Short Paper: Simulation-Based Performance Characterization of Common NOR Flash File Systems

Yannick Loeck
yannick.loeck@tuhh.de
Hamburg University of Technology
Hamburg, Germany

Christian Dietrich
christian.dietrich@tu-braunschweig.de
Technische Universität Braunschweig
Braunschweig, Germany

Ulf Kulau
ulf.kulau@tuhh.de
Hamburg University of Technology
Hamburg, Germany

Abstract

NOR flash memory is widely used as storage in embedded systems as it offers dense non-volatile memory up to the gigabit range while being more reliable than NAND flash. To avoid the cost of a standalone flash controller in such small systems, specialized flash file systems are used to handle the limited write endurance and asymmetric program/erase characteristics. While many NOR file systems exist and are extensively deployed, there is a lack of systematic comparisons of their performance.

In this paper, we present VFLASHSIM, a simulator for capturing interactions between file systems and flash memory and for running representative benchmarks. Unlike evaluations on physical hardware, our simulation-based approach avoids prohibitively long erase latencies and cell wear. We implement a NOR flash model and, using VFLASHSIM, evaluate four representative file systems, SPIFFS, LittleFS, YAFFS2, and NF2FS, in terms of write amplification, garbage-collection behavior, and RAM utilization. Our analysis shows how design choices and implementation bugs, such as excessive device scanning, unconditional sector erases, and a lack of garbage collection in allocation paths, cause substantial performance issues. Additionally, we identify how to provoke failure conditions in different file systems, which can result in unrecoverable crashes.

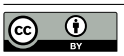
Among the evaluated file systems, YAFFS demonstrates the best overall performance and reliability at the cost of high RAM usage, whereas a modified LittleFS remains the best option for more memory-constrained systems. Since performance trade-offs vary by workload and hardware constraints, no single file system is universally optimal. With VFLASHSIM, we provide a tool that enables developers and users of flash file systems to analyze their behavior under diverse configurations and workloads.

CCS Concepts

• **Computing methodologies → Modeling and simulation**.

Keywords

NOR flash, flash file system, simulation



SenSys '26, Saint Malo, France

ACM ISBN 979-8-4007-2309-4/2026/05
https://doi.org/10.1145/3774906.3802793

ACM Reference Format:
Yannick Loeck, Christian Dietrich, and Ulf Kulau. 2026. Short Paper: Simulation-Based Performance Characterization of Common NOR Flash File Systems. In *ACM/IEEE International Conference on Embedded Artificial Intelligence and Sensing Systems (SenSys '26), May 11–14, 2026, Saint Malo, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3774906.3802793

1 Introduction

Modern embedded systems and smart IoT devices require *non-volatile memory (NVM)* for firmware and configuration storage, but also for increasingly large volumes of generated sensor and application data [5]. Flash is a family of fast and dense *non-volatile memory (NVM)* technologies with a lower cost per bit compared to alternative NVMs like FRAM or MRAM [6]. It is organized as arrays of flash cells and has become ubiquitous over a broad spectrum of embedded, consumer, and industrial electronics. While NAND flash is optimized for high density and large data storage sizes, it comes with reliability issues due to read/write disturbances, usage of *multi-level cells (MLCs)*, and it often already ships with bad blocks [18].

For systems that collect data in harsh environments, such as aerospace and remote sensing, where storage reliability is crucial, NOR flash is an attractive alternative to NAND. While NOR flash is typically smaller (see Figure 1), it does not require *error correcting codes (ECCs)* [14], and allows for byte-granular access. While commonly used for firmware storage or small data logging, NOR devices in the gigabit range are now commercially available and extend its applicability to larger-scale data storage – especially for dependable systems like NewSpace applications. For example, LituanicaSAT-2 [20] stores its mission data on NOR flash and the Greek ERMIS CubeSats [25] use NOR flash in both the *on-board computer (OBC)* and the commercial *payload data processing unit (PDPU)*.

Although more reliable, NOR flash still suffers from the common flash limitations: a mismatch between erase and write granularities and limited write endurance (about 100,000 cycles per cell). While additional hardware can mitigate these in larger storage-class systems (e.g., SSDs), the cost of an additional flash controller in terms of money, energy, heat, and area is often prohibitive for resource-constrained systems. Hence, NOR-based systems usually access the raw flash device directly and implement the *flash translation layer (FTL)* in software as part of the *flash file system (FFS)*.

Over the years, many specialized FFSs for raw flash were developed [17] and embedded (real-time) operating systems often ship with multiple FFSs to choose from [2]: For example, Linux and RTEMS bring JFFS and YAFFS to the table, while LittleFS and SPIFFS are shipped by NuttX and RIOT. Researchers also regularly [17, 23, 26] present improved FFSs, such as NF2FS [14].

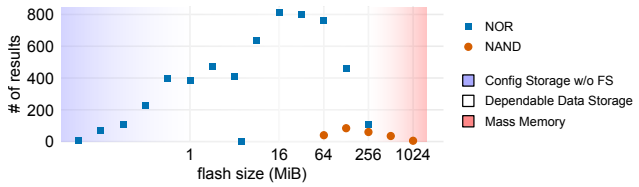


Figure 1: Number of SPI flash chips of different sizes, from mouser.com as of 2025-11-04. Overlay: Different use cases.

Given that FFSs are designed with specific flash sizes, system configurations, and workloads in mind, it is an open question how they perform over a larger parameter space (esp., size and latencies). Such a comprehensive comparative performance evaluation of commonly-used FFSs for resource-constrained devices is currently missing. This paper aims to close this gap.

We present VFLASHSIM, a framework to evaluate any FFS on a large parameter space without requiring hardware. We make the following contributions:

- We present VFLASHSIM, an extensible 'virtual flash'-based simulator for file system testing. [1]
- We evaluate four representative flash file systems for common workloads.
- We identify critical scenarios for different file systems, where performance and functionality break down.

Currently, we have implemented 'virtual-NOR–flash' in VFLASHSIM; we plan to extend this to NAND flash in future work. The rest of this paper is structured as follows: After Section 2 gives an overview of NOR flash characteristics, Section 3 argues why a simulation-based approach is best suited to compare FFSs. Section 4 presents our VFLASHSIM system, which we utilize in Section 5 to evaluate four representative FFSs. Section 6 gives a brief overview of the related work and Section 7 summarizes our results qualitatively.

2 Flash Characteristics

Flash is a type of non-volatile memory, composed of an array of NOR or NAND gates. NOR flash chips are byte-addressable and typically have a MiB-level capacity, while NAND flash chips have a larger capacity, at the cost of requiring ECC due to lower reliability, and supporting only coarser-grained page-level access. Figure 1 provides an overview of commercially available SPI-attached NOR and NAND flash chips across different capacities.

Flash is divided into sectors which are further divided into pages. *Sectors* are the granularity of erase operations (setting bits to 1) and *pages* are the granularity of program operations (setting bits to 0) [5]. On NOR flash, sectors are usually in the 4..8 KiB range, while pages are commonly 256 B in size. Many flash chips support multiple writes to one page without an erase in between as long as only 1's are changed to 0's. Functionally, this in-place-update performs a bit-wise *and* of new and stored data; updates that set a bit from 0 to 1 are ignored. Some chips report such ignored updates as write errors [22]. However, as both SPIFFS (NOR) and YAFFS (NAND) make use of this idiosyncrasy, we posit that it is widely supported among chips of both gate technologies.

[1] Code available at https://github.com/2ck/flash-playground

Table 1: Comparison of NOR Flash Characteristics.

Parameter	IS25LE01G[1]	W25Q256[2]	3DFS256M04[3]
Total size	128 MiB	32 MiB	32 MiB
Page size	256 B	256 B	512 B
Sector size[4]	4 KiB	4 KiB	8 KiB
Fast quad read lat.	4 µs	4 µs	22 µs
Slow read lat.	41 µs	41 µs	207 µs
Write lat. (typ.)	300 µs	400 µs	800 µs
Erase lat. (typ.)	100 ms	50 ms	300 ms
Chip erase (typ.)	90 s	80 s	90 s
Max. read	61 MiB/s	61 MiB/s	22 MiB/s
Max. write	0.8 MiB/s	0.6 MiB/s	0.6 MiB/s

[1] Used in SpacePatch [8], [2] NF2FS evaluation [14], [3] Radiation-hardened chip used in NASA lander [10]
[4] Also supports larger erase units, but file systems usually configured to use smallest.

Commonly, NOR flash is attached to an MCU via serial interfaces from the SPI family, including dual and quad-SPI (QSPI). Reading flash being fast, read speeds are limited by the SPI frequency and the number of available data lines. For write and erase operations, the cell latency is the bottleneck, dominating the comparably faster SPI command latency. While reading a 256 B page takes tens of microseconds, a full chip erase can take several minutes to complete.

Table 1 gives an overview of the capacities and typical operation ratings of three NOR flash chips. We list these chips as they are either used in critical space missions [8, 10] or as an evaluation target in the related work [14]. For all three flash chips, the relations of read, write, and erase latencies are in similar orders of magnitude: (1) Page-sized reads, when done with fast-read instructions at full SPI frequency, saturate the MCU–flash link. (2) Page writes are one to two orders of magnitude slower than reads. (3) Erasing a sector (16 pages in all three chips) is another two orders of magnitude slower than a page write.

The latencies shown in Table 1 are typical values as given by the manufacturers. By fast quad read, we refer to the fast read instruction with four I/O lines, whereas slow read refers to the normal read instruction on one I/O line. For both instructions, we note the time taken to read an entire page at the respective maximum supported SPI frequency. However, in bad conditions (high temperature, low voltage, high wear), up to 10 % of all flash cells do not perform according to the typical range but exhibit worst-case times [24]. Hence, being able to take such latency variations into account is important to support different environments at the system level. Without loss of generality, we will use the typical ratings going forward.

3 Simulating Virtual Flash

Flash latencies and limited program-erase cycles make measuring FFS performance on real hardware notoriously difficult. For example, we must repeatedly erase the entire chip to ensure comparability across benchmark runs. This not only adds minutes of latency, but the induced wears limits the number of benchmark runs. Even worse, on highly-worn flash cells writes become faster while erases slow down [7]. On the other hand, raw NOR flash does not require us to emulate complex hardware abstraction layers, making simulation an attractive option for a realistic evaluation

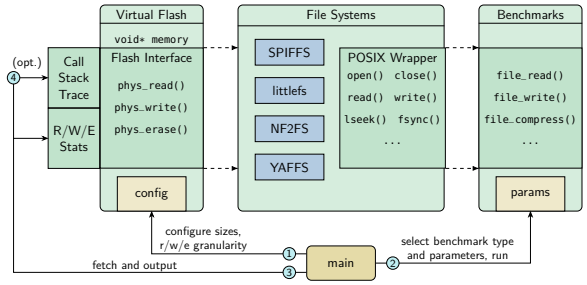


Figure 2: Architecture of VFLASHSIM.

of NOR file systems. Then only the question remains: What do we simulate? The whole system, consisting of CPU, bus, and flash chip, or do we simulate a more abstract variant of the chip?

Flash latencies dominate CPU time. Writes, and especially erases, are slow operations on flash devices, and therefore have the greatest influence on latency [9]. As described in Section 2, even though read operations are orders of magnitude faster than writes and erases, they are still bound by the SPI bus (clock, data lines). But even a fast SPI clock can only be a fraction of the system clock, which is why the impact of CPU operations on the latency of a FFS workload can be neglected. Hence, focusing on *flash wait times* will result in a realistic picture of the FFS performance.

This focus gives further benefits: We can observe, trace and count these operations independent of flash hardware, outside of a microcontroller environment, by attaching the FFS implementation to a software-simulated flash device. Afterwards, we can retroactively estimate FFS performance by multiplying operation counts with manufacturer-provided typical or worst-case times. By scaling with different flash-chip characteristics, we can also determine the sensitivity of a FFS to different latencies.

Our *virtual-flash device* is an abstract flash component that provides interfaces for byte-, page-, and sector-level operations for read, write, and erase. Internally, the virtual flash stores data in normal DRAM region as large as the capacity of the simulated flash chip. The interfaces provide guarded access to this memory and do not allow operations that would be invalid in real hardware. For example, writes greater than a page are prohibited and have to be split up into multiple operations in the file system's flash driver. As we target raw flash chips attached via SPI, only one operation can be in flight at a time (queue depth is 1). By linking our virtual flash with a FFS, we can collect fine-grained information about the number, size, and address-distribution of the different flash operations.

Simulation has important advantages over hardware for measuring FFS performance: (1) Simulation is now limited by CPU power and can be scaled to multiple machines. (2) We have no risk of damaging hardware and do not have to account for hardware aging effects. (3) Fine-grained information like per-page read/write statistics can be saved, which is more difficult on RAM-limited target systems. (4) Backtraces for identifying file-system–internal sources of flash operations can easily be generated without intrusive debugging. (5) Software-implemented fault injection, including virtual-flash checkpointing, is simple to add to our virtual flash.

4 The VFLASHSIM Simulator

In this section, we describe the design of our VFLASHSIM flash-simulation framework (see Figure 2). VFLASHSIM is a C++ program that integrates a virtual-NOR–flash component, four common file systems for NOR flash, three major workload benchmarks, as well as associated functionality for recording detailed statistics. We describe the three main components of VFLASHSIM: (1) the *virtual flash*, (2) the *file system interface*, and (3) the *benchmarks*.

Virtual Flash – We mimic the interface of NOR flash as described in Section 3. At the flash interface, we record statistics about the granularity and target address of each flash operation, which also allows us to reconstruct cell wear information.

We distinguish between partial and whole page reads and writes and record them accordingly. For writes, it is hardware-dependent whether partial writes are faster than whole page writes. Two of our three representative flash chips mention partial writes in the datasheet. Usually, the latency for the first written byte and any additional differ. Our own measurements with IS25LE01G show that the first written byte already takes almost 20 % of the page write time. 3D PLUS write that for their chip, "load[ing] the entire page size program buffer [...] will save overall programming time versus loading less than a page size" [1]. For the following evaluation, we therefore assume that small write operations take as long as it takes to program a whole page.

Optionally, we also record the call-stack trace when a function of our flash interface is called. This allows us to visualize flash-wait times as a flame graph [12], which are often used to visualize profiling and sampling data. In the y-axis, flame graphs show the call stack with distinct levels. On the x-axis, sampled call frames are *not* sorted but hierarchically grouped and ordered by function. Thereby, the width of each box represents a percentage of the total benchmark run time. For example, Figure 3a shows that `__lfs_file_flushedwrite` is active (on the stack) for the whole benchmark. But around 75 % of the time is spent in `__lfs_bd_erase`.

With VFLASHSIM, we get one call frame per flash operation. By plotting this data as a flame graph, the x-axis is relative to the total number of operations. For a flash-wait flame graph that reflects the relative execution time, we have to weight each call-stack sample by the respective operation latency on the x-axis. Both representations can be useful to understand a FFS.

File Systems – FFSs provide similar file and directory operations with different APIs (e.g., numerical file descriptor vs. file pointer). We wrap these operations in a POSIX-like interface, which is passed to the benchmarks that can thereby remain FS-agnostic. This makes VFLASHSIM extensible; only a mount configuration and those POSIX wrappers have to be adapted for supporting a new flash file system. For block file systems like ext, a flash translation layer would additionally be required, but this is not specific to the simulator. Therefore, adding a new file system requires effort comparable to integrating it into firmware, e.g., in an RTOS environment. Currently, we have integrated four representative flash file systems into VFLASHSIM; widely shipped LittleFS, SPIFFS, and YAFFS, as well as the research file system NF2FS.

(1) *LittleFS* [13] is a NOR flash file system designed for minimal RAM usage. It provides power loss resilience through copy-on-write file operations, and performs wear leveling based on block erase counts.
(2) *SPIFFS* [3] is also designed for RAM-constrained systems. Unlike hierarchical file systems such as FAT or ext2, SPIFFS maintains a single-level namespace where directory names are treated as part of the filename string rather than as separate directory structures. This design reduces memory consumption and simplifies flash management but limits organizational flexibility.
(3) *YAFFS* [21] was originally a NAND flash file system, but gained support for NOR flash with version 2. On NOR flash, which lacks NAND's per-page spare areas, YAFFS2's metadata are stored in-band, together with file data. Error correction is disabled for NOR flash. YAFFS stores file system structures in RAM, which speeds up file system operations, but necessitates large amounts of available memory on target devices.
(4) *NF2FS* [14] is a log-structured file system, storing files, directories, and metadata in out-of-place updated log areas. It uses soft updates to ensure crash consistency, finalizing an operation by flipping a single flash bit in an additional transaction header. Wear leveling in NF2FS is performed on a coarser-grained file granularity, instead of on a sector granularity.
Benchmarks – With increasing NOR capacities, it becomes suitable to not only store code and configuration but also sensor-data streams at high ingress rates. Hence, VFLASHSIM includes benchmarks that are representative of typical bursty sensing missions: data ingress, data compression/preprocessing and data outgress/processing. This usage pattern is representative for applications like CubeSats — dependable systems that collect large amounts of data on NOR flash, but have limited downlink windows and bandwidth. It is also applicable to intermittently-connected, e.g., wireless, sensor systems or any other mobile application with local storage demands.

Based on the identified flash usage pattern, we propose three main workloads: (1) Data stream in: Sequential write of one file. (2) Data preprocessing: Sequential or pseudo-random read of one file and write to another. (3) Data stream out: Sequential read of one file.

Benchmarks are parametrizable with file, read and write chunk sizes, as well as the read and write access type. On top of sequential access, we implement an additional pseudo-sequential pattern which continuously seeks forward by four chunks and then reads the previous three. Our benchmarks do not process the data, but only move data.

5 Evaluation

In this section, we quantify FFS performance with VFLASHSIM's data ingress, compress, and outgress workloads. We compare the number of read/write/erase operations performed by each file system to the theoretical minimum that would be necessary to move the payload data around (1x for read/write, 1.5x for preprocess). In addition to flash statistics, we measure RAM usage of each file system, both in the form of static buffers and dynamic allocation via `malloc()`.

For the evaluation, we use the IS25LE01G sizes and timings (see Table 1), including both fast quad and slow normal reads. Although VFLASHSIM collects detailed wear information, we consider long term wear effects out of scope for this evaluation. FFS are commonly configured with only a single erase unit, even though flash chips support erasing whole groups of sectors in one faster operation. Therefore, we do not perform any erase optimizations on the driver level.

5.1 Validation of the Simulation

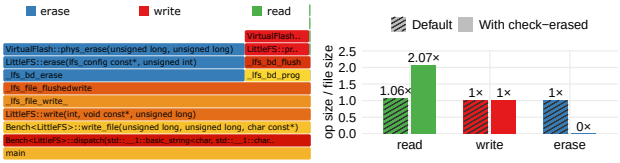
We validate the output of VFLASHSIM with both a comparison to related work and measurements in hardware. We extend the NORBench tool from [14] to record read and write counts in addition to erases. Across all benchmarks in the NORBench suite, VFLASHSIM produces operation counts identical to those obtained from the original tool. In order to verify the accuracy of VFLASHSIM with respect to real hardware behavior, we repeat the read, write, and erase benchmarks from our simulator with an IS25LE01G flash on an nRF52 MCU running the RIOT OS. We run all three benchmarks for LittleFS and SPIFFS, each with 16 different file sizes up to 32 KiB. Despite additional software layers in the form of a standard library and an MTD abstraction layer, the recorded numbers of write and erase operations exactly match those obtained with VFLASHSIM. Only the number of read operations shows a small deviation, e.g., 150 instead of the simulated 146 read operations for a 30 KiB file read in LittleFS. It should be noted that because of latency and chip wear, we did not perform a chip erase before every individual measurement. The effects of buffering and standard library calls can explain these few additional reads.

The total measured benchmark latency matched the expected values derived from the operation counts and datasheet times. Depending on the erase wait time configured in the firmware, workloads involving erase operations could complete around 10 % faster than expected. This is because flash erase operations can finish earlier than the manufacturer-provided typical timing, and a status register check performed after, e.g., 90 ms instead of 100 ms, can detect these early completions. Such a deviation from the typical timing is not expected to remain constant over the device lifetime.
Simulator Limitations – While establishing the accuracy of VFLASHSIM, the hardware measurements nonetheless expose some limitations of simulating NOR flash behavior. While our simulation provides accurate operation counts and latency results, the full breadth of NOR flash timing variation is not currently covered by VFLASHSIM. However, our NOR flash model is well suited to expansion in future work with regard to timing variation based on input data size and layout, sector wear or temperature. Additionally, the simulator's virtual flash interface does not model the effects of a buffering standard library, or additional middleware layers. When a user requires data specific to their system, measurements with the target hardware and software stack can provide additional insights.

5.2 File Systems - Caveats

While running our benchmarks, we discovered different file system caveats present in the FFSs.

LittleFS (LFS for short in figures) exhibited unexpected erase operations when writing to an empty file system placed on a flash that was erased before mounting. Figure 3a shows a flame graph of this situation, recorded with VFLASHSIM and scaled to latency. We



(a) Flamegraph of unfixed LFS. (b) R/W/E bytes per byte of file.

Figure 3: Writing a file in LittleFS with and without checking if a sector is already erased.

(a) Small sequential file read. (b) Small sequential file write, only write operations.

Figure 4: File read and write, with SPIFFS as the outlier.

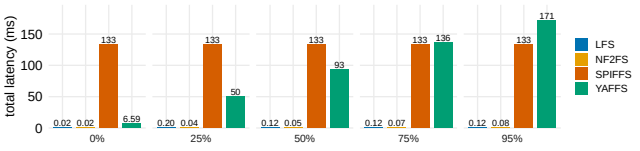
see that for the write workload, ~75 % of the time is spent on erasing flash sectors. The reason for this is that LittleFS does not keep an in-memory mapping of all erased sectors; instead, it performs a costly erase of every allocated sector before writing to it. We fixed this by adding an erased-check to LittleFS' allocation path that reads and compares it to `0xFF` before issuing an erase. This can speed up new sector allocation by three orders of magnitude (× 1562 for IS25LE01G). Figure 3b shows the resulting elimination of slow erase operations, with an increase in read operations. For the evaluation, we only consider the fixed version of LittleFS with the erased-check, as the original can not be meaningfully compared to the other FFSs with respect to latency.

By looking at the number of read/written bytes (see Figure 4a) for the read/write benchmarks, we saw unexpected spikes and slopes for SPIFFS. SPIFFS uses separate lookup pages in each sector to store file metadata, including erase counts for wear leveling. Due to its non-hierarchical design, it has to scan those lookup pages to find a file; if the file does not exist yet, it has to scan all sectors in the flash before creating it. We also observed a nearly threefold write amplification for SPIFFS (see Figure 4b). Due to its low performance, we excluded SPIFFS from most measurements.

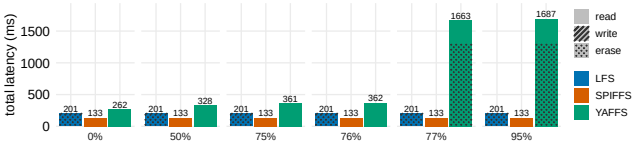
NF2FS also proved to be difficult. For many workloads, we could only get partial measurements as the author-provided implementation contains bugs that caused frequent crashes. Especially for garbage-collection benchmarks, where a larger number of files are deleted, this problem was prevalent.

5.3 File System Mounting

In sensor systems, mounting a file system frequently occurs. For example, a CubeSat system in space will at times reboot uncleanly due to power loss or faults. Therefore, we test clean and dirty re-mounting, where the file system was not properly unmounted and dirty caches were not flushed. In this case, the file system has to ensure that there is no corruption. NF2FS enters an unrecoverable state after a crash, as it can not re-mount without a prior clean unmount.

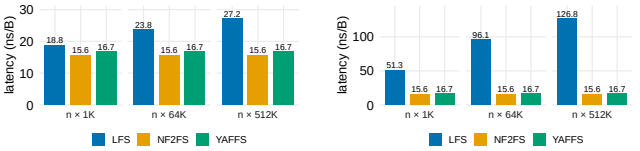


(a) After clean unmount.



(b) After crash.

Figure 5: Mount latency at different fill levels.



(a) Sequential read latency. (b) Pseudo-seq. read latency.

Figure 6: Latencies for file read.

Figure 5 shows the mount latencies at different fill levels. SPIFFS has a constant mount time as it scans the entire file system to count free, used, and deleted pages. Mounting YAFFS takes increasingly more time and reads with higher fill levels as it rebuilds its in-RAM state. LittleFS and NF2FS mount times are significantly lower compared to the other file systems, as they rebuild only a limited amount of state at the price of higher access latencies.

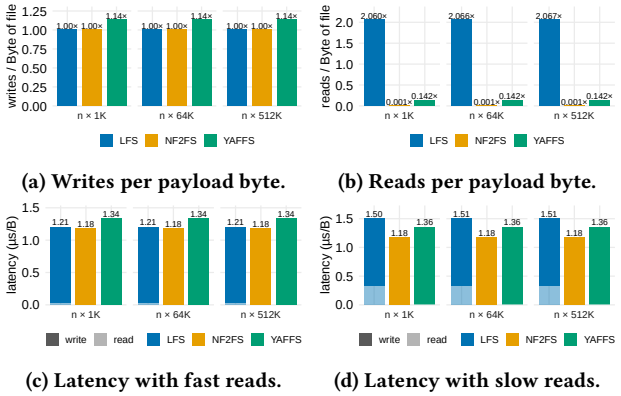
For dirty remounts, YAFFS' behavior is remarkable. Until ~76 % it only shows a slight increase in mount time, which then jumps to a full 1.6 seconds. At this point, YAFFS crosses a threshold that triggers a full garbage-collection cycle. Such spiky behavior is problematic for timing-critical systems.

5.4 Use Case: Sequential Read

We sequentially read files whose size is a multiple of 1/64/512 KiB. As the number of file system operations scales nearly linearly, we apply a linear regression on each set to calculate the ratio of total read bytes to file size. From this, we derive the read latency per input byte (see Figure 6). For such a read-only workload, we base the latency calculation on fast reads, but the relative difference between the file systems is independent of the read speed. Figure 6b additionally shows the results for a workload that seeks forward four chunks, then backward to read the skipped three. LittleFS performs worse with this read pattern, because reads cross sector boundaries more often.

5.5 Use Case: Sequential Write

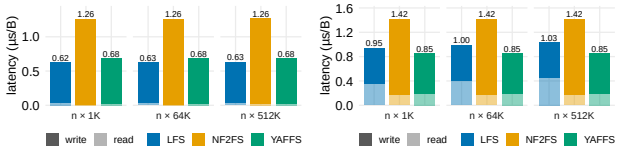
Similar to the read benchmark, we sequentially write a file of different sizes and calculate the ratio of operations per byte (see Figure 7). NF2FS performs best, with no write amplification and only a very small number of reads (e.g., 324 B for a 64 KiB file). LittleFS shows



(a) Writes per payload byte. (b) Reads per payload byte.

(c) Latency with fast reads. (d) Latency with slow reads.

Figure 7: Sequential file write of different sizes.



(a) Compress with fast reads. (b) Compress with slow reads.

Figure 8: Latencies for file compress.

no write amplification, but a read amplification of around two. This is caused by: (1) LittleFS verifying written data by re-reading it, and (2) our read–erase optimization for LittleFS. Below, we plot the latency per byte, where the impact of LittleFS's read overhead becomes apparent for slower read instructions.

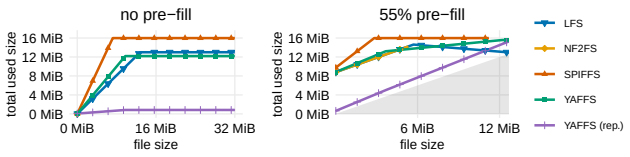
5.6 Use Case: Compression

In the compression benchmark, we read a file page by page, each time writing half the page to a second file. This highlights how file systems deal with multiple open files, and partial writes. The results are mostly the expected combination of the separate read and write results. We see a slight impact of the increased number of reads for large files in LittleFS (see Figure 8b), making it slower than YAFFS for slow reads. All file systems except NF2FS buffer writes up to page size, while the latter directly writes smaller chunks. As described in Section 4, we assume that partial writes take just as long as full page writes in the worst case. Even for chips where partial write times are supplied, writing a page in multiple parts is still slower than a single page program [11, 15].

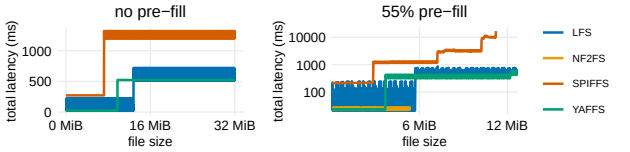
5.7 RAM Usage

For FFSs that rely on volatile in-memory metadata, RAM is a limiting factor. We measure memory usage by tracking dynamic allocations, and adding the size of statically-allocated data structures.

LittleFS and SPIFFS are designed for small embedded systems and rely mostly on static buffers and avoid dynamic allocations. The file system structures of LittleFS are around 280 B, plus an additional 104 B per file (usually on stack), with slight variations per configuration. When mounting, it additionally allocates page-sized read and write caches, and a lookahead for finding free blocks. Each byte of the latter tracks the status of eight flash blocks, so by increasing its size, allocation performance can be improved. When opening a file,



(a) Flash fill level (number of dirty pages). YAFFS (rep.) is fs-reported usage, gray shading represents minimum used space.



(b) Latency with increasing amount of written files.

Figure 9: Garbage collection on 16 MiB virtual flash. Left: Continuous write and delete of files. Right: File system is pre-filled to 55 %, those files are deleted, then more are written.

LittleFS allocates another page-sized file cache. In total, RAM usage remains around 1..2 KiB depending on the configuration but independent of the flash size. SPIFFS uses only fixed-size static buffers: a work buffer of twice the page size and base FS structures (~240 B with caches enabled). A read/write cache (max. 32 pages) speeds up lookups and reduces wear by batching small writes. File-descriptor memory must also be statically allocated (32..72 B per open file), so total RAM varies by configuration; common setups are in the order of ~2 KiB.

NF2FS and YAFFS rely heavily on larger dynamic allocations. Due to a bug, NF2FS read values from uninitialized memory and allocated nearly 1.5 GiB on mount in our tests. After a fix, we measured a more reasonable ~12 KiB peak RAM usage for 128 MiB of flash, and 2.3 KiB for a small 8 MiB flash. This includes around 512 B of cache per open file. For YAFFS, RAM usage is also dependent on flash size, but also on the size of written files. For 128 MiB of flash, we measured allocations up to 609 KiB during mount, increasing to 980 KiB after writing 32 MiB of files, and a peak of 2.4 MiB after filling it completely.

5.8 Garbage Collection

Deleting files marks their associated pages and sectors as reclaimable for a file system. We provoke garbage collection by repeatedly creating and deleting 16 KiB files. In a second benchmark we write files until we reach a fill level of 55 %, then delete/unlink those files. Finally, we continuously write files of a fixed size until the file system runs out of space. Because NF2FS crashes when deleting large or numerous files, we reduce the virtual flash size to 16 MiB, keeping all other characteristics the same. Still, we can only obtain partial measurements for NF2FS. Figure 9a shows the apparent fill level of the flash in the form of dirty pages as files are being written. The 'YAFFS (rep.)' line additionally shows YAFFS-internal reporting of used flash space. Figure 9b plots the latency incurred during writing when the file system has to perform garbage collection. The latency of NF2FS remains stable, though this is because it does not actually perform any garbage collection. When the file system is nearly full, it runs out of space and crashes. SPIFFS, which only starts reclaiming dirty sectors when the file system is nearly full,

shows a more than tenfold increase in latency. In the pre-fill case, it also runs out of space before 12 MiB of files have been written. LittleFS and YAFFS perform better, with the latter triggering garbage collection earlier and showing greater latency stability.

5.9 File System Breaking Points

While LittleFS (with our fix) and YAFFS perform well for the tested workloads, we identified several weaknesses of SPIFFS and NF2FS. SPIFFS is designed for minimal RAM usage, foregoing object lookup data structures in volatile memory (except for an optional file descriptor cache). When opening a file, the first page or pages (depending on configuration) of all logical blocks (at least one physical sector) need to be read, until a matching file is found. When creating a new file, this guarantees as many read operations as there are logical blocks. For our configuration, SPIFFS reads 16 MiB from flash for every newly opened file, which takes 261 ms with fast quad reads and 2.7 s with slow reads. Therefore, creating new files in a tight loop is a highly inefficient scenario. Setting the logical block size to a multiple n of the sector size ameliorates this issue, reducing lookups by a factor of n. This comes at the cost of losing erase granularity as well as usable space, as SPIFFS reserves two logical blocks for metadata.

NF2FS's current implementation proved to contain multiple severe bugs. It sets a default maximum file size of 32 MiB, but actually fails to write files larger than approximately 10 MiB. Files need to be closed in FILO order, because open files are managed in a linked list where only deleting the last entry was implemented. File deletion is similarly affected; no more than 145 files with a size of 256 B can be deleted before NF2FS crashes. Critically, it does not perform garbage collection when writing a file. So, when the file system is full of reclaimable dirty sectors, NF2FS still runs out of space. These limitations in the provided artifact are not architectural issues of NF2FS, but they make drawing fair conclusions from the evaluation difficult.

6 Related Work

Raw-flash simulators exist, but at different abstraction levels. Linux's nandsim [4] and NANDFlashSim [16] provide functional NAND emulation or detailed timing models, and SSD-oriented simulators [19] extend these ideas for FTL research. However, they operate at the NAND/FTL block level and can not execute embedded NOR file systems directly.

Existing flash file system implementations such as LittleFS, SPIFFS, and YAFFS ship with their own test or benchmark programs, but these tools measure only the file system they belong to and are not comparable across implementations. With NORBench [14], Huang et al. present the only prior work that includes a framework for benchmarking multiple NOR file systems. However, NORBench's emulated environment evaluates only NF2FS and records only erase counts, while the other file systems are evaluated solely with respect to total latency on hardware. As a result, NORBench can not generate hardware-independent read/write/erase traces and thereby can not identify which operations contribute to the total latency of a workload.

Table 2: Flash file system comparison.

File system	Mnt	Rd	Wr	Cm	GC	RAM	Relia.
SPIFFS	~	--	--	--	-	++	+
LittleFS (fixed)	++	-	+	+	++	++	++
NF2FS	+	++	++	-	--	++	--
YAFFS	~	++	+	+	++	--	++

7 Discussion and Conclusion

With VFLASHSIM's abstracted simulation, we can make a qualitative comparison of the tested FFSs in different categories (see Table 2). SPIFFS creates large overheads for file read and write operations, and has the highest latency increase during garbage collection. NF2FS shows the best performance for page-sized reads and writes but its usability and reliability is severely degraded by its current implementation. YAFFS and LittleFS (with our fix) remain as the most efficient flash file systems. Generally, YAFFS performs slightly better, and gains an advantage over LittleFS with slower reads due to lower read amplification. However, YAFFS' RAM requirements make it unsuitable for heavily-constrained target systems.

Acknowledgments

We thank our reviewers for their helpful feedback, and especially our shepherd for their guidance and dedication throughout the shepherding process.

This research was supported by the German Aerospace Center (DLR) under the Federal Ministry of Research, Technology and Space (BMFTR) under FKZ 50WB2421A.

References

[1] 3D PLUS. 2022. 3DFS256M04VS2801 256Mbit QSPI TMR NOR Flash Datasheet. https://www.3d-plus.com/products/memory/spi-nor-flash/.
[2] Udit Kumar Agarwal, Vara Punit Ashokbhai, Gedare Bloom, Christian Mauderer, and Joel Sherrill. 2019. Comparison of file systems in RTEMS. *ACM SIGBED Review* 16, 3 (2019), 39–44.
[3] Peter Andersson. 2013. SPIFFS: SPI Flash File System. https://github.com/pellepl/spiffs.
[4] Artem B. Bityuckiy. 2004. nandsim: The NAND Flash Simulator. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/mtd/nand/raw/nandsim.c. Linux kernel module, originally introduced in 2004.
[5] Alexander Buck, Karthik Ganesan, and Natalie Enright Jerger. 2024. FlipBit: Approximate flash memory for IoT devices. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 876–890.
[6] Jiyan Dai. 2019. *Ferroic Materials for Smart Systems: From Fundamentals to Device Applications*. John Wiley & Sons.
[7] Peter Desnoyers. 2010. Empirical evaluation of NAND flash memory performance. *ACM SIGOPS Operating Systems Review* 44, 1 (2010), 50–54.
[8] Deutsches Zentrum für Luft-und Raumfahrt (DLR). 2024. SpacePatch: A smart wearable sensor with a universal data platform to monitor the health of Artemis astronauts. https://event.dlr.de/en/ila2024/spacepatch/. Presented at ILA Berlin Air Show 2024. Accessed: 2025-11-14.
[9] Eran Gal and Sivan Toledo. 2005. A transactional flash file system for microcontrollers.. In *USENIX Annual Technical Conference, General Track*. 89–104.
[10] Adrian-James Gevero and David Rutishauser. 2022. NOR Flash Memory Scrubbing Application for Boot File Preservation of NASA's Descent and Landing Computer (DLC). In *AIAA SCITECH 2022 Forum*. 1833.
[11] Giantec Semiconductor Corporation. 2014. GT25Q64A-S (64 Mbit) Serial NOR Flash Memory Datasheet, v1.4. https://www.giantec-semi.com/uploads/241023/gg/2.3~3.6V/GT25Q64A%20S%20DTR%20DS_v1.4.pdf.
[12] Brendan Gregg. 2016. The flame graph. *Commun. ACM* 59, 6 (2016), 48–57.
[13] Christopher Haster. 2017. LittleFS Embedded Flash Filesystem. https://github.com/littlefs-project/littlefs.

[14] Hao Huang, Yanqi Pan, Wen Xia, Xiangyu Zou, Darong Yang, Liang Shi, and Hongwei Du. 2025. Simplifying and Accelerating NOR Flash I/O Stack for RAM-Restricted Microcontrollers. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 1076–1090.

[15] Integrated Silicon Solution, Inc. 2025. IS25LE01G / IS25WE01G 1Gb Serial Flash Memory Datasheet. https://www.issi.com/WW/pdf/25LE-WE01G.pdf.

[16] Myoungsoo Jung, Ellis Herbert Wilson, David Donofrio, John Shalf, and Mahmut Taylan Kandemir. 2012. NANDFlashSim: Intrinsic latency variation aware NAND flash memory system modeling and simulation at microarchitecture level. In *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 1–12.

[17] Ondrej Kachman, Gabor Gyepes, Marcel Balaz, Libor Majer, and Peter Malik. 2021. Configurable Flash Filesystem for Low-Power Sensor Devices. In *2021 International Conference on Engineering and Emerging Technologies (ICEET)*. IEEE, 1–6.

[18] Seung Soo Kim, Soo Kyeom Yong, Whayoung Kim, Sukin Kang, Hyeon Woo Park, Kyung Jean Yoon, Dong Sun Sheen, Seho Lee, and Cheol Seong Hwang. 2023. Review of semiconductor flash memory devices for material and process issues. *Advanced Materials* 35, 43 (2023), 2200659.

[19] Youngjae Kim, Brendan Tauras, Aayush Gupta, and Bhuvan Urgaonkar. 2009. Flashsim: A simulator for nand flash-based solid-state drives. In *2009 First International Conference on Advances in System Simulation*. IEEE, 125–131.

[20] Laurynas Maciulis and Vytenis Buzas. 2017. Lituanicasat-2: Design of the 3U in-orbit technology demonstration CubeSat. *IEEE Aerospace and Electronic Systems Magazine* 32, 6 (2017), 34–45.

[21] Charles Manning. 2002. YAFFS: Yet Another Flash File System. https://yaffs.net/.

[22] Charles Manning. 2017. *Yaffs Direct Interface (YDI)*. Technical Report. Aleph One Ltd. https://yaffs.net/sites/default/files/downloads/YaffsDirect.pdf Version dated 2017-06-07.

[23] Yanqi Pan, Zhisheng Hu, Nan Zhang, Hao Hu, Wen Xia, Zhongming Jiang, Liang Shi, and Shiyi Li. 2022. HNFFS: Revisiting the NOR flash file system. In *2022 IEEE 11th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. IEEE, 14–19.

[24] SkyHigh Memory. 2019. *AN202731: Understanding Typical and Maximum Program/Erase Performance*. Application Note 002-02731. SkyHigh Memory. https://www.skyhighmemory.com/download/applicationNotes/002-02731_AN202731_Understanding_Typical_and_Maximum_Program_Erase_Performance.pdf Rev. C.

[25] Christoforos Vasilakis, Alexandros Tsagkaropoulos, Angelos Motsios, and Dionysios Reisis. 2025. Autonomously Reconfigurable Telemetry and Monitoring System for CubeSats. In *2025 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, Vol. 1. IEEE, 1–5.

[26] Runyu Zhang, Duo Liu, Xianzhang Chen, Xiongxiong She, Chaoshu Yang, Yujuan Tan, Zhaoyan Shen, Zili Shao, and Lei Qiao. 2022. ELOFS: An extensible low-overhead flash file system for resource-scarce embedded devices. *IEEE Trans. Comput.* 71, 9 (2022), 2327–2340.