU communicates with NOR flash via a serial peripheral interface (SPI). NOR flash serves as external storage for the following two common use cases: (1) *Code Storage.* Binaries and configuration files are uploaded to NOR flash and retrieved

Table 2. A Close Look to NOR Flash's P/E Characteristics. *The box [0] indicates the charge injection, while other programmed (or written) bits remain unchanged.*

Origin	Operation	End	Essence
00000000	Erase	11111111	Evict 8 charges
11111111	Program 01010101	[0]1[0]1[0]1[0]1	Inject 4 charges
01010101	Program 00010101	[000]1[0]1[0]1	Inject 5 charges
00010101	Program 11111010	00010[0][0][0]	Inject 2 charges

by MCU during system boot and runtime. These files can be updated through Over-The-Air (OTA) updates. Note that OTA updates are getting more crucial nowadays for timely security patches, feature enhancements, and bug fixes [5, 14]. (2) *Data Storage.* MCU logs sensor data and system logs periodically, which are temporarily persisted in NOR flash. Then, logged data will be uploaded to a "cloud" device, such as the server or smartphone, for durable storage and deeper analysis [7, 14, 18]. Note that using NOR flash's XIP property is also popular [28], but it is out of the scope of this paper.

Therefore, it is important to offload NOR awareness from MCU developers to file system software. However, building such an efficient and general file system is not straightforward, especially considering MCU's limited resources and NOR flash's characteristics. The restrictions are as below:

- *Low RAM usage.* MCU has limited RAMs. Therefore, NOR flash file systems should restrict RAM usage.
- *Balanced wear.* NOR flash has lower P/E cycles compared to NAND. However, write-intensive operations (e.g., logging) frequently wear some hot blocks but may never wear some cold blocks, leading to unbalanced wear. Therefore, it is crucial to perform WL and avoid quick wear-out.
- *Low software tax.* The file system should not introduce excessive software overheads that squander NOR flash I/O performance and reduce the real-time ability.

2.3 Existing NOR Flash File System

To satisfy the above goals, existing NOR flash file systems migrate traditional I/O stack functionalities, i.e., data organization (in the file system layer) and WL (in the soft controller), into NOR flash, as shown in Figure 1(c). Among these works, SPIFFS [16] and LittleFS [17] are two representatives.

LittleFS [17] consists of a CoW-based file system and a greedy WL layer, as shown in Figure 2(a). The file system layer organizes data using skip lists to enable real-time binary execution (i.e., fast random read). However, this design will lead to severe I/O amplification for write-intensive workloads. Specifically, extending the skip list should update list nodes (which are embedded in data blocks) in a copy-on-write (CoW) manner, which introduces cascade pointer updates (or wandering tree [38]) to rewrite relevant data blocks (which store to be updated list nodes) and other blocks that

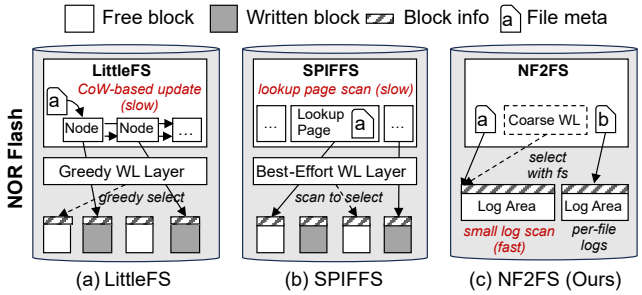


Figure 2. Comparisons of NOR Flash File Systems. *Existing file systems either introduce CoW-based index updating (i.e., LittleFS uses skip list to organize data) or suffer from slow lookup page scanning (i.e., SPIFFS allocates a lookup page for each block to organize data). NF2FS addresses these issues by relaxing management granularity to a coarse log/file granularity, thus enabling fast log (i.e., file/dir) scanning.*

point to them. Note that although F2FS [38] solves this problem by updating some indexes in place, this strategy relies on a hardware controller to support in-place updates. Moreover, the greedy WL layer allocates free blocks in a round-robin manner, which fails to balance wear among hot/cold data.

SPIFFS [16] overcomes the above issues, which introduces a flat file system (i.e., data are stored without hierarchy) and a best-effort WL layer, as shown in Figure 2(b). First, the file system layer allocates lookup pages (i.e., the first page of each block) to index block data without pointers. Therefore, it speeds up write by avoiding cascade pointer updates. Nevertheless, locating data (i.e., indexing) requires scanning lookup pages, which increases searching latency (e.g., create and random read) and reduces the real-time ability. Second, the best-effort WL layer also scatters blocks' P/E information in lookup pages. Specifically, it always allocates the least erased block at its best effort, and thus, the wear is perfectly balanced. However, the WL layer introduces significant overheads for block allocation as it must examine all blocks. Moreover, if the selected block is already allocated, SPIFFS should aggressively migrate valid data to another free block, which adds additional I/Os for each allocation.

As a result, none of the existing NOR flash file systems can satisfy the requirements in §2.2 simultaneously. Under the RAM-restricted scenario, they both lead to severe I/O amplification, either by adopting fine-grained block-based data organization or by scanning lookup pages to search data and decentralized per-block P/E information. This fine-grained management is efficient for RAM-sufficient scenarios; However, it squanders NOR flash's performance and the real-time guarantees under RAM-restricted MCU.

3 Observations and Motivation

As discussed in §2, existing NOR flash file systems suffer from heavy I/O overheads due to their fine-grained management

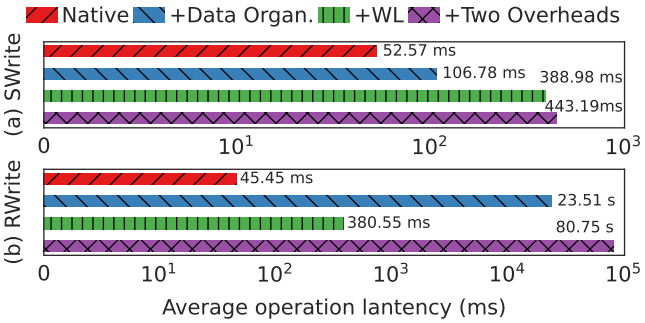


Figure 3. Overhead Breakdown in NOR Flash File Systems. *We emulate the aging storage layout to show how data organization and WL works. Specifically, SWrite sequentially writes a 2 MB file with 4 KB per I/O. RWrite performs 20 × 1 KB random updates. Native directly erases and writes data; +Data Organization adds an index-based file system; +WL adds a best-effort WL layer; +Two Overheads adds both.*

(i.e., block-level data organization and block-level WL). In this section, we first measure these overheads in §3.1. Then, we propose opportunities to mitigate them in §3.2.

3.1 Overheads of Existing NOR Flash File Systems

We study write overheads of data organization and WL using skip list and best-effort WL, which are already used in LittleFS and SPIFFS [16, 17] to support real-time reads and longer lifespans. Moreover, we do not test the lookup page-based indexing and greedy WL due to their poor searching performance (e.g., create and random read) and imbalanced wear. As the best-effort WL only works when there is no free space, we emulate an aging storage layout.

Fine-Grained Management Overheads within each Single Layer. As Figure 3 shows, index-based data organization increases operation latency by 1.03× and 517.27× (in SWrite/RWrite) compared to *Native*, which directly erases blocks and writes data. Therefore, using an index-based file system to organize data fails to exploit NOR flash write bandwidth and support OTA updates. The reason is that maintaining indexes in NOR flash introduces CoW I/Os, which causes severe I/O amplification, i.e., modifying a small index requires rewriting the whole block. Even worse, CoW can lead to the cascade effect because other indexes that point to the current index should also be updated.

On the other hand, best-effort WL increases operation latency by about 335 ms in SWrite (6.40×) and RWrite (7.37×) compared to *Native*. The reason is that it maintains decentralized WL information (i.e., P/E cycle stored in each lookup page) to mitigate index updating overheads. However, this WL strategy worsens write performance as allocating needs to scan lookup pages and measure erasures of all blocks.

Accumulated Overheads within the Stacked Two Layers. Unfortunately, when these two layers are stacked, the overheads are more than accumulated. As shown in Figure 3(a), compared to *Native*, *+Data Organation* and *+WL* increase operation latency by 54.21 ms and 336.41 ms, whose sum is just the growing latency of *+Two Overheads*. However, the overhead is more serious in RWrite. As shown in Figure 3(b), although *+Data Organation* introduces a huge latency gap compared to *+WL*, by stacking these two layers, the WL layer (i.e., *+WL*) further exacerbates the latency by 2.43× (i.e., *+Two Overheads*). The reason is that RWrite introduces numerous extra I/Os from CoW and its recursive index modifications, which also increases the WL layer's overheads with more allocations and more lookup page scanning.

Table 3. MCU Workloads using NOR Flash. *These workloads usually have small file sizes (i.e., B-MB) and deterministic I/Os (i.e., similar and simple-to-predictable I/O patterns).*

MCU Workloads	I/O Size	I/O Pattern
TinyML [35, 39]	Hundreds KB	Sequential read
Health monitor [40][a]	Tens B per entry	Sequential I/O
Event logging [33]	Tens B per entry	Sequential I/O
Configurations [27]	Several/Hundreds B	Sequential I/O
Busybox binaries [20]	4 KB block	Random read
OTA update [26]	4 KB block (chunk)	Rand/Seq Write[b]

[a]: The data format is 'x,y,z', each of which is a float value (§3.2 in [40]).
[b]: NOR flash prefers single binary rather than two copies [24] due to its small size, which is updated either by random writing or by rewriting.

3.2 Opportunities to Mitigate File System Overheads

Through the above analysis, we are motivated to mitigate data organization and WL overheads by relaxing management granularity to a coarse file granularity; thereby cooperating them within a file system layer and removing the separate WL layer. Our insight is based on two observations:

Relax Management to a File Granularity. As shown in Table 3, NOR flash applications/workloads for MCU are relatively simple and deterministic (but a general file system is required to offload development complexity). As a result, block wear in a file is usually balanced. For example, most larger (MB level) software binaries (e.g., Busybox [20]) are cold and rarely updated. In contrast, smaller logs (Byte–MB level) and configuration files (Byte level) are hot and frequently updated to perform real-time logging and configuration changes [7, 27, 33]. Therefore, it is possible to manage WL across files rather than blocks. On the other hand, these files/dirs are small, so flattening files and directories to decrease indexes is fairly efficient. Note that it is much more efficient than SPIFFS, which scans lookup pages to locate data without directory hierarchy. As a result, both data organization and WL can be relaxed to a coarse file granularity.

Cooperate within a File System. When both data organization and WL are managed in a coarse file granularity, the WL layer is naturally integrated into the file system, as shown in Figure 2(c). This is different from previous NOR

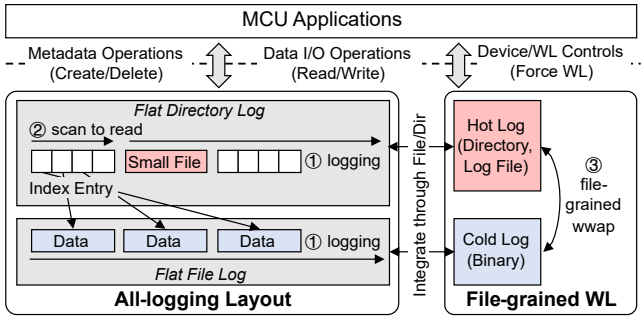


Figure 4. NF2FS Overview. *Files/Directories are represented by logs. Therefore, (1) block-level data organization is converted to the flat file/directory layout, and (2) file-grained WL is achieved by coarse file/directory (i.e., log) swapping.*

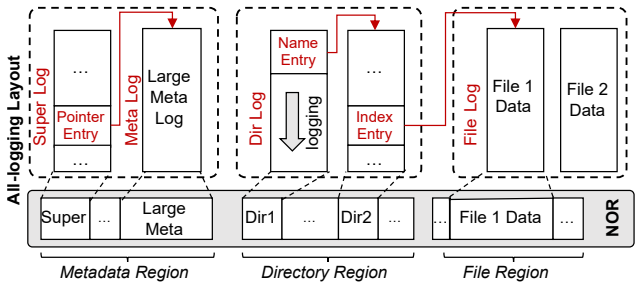


Figure 5. The NOR-friendly *All-logging Layout*. *Despite file and directory, NF2FS also builds system-wide metadata (e.g., super block) into a log to avoid unnecessary erasing.*

flash file systems that separate their functionalities into two layers, where each layer is *decoupled and replaceable*. Particularly, LittleFS can replace the greedy WL layer with the best-effort one to provide better WL at the cost of a significant performance drop. However, in our design, file-based WL is only reasonable/possible when the file system also manages NOR flash in a coarse file granularity.

In summary, we observe that the file characteristics (i.e., small for efficient scanning) and I/O patterns (i.e., simple and deterministic) of NOR flash applications can be leveraged to simplify and optimize existing NOR file systems.

4 NF2FS Design

Motivated by §3, we propose NF2FS, the NOR Flash-Friendly File System. NF2FS manages NOR flash in a coarse file granularity and minimizes I/O amplification (due to frequent index updates and NOR flash scanning), while combining file-grained WL to prolong NOR flash lifespan. This section first highlights the overview of NF2FS (§4.1) and then dives into details of the key technique, all-logging layout (§4.2). Under this layout, we further design efficient WL (§4.3) and garbage collection (§4.4) strategies. Finally, we show how to maintain crash consistency by ordering atomic I/Os (§4.5).

4.1 Overview

We illustrate the overview of NF2FS in Figure 4. The key to achieving coarse-grained data organization and WL is to abstract files and directories as logically contiguous areas (i.e., log [41, 42]). Specifically, ① in-NOR data are all organized in a log manner, where hot metadata (e.g., indexes) and small file data (e.g., configuration files) are embedded in the *directory log*, while large files (e.g., cold binaries and hot log files) occupy separate *file logs*. ② The data in the file log is indexed by an *index entry* (stored in the *directory log* and updated by appending a new *index entry*), and data can be retrieved by first scanning to locate the *index entry*, and then locating data through the *index entry*. ③ Consequently, the *directory log* updates more frequently than the binary, and NF2FS swaps the two logs to balance wear in a file/directory granularity.

4.2 All-Logging Layout

As shown in Figure 5, the all-logging layout considers entities in the file system as logically contiguous log areas, including files, directories, and system metadata (e.g., super block). These areas are cross-indexed using out-of-place updated pointers, thereby forming the file system hierarchy. We now present the design of four key log areas as below:

- *Super log* stores basic file system information in blocks 0 and 1. Unlike traditional super block, all data in the super log (e.g., magic number) are dynamic and appended as log entries. The super log is the only log area that does not move since it is used to locate other movable log areas.
- *Meta log* stores file system metadata, such as bitmap entry for space allocation. It generally has a fixed size, but the location is dynamic for WL. Each meta log only stores one type of system metadata. To locate multiple meta logs, the super log appends *pointer entries* to reference them.
- *Directory log* appends directory data, such as name entry. The directory log is stored as a linked list (for blocks) to avoid large contiguous space allocation [42], which is traverse-only to avoid pointer modifications. The root directory log is pointed to by the super log through the root *name entry*. Following the root directory log and name entries, the file system name hierarchy is constructed.
- *File log* appends file data and is referenced by extent-like *index entry* [30] (stored in its parent directory log). Different from the directory, a file is extended either by expanding the current file log (if physical space is contiguous) or by allocating a new file log, and file logs of one file are then reorganized by appending a new *index entry*.

As a result, accessing a file is done by scanning directory logs to locate the *index entry* and then retrieving data in corresponding file logs. On the other hand, WL is achieved by swapping hot/cold log areas and updating references (that point to swapped log areas) in their parent directory logs.

4.3 Adaptive Log Area Wear Leveling

In NF2FS, WL is used to swap hot and cold log areas for wear balance. We separate WL into two stages, namely *early stage* and *age stage*, to provide reasonable tradeoffs between lifespan and performance. The transition between the two stages is determined by NOR flash usage, which is optional and configurable, depending on different requirements (§5.3).

Specifically, during *early stage*, NF2FS uses a greedy strategy to allocate free blocks for new log areas similar to LittleFS [17]. This strategy has little overhead, which simply records the last allocated block and scans from it. After a long run, during *age stage*, NF2FS follows the best-effort strategy to select the least worn blocks for new log areas. Different from SPIFFS [16], NF2FS ensures the selected blocks are always unused to avoid data migration in critical I/O paths. For certain conditions (e.g., the MCU is idle), NF2FS aggressively swaps log areas and balances wears proactively.

The biggest challenge lies in how to quickly determine the least worn blocks and the log areas to be swapped in NOR flash since (1) timely maintaining centralized block P/E information in NOR flash leads to frequent in-place updates, and (2) maintaining decentralized P/E information (stored in each block) is feasible but requiring to scan all blocks for P/E comparisons. To address the problem, we combine the two strategies by maintaining decentralized P/E information in each block and periodically aggregating them (in file granularity) into a centralized sort array. Therefore, the sort array records coarse-grained file P/E information, which is efficient for wear measurement (and the sort array will be periodically reconstructed) and free from in-place updates.

As a result, WL-aware block selection can be done by checking the centralized sort array, which is efficient and does not introduce heavy NOR flash scanning overheads. In our implementation (§5), the sort array is stored as a log entry (i.e., *sort array entry*) in a separate *meta log*.

4.4 Garbage Collection

Through the runtime of the MCU system, the log entries keep accumulating. To avoid running out of NOR space, garbage collection (GC) is triggered to collect valid log entries (identified by *Type* in the entry header, §5.6) in the victim log, then copy and compact them in a new log area, and finally, the victim log is erased for further allocation [38]. The reason for the invalid log entries is mainly because updated entries logically overwrite/invalidate the old ones.

Under the all-logging layout, GC can be categorized into two types: (1) *victim log that has no references*. The super log is the only log area without references. Therefore, the GC process follows the normal procedure as described above. (2) *victim log that has references*. The remaining log areas have references: the meta log is pointed by the super log; the directory log is either pointed by the super log (the root directory) or the parent directory log; the file log is pointed by the directory log. For these log areas, when data migration is finished, GC further updates references by appending new pointers to their parent directory logs.

4.5 Crash Consistency

We follow *synchronous soft update* [34, 37] to guarantee consistency, which emphasizes two aspects: ordering and atomicity. NOR flash naturally preserves ordering thanks to the SPI communication protocol. However, it does not introduce any atomicity guarantees. Nevertheless, as a bit value contains only two states (i.e., 0/1), it can guarantee single-bit operation atomicity with unidirectional bit updating (§2.1). Note that journaling and checkpointing are two other crash consistency solutions, but the former is too expensive for NOR flash due to double writes, while the latter is mainly used for block-based file systems that rely on block layer for caching, which is not available for NOR flash.

To guarantee atomicity, we introduce NOR flash Atomic Operations to ensure the atomicity of I/O. We leverage NOR's single-bit atomicity to achieve lightweight transaction-style updates by (1) issuing a transaction header plus the to-be-protected data through SPI, and then (2) committing the transaction by atomically flipping a bit from 1 to 0 in the header to indicate the completion of the I/O (or transaction).

With atomic I/Os, we can carefully order I/Os to ensure crash consistency. The I/O order rules are derived from soft update [34]. Specifically, ① never points to a structure before it has been initialized; ② never reset the old pointer to a live resource before the new pointer has been set. Note that the rule "never re-use a resource before nullifying all previous pointers to it" is already satisfied with the all-logging layout by updating data in an out-of-place manner.

We run two examples to show how these rules work. First, for a simple file append operation, the new *index entry* is appended to the *directory log* only after the data is fully written to the file log (rule ①). Consequently, the crash during the operation results in an incomplete *index entry*, and thus NF2FS will never try to access the crashed data. Second, for a much more complex operation that involves data migration, such as file GC, NF2FS ensures that the old data is not invalidated until after the new data is fully migrated (rule ②). Nevertheless, the soft update technique is not enough to address data leak [31], which is important in NOR flash due to its limited space. For sure, it is possible to collect the leaked data using a background thread; however, MCUs are usually resource-constrained and many of them contain only one processor. We address these issues later in §5.6.

5 Implementation

We implement NF2FS in FreeRTOS with POSIX-compatible I/O interfaces, including ~9,000 lines of C codes. In this section, we centralize our implementation considerations and optimizations on how to make NF2FS efficient and practical.

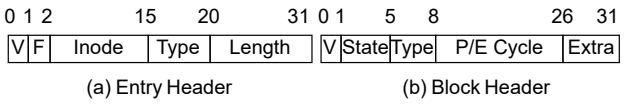


Figure 6. Header for Layout Management. *The 32-bit headers are used to facilitate layout management such as compact layout (§5.1) and WL (§5.3). One important function is to provide atomicity for write and allocation (§5.6). Specifically, V is the first written bit to begin a transaction, while F and State are rewritten when the transaction is finished.*

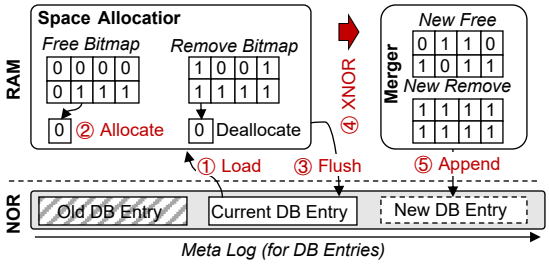


Figure 7. Space Allocation. *NF2FS designs dual bitmap (i.e., free bitmap and remove bitmap) for NOR-friendly space management, which can be reconstructed by merging the two bitmaps with XNOR operations.*

5.1 Region-based All-Logging Layout

For simplicity, NF2FS divides the whole NOR flash in a coarse-grained *region* granularity, where each region has a fixed size and contains only one type of *log area*. The design motivation and details are introduced as follows:

Why Region Instead of Log Area? Balancing wear across log areas is non-trivial as they have variable sizes, thereby requiring fine-grained metadata to record the offset and size of each log area. By dividing NOR flash into fixed-size regions, we can simplify wear leveling by swapping the fixed-size region instead of the variable-size log area. Currently, the region number is configurable according to the tradeoff between RAM usage and region swap granularity, and we choose 128 as the default number to achieve the finest swap granularity under MCU's RAM restriction (i.e., 1 KB RAM, §6.6). There are four region types: *metadata region* stores meta logs and is fixed as the first region (note that super log is fixed as block 0 and 1); *directory region* stores directory logs; *file region* stores file logs, and *reserve region* assists with region swap (i.e., WL), which is fixed as the last region.

Log Area Built on Region. Log areas are allocated within regions. As shown in Figure 5, the *meta log* is allocated from the *metadata region*, while the super log can be only allocated in block 0 or 1. Currently, *metadata region* and *reserve region* are not involved in WL to facilitate management. However, *metadata region* must perform GC to avoid running out of space. Similarly, *directory logs* and *file logs* are allocated from the *directory* and *file regions*, respectively.

Compact Log Layout. To fully exploit NOR flash space, we store log entries compactly within a log area (e.g., *file log* and *directory log*), which means the entry is aligned at the one-byte level. To achieve this, for variable size log entries, we embed 32-bit *entry headers* within them, as shown in Figure 6(a). The header records the length of the log entry (`Length`); therefore, NF2FS can locate the next log entry by simply adding the current address with `Length`. The header further achieves I/O atomicity, as later introduced in §5.6.

5.2 Log Area Allocation

NF2FS aims to allocate physically contiguous space for log areas. However, as directory and file data are dynamic, we only make *super log* and *meta log* be physically contiguous, while *directory log* and *file log* are logically contiguous.

Allocation Strategy. NF2FS reserves the first two blocks for *super log*, and other blocks in the *metadata region* are for *meta logs*. The *meta log* allocation size is set to 4 blocks, which is a tradeoff between WL frequency and WL effectiveness (Figure 14). Moreover, *file logs* and *directory logs* are dynamically extended. By default, we allocate 1 block at a time to extend the *directory log* (using the *extend entry*, §5.4). Moreover, we guarantee a file's physical contiguity with our best effort. Specifically, we extend current *file log* if there is contiguous space. If not, we create a new *file log* and update the *index entry* in its parent *directory log* (§5.4). If a file is fragmented with multiple *file logs*, NF2FS performs file GC to merge them (§5.5). Finally, space allocation is managed by the *dual bitmap entry*, as introduced below.

Dual Bitmap Entry. Bitmap is widely used in file systems for block allocation [30]. However, they can not be directly used in NOR flash. The key problem is NOR flash's unidirectional bit updating (§2.1), while bits in bitmap require bidirectional changes. Therefore, we present *dual bitmap*, which contains a *free bitmap* and a *remove bitmap*. Specifically, the bit value of 1 in the *free bitmap* indicates the block is free, while the bit value of 0 represents block allocation. The *remove bitmap* is used when related bit values in the *free bitmap* are 0, and the bit value of 0 suggests that the block is deallocated. To enable region-grained space management, we maintain *dual bitmap entry (DB entry)*, which is 16 bytes in size and can represent a total of 64 blocks in a region (for our tested NOR flash chip W25Q256 [19]). Note that *DB entries* do not have *entry headers* to align with 4 KB block size. Nevertheless, NF2FS is still consistent as *dual bitmap* will be reconstructed when failures occur (§5.6).

DB Entry Reconstruction. One *DB entry* can only represent block allocation and deallocation once, which means further allocation on the deallocated block is not allowed. To overcome this, NF2FS reconstructs a new *DB entry* by merging the *free bitmap* and the *remove bitmap* by XNOR. The insight is that a block is free only if its related bit value

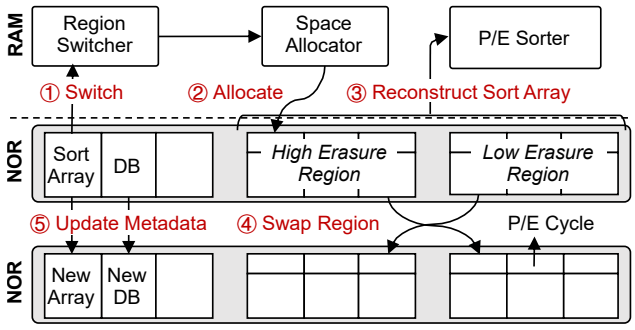


Figure 8. Wear Leveling. *The swap operation is similar to swapping two variables, requiring three-step region migration.*

in two bitmaps are the same: the same bit value of 1 indicates that the block is unused, while the same bit value of 0 indicates that the block has been allocated and deallocated. Therefore, a simple XNOR operation can generate a new *free bitmap*, and bit values in the *remove bitmap* are reset to 1.

Space Allocator. As shown in Figure 7, NF2FS deploys a *space allocator* for log allocation. For brevity, we omit the WL and introduce it later in §5.3. Here, the *space allocator* checks the log type to decide which region to allocate. Then, ① it loads a *DB entry* (that represents a region selected by WL) into a 16-byte buffer, ② the space allocator finds and allocates the first free block through *free bitmap* scanning, ③ when the region is fully allocated, the buffer is flushed back using in-place updates, ④ later, *DB entries* will be reconstructed based on WL strategies (see §5.3). ⑤ Finally, NF2FS appends a new *DB entry* into *meta log* and invalidates the old one.

It is a concern that a power failure might lead to the loss of allocation states. To address the issue, we leverage *block header* to guarantee allocation atomicity (§5.6).

5.3 Region-Based Wear Leveling

NF2FS aims to adopt region-based WL to provide the best tradeoff between lifespan and performance. WL achieves two goals: (1) select the least worn region for allocation, and (2) swap regions to balance hot/cold data. To diagnose wears, NF2FS assigns a 32-bit *block header* for each block to record the P/E cycle, as shown in Figure 6(b).

Early Stage. NF2FS simply allocates blocks in a round-robin manner because the P/E is not obvious. Moreover, it reconstructs all *DB entries* (stored continuously) after allocating the last region. When the number of *DB entry* reconstruction reaches 1% NOR lifespan (i.e., 1 K for W25Q256 [19]), NF2FS activates WL. The stage transition is mainly caused by changing allocation strategy from sequential allocation to using *long-term sorted P/E array*, as introduced below:

Long-Term Sorted P/E Array is introduced as a centralized structure to absorb decentralized P/E cycles stored in each *block header*. The array maintains and sorts the P/E cycles of each region and is stored in one *meta log* as a log entry. To build the array, NF2FS requires extra but temporal RAMs (i.e., 1 KB for 128 regions), and the RAM analysis is in Table 6.

Age Stage. During the age stage, NF2FS first scans *block headers* to construct the *sort array* (i.e., *long-term sorted P/E array*). Then ① *region switcher* switches to the least worn regions[2]. ② *Space allocator* allocates blocks from the switched region. ③ When regions are switched by pre-defined times (i.e., 1024, indicating 16-byte *DB entries* fill a 4-block *meta log*, see sensitive analysis in §6.3), NF2FS scans *block headers* and reconstructs a new sort array. ④ NF2FS swaps data among low-high erasure regions with the *reserve region*. Specifically, given regions A and B, NF2FS first copies data from A to the *reserve region*, then from B to A, and finally from the *reserve region* to B. ⑤ After swapping, NF2FS updates metadata (e.g., *sort array* and references) to finish WL.

As region swap is time-consuming, the age stage is optional and can be done when the device is idle (i.e., delaying ④, others are the same). Moreover, even without the *age stage*, our *all-logging layout* is compact enough to prolong NOR flash lifespans (see average P/E cycle in Figure 13).

Table 4. Log Entries in the Directory Log.

Entry Types	Description
Dname Entry	Stores the sub-directory name string and the pointer to the sub-directory log.
Fname Entry	Stores only the sub-file name string.
Data Entry	Stores inline data (i.e., small files), whose threshold is 64 B, similar to LittleFS [17].
Index Entry	Stores pointers to file logs of a large file.
Extend Entry	Stores a back-pointer to extend directory log, which is the first entry of directory blocks.
Space Entry	Stores the space that can be reclaimed in the directory log (for directory log GC, §5.5).

5.4 Directory and File Organizations

Directory Organization. *Directory logs* maintain several types of entries for file system semantics. Each entry contains a *entry header* to indicate which file/directory it belongs to (i.e., *Inode* in Figure 6(a)). They are listed in Table 4.

In NF2FS, directory operations (e.g., open) scan hierarchical log areas for path resolution, which suffers from read amplification. Therefore, we construct a *tree buffer* to cache name entries (similar to VFS dentry cache). To restrict RAM usage, the buffer size is set to 256 B by default. Moreover, the directory log also supports complex operations, e.g., rename can be achieved by simply appending a new name entry (in the parent *directory log*) and invalidating the old one, while delete can be achieved by invalidating all related log entries. We introduce their crash consistency details in §5.6.

File Organization. NF2FS stores a file's data in one or more file logs, which are organized by an *index entry* in the parent

[2]These hot regions' *DB entries* are stored in a separate *meta log*, while cold regions' are stored continuously in another *meta log* (see the *early stage*).

directory log. The *index entry* is a $(offset, size)$ array to locate data in file logs. ① To read data at ofs, NF2FS first scans the parent *directory log* to locate the *index entry* where $offset \leq ofs < offset + size$, and then reads data from the corresponding *file logs*. ② To write data (either sequential or random write), NF2FS writes the updated data into a file log (either by extending the existing log or by creating a new log, §5.2), appends a new *index entry* to reorder data, and finally, invalidates the old *index entry*. In NF2FS, we allocate a file buffer to store small file data or large file index (512 B by default), and therefore, the max size of the $(offset, size)$ array is restricted to 42 (i.e., $4\ B\ entry\ header + 42 \times 12\ B$ $(offset, size) = 508\ B$). When a file has multiple *file logs*, NF2FS performs *file GC* to merge some of them.

5.5 Garbage Collection

We introduce the GC process of four types of log areas in the all-logging layout as below:

- *Super log*. Since the super log can be located in either block 0 or block 1 (§4.2), once either is full, the GC process scans the super log and copies the valid entries to another block. Finally, the old block is erased for invalidation.
- *Meta log*. The meta log stores *DB entry* (§5.2), *sort array entry* (§5.3), and *journal entry* (§5.6). For simplicity, we separate these entries into four meta logs (*DB entries* have 2 *meta logs* for two-stage WL, §5.3). As a result, entry migration is unnecessary, as when any of the four meta logs are full, NF2FS simply allocates a new meta log to write new entries, and the old meta log can be safely erased.
- *Directory log*. NF2FS tries to reclaim block-size space whenever a directory log expansion happens. It maintains a *space entry* to record the space that can be reclaimed. If the space exceeds a block size, NF2FS scans the directory log, migrates valid entries to a newly allocated directory log, and finally invalidates the old directory log.
- *File log*. NF2FS tries to merge fragmented logs and reduce the index entry size if a file has multiple file logs (i.e., the $(offset, size)$ array size is larger than 20). Specifically, the GC process scans the $(offset, size)$ array, rewrites fragmented data into a new contiguous file log, regenerates a new index entry, and finally invalidates the outdated index entry and file logs. Note that NF2FS can cancel file GC if it finds that file data is not extremely fragmented.

5.6 Crash Consistency and Failure Recovery

In this subsection, we first introduce how to achieve atomic I/O in NOR flash. Then, we show how NF2FS leverages atomic I/O to provide lightweight crash consistency.

I/O Atomicity. NF2FS uses *entry headers* and *block headers* to protect the atomicity of log entry write/invalidation and block (de)allocation, as shown in Figure 6.

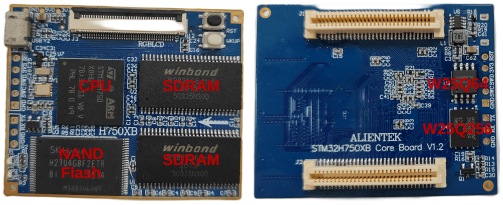
- *Log entry write atomicity.* To guarantee write atomicity, *entry header* is stored ahead of log entries with one-bit transaction flags *V* and *F*. Specifically, V is set to 0 to begin a transaction; after the entry is fully written, F is set to 0 to commit the transaction.
- *Log entry invalidation atomicity.* To guarantee invalidation atomicity, NF2FS performs bit-flip on the highest valid bit of *entry header*'s `Type`. For example, "0x1a" represents *Index Entry*, while "0x0a" represents its invalidation.
- *Block allocation atomicity.* To guarantee allocation atomicity, *block header* is stored ahead of each block with flags *V* and *State*. The bit values in *State* are arranged with the Hamming distance of 1, where "0x5", "0x3", "0x1", and "0x0" represent `gcing`, `allocating`, `finished`, and `old`. As a result, allocation atomicity is guaranteed by first setting *State* to `allocating` (or `gcing`), and to `finished` after certain operations (e.g., persisting data to the block).
- *Block deallocation atomicity.* NF2FS simply uses one bit-flip to set *block header*'s *State* from `finished` to `old`.

Crash Consistency. Overall, NF2FS provides crash consistency by carefully ordering atomic I/Os according to synchronous soft update rules (§4.5). However, the soft update cannot avoid data leaks. For example, a crash occurring during the writing of an index entry in an append operation does not impact consistency correctness, as NF2FS can verify entry integrity during access (by checking the entry header). However, such a crash can lead to data leaks because the written data may remain unreferenced and cannot be released. NF2FS overcomes the problem by checking the validity of each data block during recovery via block header scanning.

The key to verifying the validity of data blocks is to check whether the block is referenced correctly. There are two types of operations to be handled. First, for operations that contain block allocation without deallocation (e.g., append), NF2FS incorporates a backward pointer behind the block header, which points to the entry that references the newly allocated block (e.g., index entry). Consequently, NF2FS regards the block as valid only if the block header is valid and the related entry (via backward pointer) is also valid.

Second, for operations with block deallocation (e.g., GC, WL, and delete), simply using a backward pointer cannot work. For example, if the crash happens right before the invalidation of a block, the should-be-invalidated data block is regarded as a valid one during recovery, thereby causing a data leak. In this case, we introduce *journal entry* and append it to the meta log. The *journal entry* records the updated pointers and the current migration step. During recovery, NF2FS can redo the invalidation operations according to the *journal entry*. Note that the journal entry can also be used to avoid data leaks in the first case; however, journaling introduces the double-write overheads, and NF2FS aims to minimize these overheads to maximize NOR flash lifespan.

Note that for operations without any block operations (e.g., create a single *fname entry*), NF2FS lazily checks consistency by verifying entry header flags during the first access.



(a) Front-Side Resources (b) Back-Side Resources

Figure 9. Emulated Hardware Platform for MCU with NOR Flash. *It consists of a 32-bit Cortex-M7 core, a 512 MB NAND flash, 2 × 32 MB SDRAM, and two NOR flash chips (8 MB W25Q64 and 32 MB W25Q256). We select W25Q256 as the target NOR flash, restrict board RAM to 32 KB, and leverage NAND flash for storing the data to be transferred to the NOR.*

Failure Recovery. We summarize the overall crash recovery routine. To detect crash, we append a *magic number entry* to the super log to represent a normal unmount. Therefore, crashes can be detected if the entry is invalid. During failure recovery, NF2FS (1) first redoes operations according to *journal entry*, (2) and then scans *block headers* of each block to collect leaked data using backward pointer check, (3) and finally, NF2FS reconstructs a new *dual bitmap*. Afterward, NF2FS is ready to serve the next I/O requests. Note that as operations recorded by *journal entry* (e.g., GC and WL) are infrequent, the recovery time is often fast (§6.6).

Correctness Discussion. NF2FS follows *synchronous soft update* [34, 37] to guarantee the correctness of I/O ordering for crash consistency (in §4.5). Our major adaptation on NOR flash is using headers with carefully managed bit values to guarantee I/O atomicity, and leveraging backward pointer and journal entry to avoid data leaks. Consequently, our approach is compatible with the idea of soft update and ensures correct crash consistency.

6 Evaluation

Our experiments aim to answer the following questions:

- Does NF2FS outperform existing NOR flash FSes? (§6.2)
- Does NF2FS prolong NOR flash lifespans? (§6.3)
- Does NF2FS mitigate I/O amplification in writing? (§6.4)
- How data organization, WL, and GC affect NF2FS? (§6.5)
- What are RAM and recovery overheads of NF2FS? (§6.6)

6.1 Experimental Setup

Testbed. We select Positive Atomic STM32H750 Polaris development board [23] as the emulated MCU hardware platform, as shown in Figure 9. RAM usage is restricted to 32 KB during the evaluation. Tested file systems run on the real-time OS FreeRTOS 2.1 [25] to manage a 32 MB W25Q256 [19] SPI NOR flash, which is widely used in embedded systems.

NOR Flash-Based Virtual File System (NFVFS). We implement NFVFS in FreeRTOS to evaluate the compared NOR flash file systems with unified POSIX interfaces. Unlike traditional VFS in Linux [11], NFVFS offloads complex operations, such as path resolution, to underlying file systems.

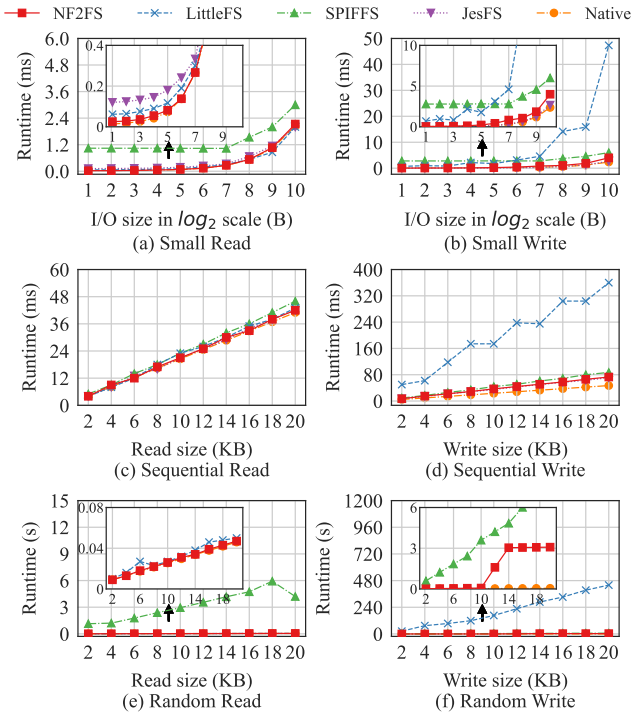


Figure 10. I/O Test. *Subfigures (a) and (b) perform small I/O with 2-1024 B sizes. (c) and (d) perform sequential I/O in a 1 KB I/O unit. (e) and (f) perform random I/O (on a 2 MB file) in a 1 KB I/O unit, which is unsupportable by JesFS.*

Compared NOR Flash File Systems. We compare NF2FS with LittleFS [17], SPIFFS [16], and JesFS [10]. We also compare with *native file system* that directly manages NOR flash without a file system and a WL layer. Note that we exclude JesFS in some tests as it only supports sequential I/Os.

NORbench. Evaluating NOR flash file systems is challenging without a public standard benchmarking tool. Therefore, according to our analysis of NOR flash workloads in Table 3, we present NORbench, a framework intended to test their effectiveness. To the best of our knowledge, NORbench is the first framework for benchmarking NOR flash file systems.

6.2 Performance Evaluation

I/O Performance. As shown in Figure 10, compared to LittleFS and SPIFFS, NF2FS has better performance in all I/O patterns. Moreover, its runtime is close to *Native*, which is raw I/O without a file system and a WL layer. Note that NF2FS's runtime has a sudden increase in random write, where GC is triggered twice (when the write size is 12 and 14 KB). LittleFS is poor in writing with complex indexes. SPIFFS is poor in small I/Os and random read for its page-level I/O granularity (i.e., 256 B) and lookup page-based indexing.

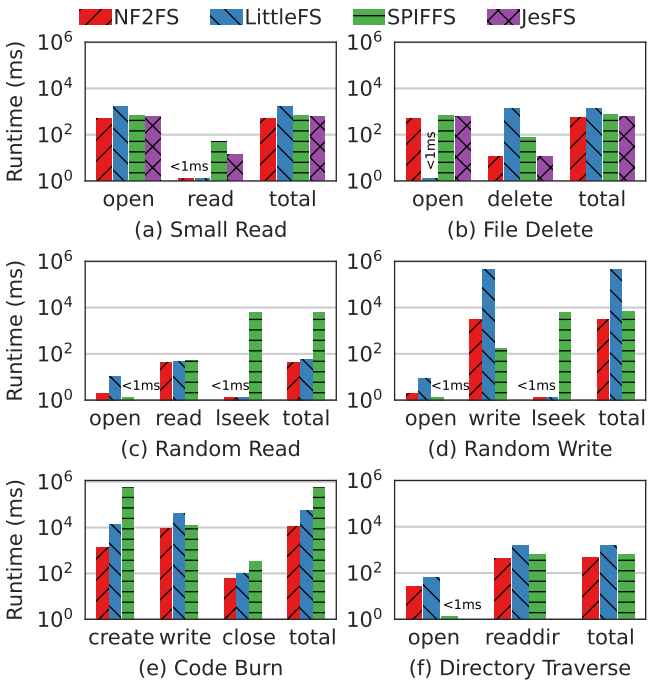


Figure 11. Operation Test. *Subfigures (a) and (b) read/delete 100 × 32 B files, subfigures (c) and (d) perform 20 × 1 KB random read/write on a 2 MB file, subfigures (e) and (f) burn and traverse the Busybox [20]. JesFS is only tested in (a) and (b) as it does not support random I/O and directories.*

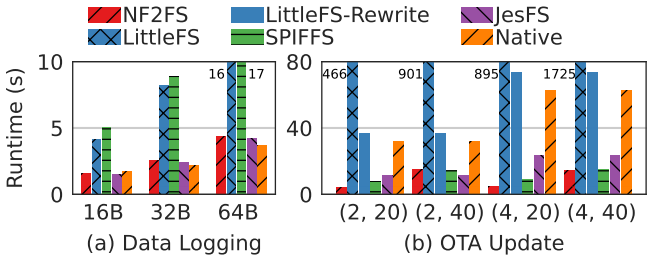


Figure 12. Real-world Workloads. *Sub-figure (a) logs 16 K entries of 16–64 B sizes. Sub-figure (b) follows SWupdate [26] to download a patch (20/40 updated blocks) and update a binary (2/4 MB) accordingly. In OTA update, NF2FS, LittleFS, and SPIFFS update the binary using random writes, while LittleFS-Rewrite, JesFS, and Native (which erases blocks immediately) rewrite a new binary and delete the old one.*

Other File/Dir Operations. We further generate six workloads to test other file/dir operations, where NF2FS also has the least runtime (Figure 11). Note that LittleFS has little open time in subfigure (b) for its path-based file deletion. SPIFFS has little open time in subfigures (c), (d), and (f) due to its flat architecture (i.e., SPIFFS only scans several lookup pages to open a file if NOR flash only stores limited data).

Real-world Workloads. Figure 12 studies NF2FS in two representative real-world scenarios. The results suggest that NF2FS's logging performance is comparable to direct write (i.e., *Native*), but LittleFS and SPIFFS suffer from heavy index updating and small I/O overheads, respectively. In *OTA Update*, NF2FS outperforms compared file systems thanks to its lightweight random write. However, LittleFS suffers from high index updating overheads (both random write and rewrite), SPIFFS will degrade read performance for its flat architecture (i.e. lookup page-based indexing), JesFS should reserve extra space (despite NOR flash's small size) to rewrite a new binary, whose overhead increases as the binary size grows, and *Native* takes a long time to erase the old binary. Otherwise, its performance is similar to JesFS.

6.3 Lifespan Evaluation

Test Setup. We generate two workloads to measure whether NF2FS can prolong NOR flash lifespans. Specifically, we first pre-occupy the 32 MB NOR flash with 16 MB cold data (*binaries*). Then, (1) we create/delete 1 million log files, and each of them logs 16 K × 32 byte entries (*logging*). (2) we create/delete 16 billion 32-byte small files (*small files*). As the total data written is equal, we test whether *all-logging layout* is compact among varying I/O sizes. Moreover, NF2FS's WL activation threshold is set to reconstruct *DB entries* 1 K times (i.e., 1% lifespan of W25Q256 [19]). The sort array reconstruction threshold is when *DB entries* fill a *meta log* (i.e., 4 blocks). We also evaluate SPIFFS with 64 byte fine-grained page (*SPIFFS-FG*) to show the benefits of the compact layout.

Lifespan Analysis. Results are shown in Figure 13, where NF2FS has lower P/E cycles and balanced WL thanks to its compact log layout (§5.1) and region-based WL (§5.3). However, LittleFS and JesFS suffer from wear imbalances, while SPIFFS has severe write amplification for its page-level I/O granularity (256 B). The write amplification problem is more serious for small files (e.g., configuration files), where LittleFS, SPJIFFS, SPIFFS-FG, and JesFS have 3.16×, 19.72×, 2.53×, and 75.89× more P/E cycles compared to NF2FS. Note that although *SPIFFS-FG* mitigates write amplification in *small files* (compared to SPIFFS), it increases large file metadata overheads and results in more P/E cycles in subfigure (d). JesFS has the most imbalanced wear in *small files* for its limited files (only 1000 files), block-level (4 KB) management granularity, and in-place updates. Moreover, blocks 0 and 1 in NF2FS (*super log*) have higher erasures in *Small Files*, which can be solved by simply extending *super log* size.

WL Sensitive Analysis. The sort array reconstruction threshold is when *DB entries* fill a *meta log*, which set to 4 blocks. Normally, the larger the *meta log* is, the more infrequent the WL is. We use the same tests to perform sensitive analysis, as shown in Figure 14. Results suggest that 4 blocks (NF2FS-4) is an optimal tradeoff between WL frequency and wear evenness. In contrast, NF2FS-1 wears out the reserve region (i.e.,

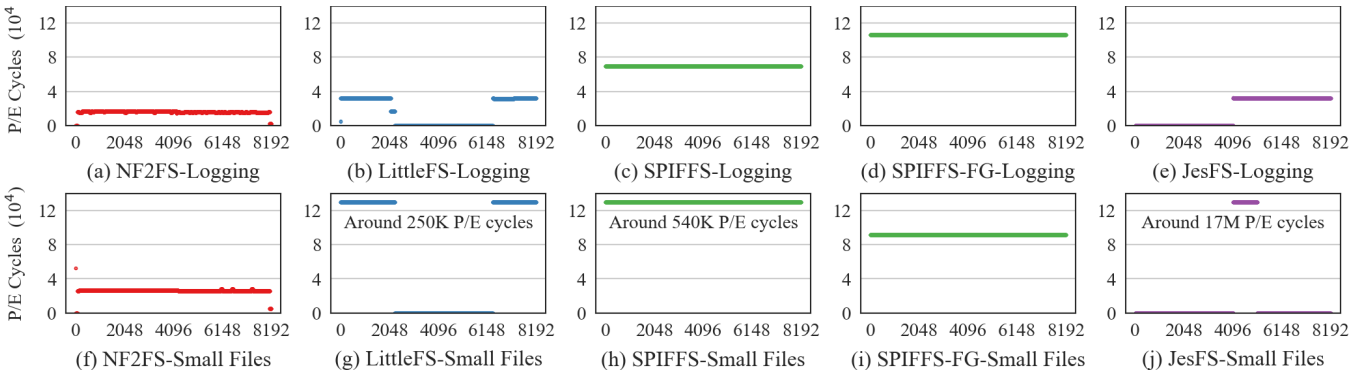


Figure 13. Lifespan Test. *X-axis indicates block number, while Y-axis indicates P/E cycles. Results suggest that, compared to existing NOR flash file systems, NF2FS has longer lifespans with lower P/E cycles and balanced WL among varying I/O sizes.*

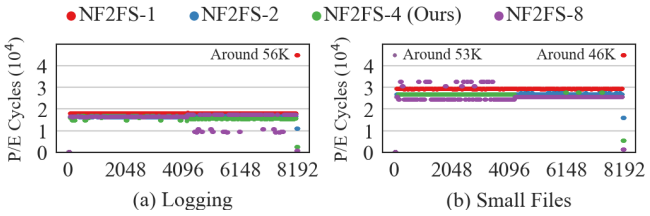


Figure 14. WL Sensitive Analysis. *The four colors represent the meta log size of 1, 2, 4, and 8 blocks, respectively. The smaller the number is, the more frequent the sort array reconstruction is (i.e., region swap). Results suggest that NF2FS-4 is an optimal tradeoff between WL frequency and wear evenness.*

the last region that is used for region swap, §5.1), NF2FS-2 performs WL more frequently, and NF2FS-8 wears unevenly.

Moreover, our age stage threshold is set as 1% NOR flash lifespan (i.e., 1 K DB entry reconstruction for 100 K P/E cycle W25Q256 [19]), which is configurable and can be set more aggressively according to users' requirements. For example, in our test, even setting the threshold to 50% lifespan can achieve acceptable WL before NOR flash wears out.

6.4 I/O Amplification

In this subsection, we measure the I/O amplification of write-related operations. Specifically, (1) we create 100 new files (i.e., *Create*). (2) We sequentially write a 2 MB file with 4 KB per I/O (i.e., *SWrite*). (3) We perform 20 × 1 KB random updates on a 2 MB file (i.e., *RWrite*). We collect the runtime, erase number, and read/write size, and calculate the amplification factor relative to the lowset I/O size (i.e., NF2FS).

Table 5 shows that NF2FS reduces up to 99.90% data reads and 97.20% data writes thanks to the compact and efficient all-logging layout. In contrast, LittleFS suffers from heavy read amplification in *SWrite* and write amplification in *RWrite* due to the CoW-based index updating. It also triggers multiple erasures even though there is enough space. SPIFFS suffers from heavy read amplification in *Create*. The reason is that

Table 5. I/O Amplification for Write-related Operations. *Results suggest NF2FS successfully mitigates I/O amplification and has a lower runtime for write-related operations.*

	File System	Time (s)	Era. (#)	Read/Write Size (MB)	Amplification Factor
Create	NF2FS	0.1	0	0.048/ 0.001	1.00×/ 1.00×
	LittleFS	0.5	0	0.191/ 0.003	3.98×/ 3.00×
	SPIFFS	114.7	0	50.473/ 0.005	1051.52×/ 5.00×
	JesFS	0.6	0	0.246/ 0.006	5.13×/ 6.00×
SWrite	NF2FS	7.4	0	0.002/ 2.006	1.00×/ 1.00×
	LittleFS	32.6	513	2.006/ 2.003	1003.00×/ 0.99×
	SPIFFS	9.7	0	0.299/ 2.341	149.50×/ 1.17×
	JesFS	7.4	0	0.002/ 2.006	1.00×/ 1.00×
RWrite	NF2FS	3.9	0	0.632/ 0.660	1.00×/ 1.00×
	LittleFS	469.3	6024	47.174/23.533	74.64×/35.66×
	SPIFFS	6.1	0	2.705/ 0.030	4.28×/ 0.05×

it uses lookup pages to index block data, which requires scanning all lookup pages to confirm that the created file is non-existent. Moreover, SPIFFS has a much smaller write size in *RWrite* because it does not trigger GC (while NF2FS and LittleFS trigger). Note that SPIFFS only triggers GC when there is no free space, which takes a long time as the best-effort GC requires scanning lookup pages to measure block erasures. JesFS performs well in *Create* and *SWrite* at the cost of losing many useful features (e.g., random I/O).

6.5 Aging Performance

We further study how data organization, WL, and GC affect performance when aging (i.e., erase before allocate, §3.1).

Data Organization and WL Overheads Breakdown. We study how the all-logging layout data organization and WL affect performance when aging, as shown in Figure 15. Compared to Figure 3, the all-logging layout has little overheads on *SWrite* and significantly reduces *RWrite* latency. Moreover, the region-based WL has little impact on performance.

GC Overheads. Figure 15 also implies that GC overheads can be mitigated. Specifically, *RWrite* triggers GC twice and

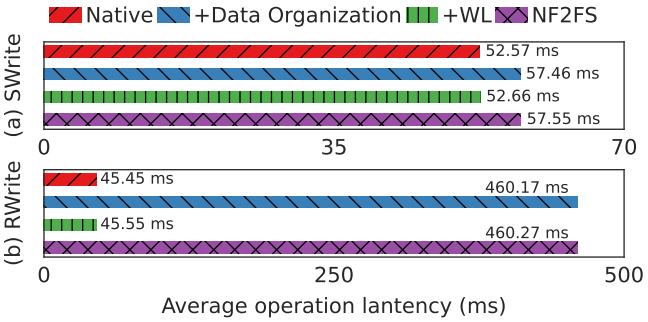


Figure 15. Aging Performance Breakdown of NF2FS. *Compared to Figure 3, our all-logging layout (i.e., data organization) significantly decreases write latency, while the region-based WL (for allocation, no data swap) has low overheads. Note that data swap is infrequent and measured in §6.6.*

has a much smaller operation latency compared to *RWrite* in Figure 3 (i.e., 23.51 s for LittleFS's index-based data organization). Moreover, SPIFFS takes ~1.2 s to GC 4 blocks (NF2FS GC 124 blocks with ~8.3 s), while JesFS does not have GC.

6.6 RAM Usage and Recovery

RAM Usage. We also measure peak RAM usage in our tests, as shown in Table 6. As NF2FS requires extra RAMs for WL (i.e., 1 KB to construct a sort array with 128 regions), we restrict normal RAM usage to less than LittleFS and SPIFFS. Moreover, JesFS is simple and almost uses no RAM.

Recovery. As shown in Table 6, NF2FS has the least format and mount time. We then randomly crash the machine during NORbench workloads. After remount, NF2FS can recover to a consistent state. Normally, the failure recovery time is 159 ms (to scan *block header* and reconstruct *dual bitmap*). However, recovery takes more time if it redoes the operation recorded by *journal entry* (always redo 1 operation because the valid *journal entry* is up to 1). Specifically, recovering *file log* GC (which happens during OTA updates) takes ~1.5 s; WL is infrequent and optional, which can be triggered when idle and takes up to 45.33 s for recovery; Super/Meta/Dir log GC takes ~100 ms, while other operations (e.g., delete) takes less than 1 ms. Note that SPIFFS takes 584 ms for recovery but runs *SPIFFS_check* to guarantee consistency, which consumes more than 1 hour in our test.

7 Discussion and Related Works

NOR Flash with Raw NAND File System. Several works, such as JFFS2 [41] and HNFFS [43], follow raw NAND file systems (i.e., Figure 1(b)) to build their services, which suffer from poor scalability. Specifically, JFFS2 is designed for extremely small NOR flash (i.e., 2 MB). It uses a complex RAM RB-tree for fast I/O, making the RAM consumption grow linearly with increasing storage capacity. HNFFS attempts to mitigate the problem by merging index nodes. However, it fails to address the linear RAM usage problem. Moreover, these two file systems should scan NOR flash during system startup, which is extremely slow. In contrast, our approach has low RAM overhead and fast system startup.

Table 6. Peak RAM Usage and Recovery Time.

File System	RAM Usage (B)	Recovery Time (ms)		
		Format	Mount	Failure Rec.
NF2FS	2216 + WL[a]	<4	<3	159 + Redo[b]
LittleFS	3712	118	<3	<13
SPIFFS	3536	573	573	584 + Check[c]
JesFS	<100	54389	<3	285

[a]: WL requires 1 KB temporal RAMs to sort regions.
[b]: Redo GC, WL, and delete if they were running during crashes.
[c]: The check time is more than 1 hour.

NOR Flash with Dedicated File System. JesFS [10] and EFS [2] are two other NOR flash file systems targeting dedicated scenarios (e.g., storing binaries). Therefore, they support extremely limited operations (e.g., sequential I/O, no-WL), and thus do not suit general-purpose MCU workloads. Moreover, TFFS [33] is a transactional file system for microcontrollers, but it also fails to maintain name hierarchies (i.e., directory), file truncation, and rename, thereby squandering powerful file system semantics.

NOR Flash without a File System. Several embedded systems, such as automotive and industrial systems [1, 12], treat NOR flash as write-once-read-more firmware to store binaries, making it possible to directly control NOR flash. However, currently, executing more complex operations (e.g., OTA updates and data logging [14, 18]) with high performance and longer lifespans becomes important. Although developers can embed the logic into binaries, it will significantly complicate system design and impede scalability.

NOR Flash vs. Embedded Persistent Memory (PM). Recently, embedded PM has been promised to replace NOR flash for its better performance, erase-free nature, and larger capacity. However, the PM market is rather than mature [8, 29]. Moreover, current PM research primarily focuses on large-capacity Intel Optane PM, and existing PM file systems leverage numerous RAMs to accelerate indexing [36, 42], which is unsuitable for small IoT devices. Although PMFS [32] introduces an in-PM file system, it cannot be directly applied to NOR flash due to its numerous in-place pointer updates.

8 Conclusion

This paper proposes NF2FS, which relaxes data organization and WL to a coarse file granularity. By designing an all-logging layout to manage files and directories in a log manner, we show how to efficiently incorporate NF2FS within NOR Flash. We implement NF2FS in FreeRTOS. Experimental results on a real embedded system show that NF2FS can significantly improve I/O performance and prevent quick NOR flash wear-out under RAM-restricted MCU environments.

References

[1] NOR | NAND Flash Guide: Selecting a flash memory solution for embedded applications. https://my.micron.com/content/dam/micron/global/public/products/product-flyer/nor-nand-flash-guide.pdf.

[2] Embedded File System. https://www.keil.com/pack/doc/mw/FileSystem/html/emb_fs.html, 2004.

[3] UBIFS - UBI File-System. http://www.linux-mtd.infradead.org/doc/ubifs.html, 2010.

[4] MSP430F677x, MSP430F676x, MSP430F674x Polyphase Metering SoCs. https://www.ti.com/lit/ds/symlink/msp430f6777.pdf, 2018.

[5] Use Flashless Microcontrollers to Lower System Costs and Increase Performance. https://www.digikey.com/en/articles/use-flashless-microcontrollers-to-lower-system-costs-and-increase-performance, 2019.

[6] NOR vs NAND: So You Think You Know the Music? https://www.jblopen.com/nor-vs-nand-so-you-think-you-know-the-music/, 2020.

[7] An introduction to IoT logging types and practices. https://www.techtarget.com/iotagenda/tip/An-introduction-to-IoT-logging-types-and-practices, 2022.

[8] Emerging Non-Volatile Memory A 2022 Market Update. https://conferenceconcepts.app.box.com/s/rozqo2kogzxmvql52iw6wxxxglepbkmc, 2022.

[9] IC Insights: NOR Flash market is expected to grow by 21% in 2022. https://www.semimedia.cc/?p=12622, 2022.

[10] Jo's embedded serial file system. https://github.com/joembedded/JesFs, 2022.

[11] Linux kernel source tree. https://github.com/torvalds/linux/tree/master/fs/jffs2, 2022.

[12] Micron Automotive Portfolio Featured in Li Auto L9 and Desay SV IPU04. https://www.micron.com/about/blog/2022/august/micron-automotive-portfolio-featured-in-li-auto-l9-and-desay-sv-ipu04, 2022.

[13] NAND Flash Memory. https://pdf1.alldatasheet.com/datasheet-pdf/view/513742/MICRON/MT29F2G01ABAGDSF.html, 2022.

[14] NOR Flash for wearables and hearables - Use Cases. https://conferenceconcepts.app.box.com/s/74528axuvazf11pnagm0bcdb8bbe05ov, 2022.

[15] Q-FLASH. https://pdf1.alldatasheet.com/datasheet-pdf/view/75850/MICRON/MT28F128J3.html, 2022.

[16] SPI Flash File System. https://github.com/pellepl/spiffs, 2022.

[17] The design of littlefs. https://github.com/littlefs-project/littlefs, 2022.

[18] Ultra-low Power NVM System Design Strategies for IoT and Wearable Applications. https://conferenceconcepts.app.box.com/s/47frkgyqkmolw9542j9e61jm0sofyll4, 2022.

[19] W25Q256 Datasheet, PDF. https://www.alldatasheet.com/view.jsp?Searchword=W25Q256, 2022.

[20] BUSYBOX. https://www.busybox.net/, 2023.

[21] Flash translation layer (FTL) and mapping. https://en.wikipedia.org/wiki/Flash_memory_controller#Flash_translation_layer_(FTL)_and_mapping, 2023.

[22] Universal Flash Storage. https://en.wikipedia.org/wiki/Universal_Flash_Storage, 2023.

[23] (1 Pcs)ALIENTEK Polaris STM32H750XBH6 Development Board H750/F750 Core Board. https://www.ebay.de/itm/204944016747?srsltid=AfmBOorsXFfyPDlEVgb0q9PVpbVnIB353OacZcUlndNbT_V3_DpLUIaF, 2024. Accessed on 13.11.2024.

[24] Delta Update with SWUpdate. https://github.com/sbabic/swupdate/blob/master/doc/source/delta-update.rst, 2024.

[25] FreeRTOS™ Real-time operating system for microcontrollers. https://freertos.org/, 2024.

[26] Single copy - running as standalone image. https://sbabic.github.io/swupdate/overview.html#single-copy, 2024.

[27] The UCI system. https://openwrt.org/docs/guide-user/base-system/uci, 2024.

[28] Tony Benavides, Justin Treon, Jared Hulbert, and Weide Chang. The Enabling of an Execute-In-Place Architecture to Reduce the Embedded System Memory Footprint and Boot Time. *Journal of Computers*, 3:79–89, 2008.

[29] Alexander Buck, Karthik Ganesan, and Natalie Enright Jerger. FlipBit: Approximate Flash Memory for IoT Devices. In *Proceedings of the 2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 876–890, 2024.

[30] Mingming Cao, Suparna Bhattacharya, and Ted Ts'o. Ext4: The Next Generation of Ext2/3 Filesystem. In *Proceedings of the 2007 Linux Storage & Filesystem Workshop (LSF)*, San Jose, CA, 2007.

[31] Zhuan Chen and Kai Shen. OrderMergeDedup: Efficient, Failure-Consistent Deduplication on Flash. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, pages 291–299, 2016.

[32] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems (Eurosys)*, New York, NY, USA, 2014.

[33] Eran Gal and Sivan Toledo. A transactional flash file system for microcontrollers. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATC)*, page 7, USA, 2005.

[34] Gregory R Ganger, Marshall Kirk McKusick, Craig AN Soules, and Yale N Patt. Soft updates: a solution to the metadata update problem in file systems. *ACM Transactions on Computer Systems (TOCS)*, 18(2):127–153, 2000.

[35] Massimo Giordano, Rohan Doshi, Qianyun Lu, and Boris Murmann. TinyForge: A Design Space Exploration to Advance Energy and Silicon Area Trade-offs in tinyML Compute Architectures with Custom Latch Arrays. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 1033–1047, New York, NY, USA, 2024.

[36] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, page 494–508, New York, NY, USA, 2019.

[37] Hayley LeBlanc, Nathan Taylor, James Bornholt, and Vijay Chidambaram. SquirrelFS: using the rust compiler to check file-system crash consistency. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 387–404, Santa Clara, CA, July 2024. USENIX Association.

[38] Changman Lee, Dongho Sim, Joo Young Hwang, and Sangyeun Cho. F2fs: A new file system for flash storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, volume 15, pages 273–286, 2015.

[39] Nuntipat Narkthong, Shijin Duan, Shaolei Ren, and Xiaolin Xu. MicroVSA: An Ultra-Lightweight Vector Symbolic Architecture-based Classifier Library for Always-On Inference on Tiny Microcontrollers. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 730–745, New York, NY, USA, 2024.

[40] Jie Ren, Fuyu Guan, Ming Pang, and Shuangling Li. Monitoring of human body running training with wireless sensor based wearable devices. *Computer Communications*, 157:343–350, 2020.

[41] David Woodhouse. JFFS: The journalling flash file system. In *Ottawa linux symposium*, 2001.

[42] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, pages 323–338, Santa Clara, CA, 2016.

[43] Yanqi, Pan and Zhisheng, Hu and Nan, Zhang and Hao, Hu and Wen, Xiaand Zhongming, Jiang and Liang, Shi and Shiyi, Li. HNFFS: Revisiting the NOR Flash File System. In *Proceedings of the 2022 IEEE 11th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 14–19, 2022.