mnemofs: A NAND flash file-system for Apache Nuttx

Saurav Pal, Alan C. Assis

***Abstract*—NAND flash memory is a key enabler for modern sensor systems due to its low cost, compact size, and low power consumption. However, its unique characteristics such as erase-before-write requirements, bad block management and randomized bit-flips demand a specialized file system design. This letter proposes mnemofs, a file system that incorporates wear leveling, power-loss resilience, block garbage collection, low RAM consumption and small binary size to maintain reliability while utilizing NAND flash's advantages in embedded systems. We analyze tradeoffs propose enhancements for sensor applications where robust, long-term data storage is critical.**

***Index Terms*—Sensor, data, file system, embedded systems, NAND flash.**

I. Introduction

SENSORS require a way to store large volumes of data on the device itself, and it's desired that the space is cheap and compact. NAND flash is one of the types of programmable, non-volatile storage technologies built using NAND gates. One of the greatest advantages of this storage technology is its high net bit storage density (NBSD) [1], [2]. The smaller number of lines needed for NAND flashes contributes significantly to this advantage. These features make it the ideal choice for sensors. Every NAND flash type has one or more non-erased states, depending on the its BPC (bits per cell), in addition to a single erased state. This single erased state has all the bits set [1].

NAND flashes are very singular in their characteristics. The smallest read (and write) unit for NAND flashes is a single page [1], commonly 16 KiB in size [1], [3], [4]. However, a page can only be updated by first erasing the block it is part of. From the perspective of a file system, the most relevant units of a NAND flash are blocks and pages. Each page consists of a data and a spare area, in an approximate 4:1 ratio [1]. The spare area contains some metadata about the page, such as the bad block marker, error correction codes (ECCs), etc. Some of this data ensures data integrity through redundancy in case of bit flips, which occur at a higher frequency in NAND flashes than their NOR counterparts due to their compactness.

A lot of file systems have emerged over the decades. Block-based file systems like FAT32 [5] and EXT2 [6] were simple and fast. However, they are not effective against power losses. An unfortunately timed power-loss could potentially render the data on device unreadable. On the other hand of the spectrum, log-based file systems like JFFS [7], YAFFS [8] and SPIFFS [9] treat the entire storage space as a single journal, and going through the logs sequentially allows a recreation of the entire file system. These are power-loss resilient, as logs are appended with a checksum to guarantee a completed write operation, and inherently leveling the wear as the logs are written sequentially. However, reading data is slow (with a usual time complexity of $O(n^2)$) and they consume a lot of memory (usually with a space complexity of $O(n)$). Further, a lot of earlier logs may be rendered redundant due to newer logs. Once the device is full, cleaning up the device to allow more logs to be stored can be quite challenging and memory consuming as well.

Journaling file systems like EXT 3 [10], EXT 4 [11] and NTFS [12] are a middle ground approach. These have a file system tree like the block-based file systems and a small journal like log-based file systems. This helps them to perform operations relatively fast and power-loss resilient, but this leads to an increase in the wear in the region of the journal and the complexity of the file system. Often, traditional file systems can be run over Flash Translation Layer (FTL) as well. UBIFS uses an Unsorted Block Images (UBI) layer [13], and maintains in-memory data structures for wear leveling and its overhead makes it unsuitable for small NAND flashes. Thus we present mnemofs, a file system which is geared towards optimizing the typical file system usage demonstrated by sensors while navigating the unique challenges of NAND flashes. It uses an ingenious CTZ skip list data structure for representing files.

II. Counter Trailing Zero (CTZ) Skip Lists

A. Counter Trailing Zero Operation

The Count Trailing Zeros operation represents a fundamental bitwise computation that determines the number of consecutive least significant zero bits in the binary representation of a positive integer x. Formally, we may define the CTZ of a natural number x as $\tau(x)$ such that:

$$\mathcal{T}(x) = \left\{ i \in \mathbb{N}_0 \mid 2^i \mid x \right\},\ x \in \mathbb{N} \tag{1}$$

$$\tau(x) = \max\left(\mathcal{T}(x)\right),\ x \in \mathbb{N} \tag{2}$$

Notably, contemporary processor architectures including x86 [14] and ARM [15] implement this operation as a single machine instruction, thereby achieving optimal computational complexity of $\mathcal{O}(1)$ in both time and space.

B. Architecture

The CTZ skip list [16] is a specialized variant of the traditional skip list data structure, distinguished by two principal characteristics:

- **Pointer Cardinality Variability**: Each CTZ block maintains a number of pointers that is correlated to its positional index within the structure.
- **Reverse Linkage Orientation**: In contrast to conventional implementations, all pointers originating from a CTZ block at index i exclusively reference antecedent blocks satisfying $k < i$.

1) Pointer Set Formalization: For a CTZ block residing at index x, we formally define its pointer set $\mathcal{P}(x)$ as follows:

$$\mathcal{P}(x) = \begin{cases} \emptyset & \text{if } x = 0 \\ \{x - 2^i \mid i \in \mathcal{T}(x)\} & \text{if } x \in \mathbb{N} \end{cases} \tag{3}$$

Therefore, the cardinality of this pointer set exhibits the following properties:

$$|\mathcal{P}(x)| = \begin{cases} 0 & \text{if } x = 0 \\ |\mathcal{T}(x)| = \tau(x) + 1 & \text{if } x \in \mathbb{N} \end{cases} \tag{4}$$

Assuming a pointer size of w_b bytes and a fixed block size of b bytes, the available storage capacity for data within a CTZ block at index x computes to $b - w_b \cdot |\mathcal{P}(x)|$ bytes.

2) Pointer Location: The mnemofs implementation introduces a modification to the original CTZ skip list design [16], wherein CTZ blocks correspond directly to NAND flash pages. In this configuration, pointers are positioned at the terminal segment of each CTZ block. For a non-zero CTZ block index x, each pointer referencing block $x - 2^i$ (where $i \in \mathcal{T}(x)$) resides at offset f such that:

$$f = b - w_b \cdot (i + 1) \tag{5}$$

3) Offset to Index Conversion: The mapping from byte offset f to CTZ block index x and intra-CTZ-block offset p follows these relations, which are modified versions of the original equations from C. Haster [16]:

$$x = \left\lfloor \frac{f - w_b \cdot (\kappa(\lfloor f/(b - 2w_b) \rfloor - 1) + 2)}{b - 2w_b} \right\rfloor \tag{6}$$

$$p = f - (b - 2w_b) \cdot x - w_b \cdot (\kappa(x) + |\mathcal{P}(x)|) \tag{7}$$

4) Traversal Algorithm: Traversal between indices x_1 and x_2, such that $x_1 \geq x_2$, proceeds via a novel two-phase greedy approach:

a) Rising Phase: Increases stride length iteratively, where the stride at every iteration is $2^{\tau(x_{\text{current}})}$:

$$x_{\text{new}} = x_{\text{current}} - 2^{\tau(x_{\text{current}})} \tag{8}$$

The phase completes within at most $31 - \lambda \cdot (x_2 \oplus x_1)$ iterations, where $\lambda(n)$ is the Count Leading Zero function and is formally defined as:

$$\lambda(n) = \begin{cases} 32 & n = 2^{32} - 1 \\ \max\{k \in \mathbb{N} \mid n < 2^{32-k}\} & \text{otherwise} \end{cases} \tag{9}$$

b) Falling Phase: Decreases strides using the most significant set bit. The set of set bits of a number can be described as:

$$\mathcal{S}(d) = \{2^i \mid (d \,\&\, 2^i) \neq 0\} \tag{10}$$

Where, $\&$ denotes the bitwise AND operation. This phase selects maximal elements from $\mathcal{S}(x_{\text{current}} - x_2)$ without replacement every iteration, where x_{current} is the index of the CTZ block under consideration.

This algorithm significantly reduces travel between blocks as illustrated by Figure 1.

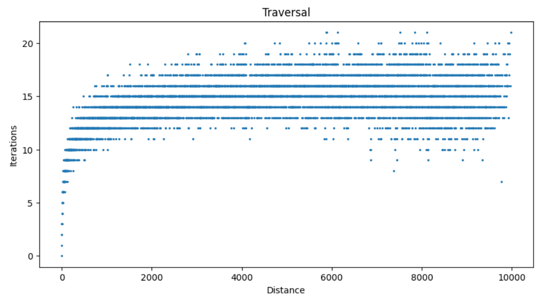

Fig. 1. CTZ skip list traversal

5) Header Structure: A CTZ skip list header comprises of the terminal CTZ block index, the physical page address of the CTZ terminal block and the size of the CTZ skip list.

6) Modification Operation: The CoW conditions follow specific update rules. While blocks with indices $x_k < x$ remain unchanged during modifications, all blocks where $x_k \geq x$ must be updated to reflect the changes. A special case is when a new block is being appended at index $x+1$, where x is the last available index, the operation requires only two key steps: first, creating all necessary pointers specified in $\mathcal{P}(x+1)$, and second, updating the header to reference the new block at $x+1$ instead of the one at x.

III. Directory System Architecture

Directory operations exhibit different usage patterns between human users and sensor systems. While human-computer interaction frequently involves directory manipulations, sensor applications predominantly operate at the file level. This fundamental difference motivates mnemofs's minimalistic directory design, which prioritizes computational efficiency, minimal memory footprint, and compact binary representation. The dominance of sequential access in directory related system calls further justifies our implementation choice of a singly-linked list of fixed-size entries called directory entries (dentries). Each directory entry serves as a comprehensive metadata container for both files and subdirectories. Along with the usual metadata about the file system element, it includes a 32-bit cryptographic hash of the name, a single-byte name length descriptor and the name string itself.

The search algorithm implements an efficient two-phase verification process. The first phase performs a constant-time 32-bit hash comparison, followed by exact string matching only when necessary. The hash function, described in Algorithm 1 and visualized in Figure 2, was specifically designed to achieve uniform distribution across the 32-bit value space while maintaining prefix independence—ensuring that names with common prefixes produce distinct hashes, and results in a Shannon Entropy of 16.61. Notably, the function guarantees a non-zero output for null names and deterministic output, while serving as a write completion checksum.

Algorithm 1 Deterministic Hash Computation

```
Input: Byte array $arr$ of length $sz$
Output: 32-bit unsigned integer hash value
1: Initialize $hash\_val \leftarrow 0$
2: Compute midpoint $n \leftarrow \lfloor sz/2 \rfloor$
3: for $i \leftarrow 0$ to $sz - 1$ do
4:     $term1 \leftarrow arr[i] + i$
5:     $term2 \leftarrow arr[sz - i - 1] \gg (sz - i - 1)$ {Right shift}
6:     $product \leftarrow (term1 \times term2) \bmod 2^{32}$
7:     $xor\_val \leftarrow product \oplus |n - i|$
8:     $shift\_amount \leftarrow i \bmod 32$
9:     $hash\_val \leftarrow (hash\_val + (xor\_val \ll shift\_amount)) \bmod 2^{32}$
10: end for
11: return $hash\_val$
```

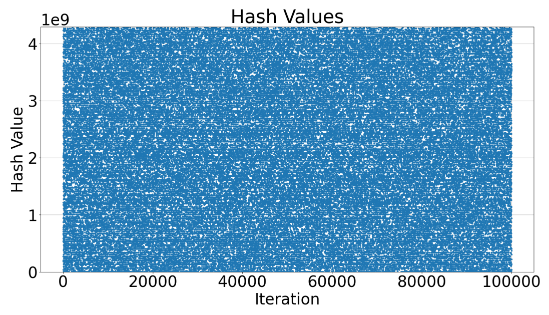

Fig. 2. Distribution of hash values across 100,000 iterations

A. Update Semantics

As updates to directories are rare for sensors, modifications involve making a copy of the directory is made with changes, and then this update is propagated upwards in the tree till it reaches the root. This approach provides strong consistency guarantees while maintaining the system's performance characteristics, particularly important for sensor applications where directory updates are infrequent but must be handled reliably when they occur.

The master node serves as the root pointer for the entire file system hierarchy. Given the dynamic nature of file system operations, where the root may undergo multiple updates due to directory modifications or system flushes, each master node

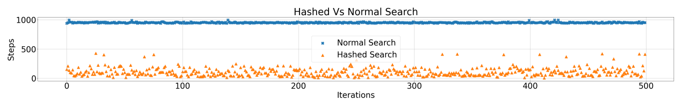

Fig. 3. Performance comparison between traditional and hashed search methods.

incorporates a revision number. This design enables efficient identification of the latest master node instance. Additionally, master nodes contain supplementary metadata critical for file system operation and recovery.

B. Master Block Implementation

To ensure reliable access to the root pointer while accommodating frequent updates, mnemofs employs a specialized storage scheme for master nodes. These nodes are stored sequentially within dedicated master blocks, with each master node occupying one page of storage. The system maintains two identical master blocks for redundancy, with each block serving as a mirror of the other. The active journal contains pointers to the current master blocks in use. This is shown in Figure 4. A master block reaching capacity triggers an automatic file system flush operation, ensuring continuous availability of master node storage space.

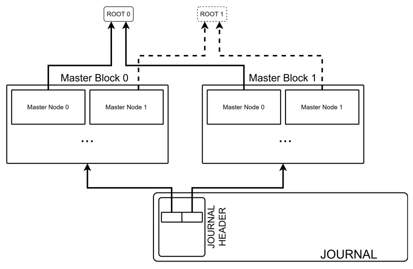

Fig. 4. Master Block

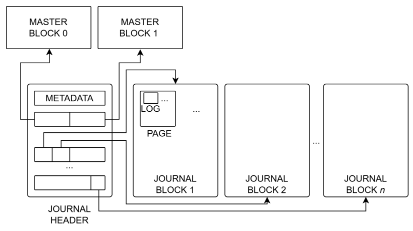

Fig. 5. Journal

IV. Journaling Mechanism

The journal subsystem provides atomic update semantics for CTZ skip list modifications. Whenever a CTZ skip list undergoes changes, its updated header information is recorded in the journal as a log entry. The journal structure can be visualized in Figure 5. The structure consists of n log blocks accompanied by a dedicated header block that serves as the control center for journal operations. This header maintains several critical metadata elements, notably a unique magic number for identification, the current revision number, and pointers to the master blocks. Each log notably contains a write-completion checksum at the end.

V. Allocation Subsystem

The mnemofs allocator manages physical storage allocation at both page and block granularities. Drawing a heavy inspiration from littlefs's block allocator [16], our implementation

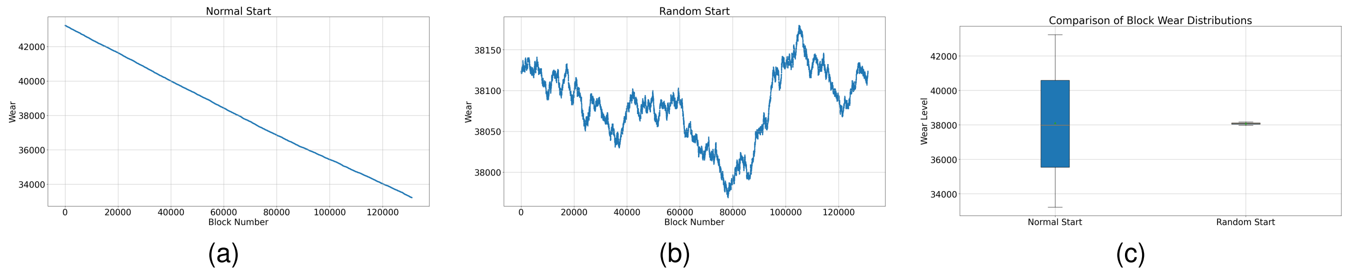

Fig. 6. The difference between allocation styles. (a) Normal Allocation starting from the start of the device, (b) Random Allocation starting from a random location on the device, (c) Box plot on the wear.

TABLE I
ALLOCATION METHODS

Parameter	Normal Start Allocation	Random Start Allocation
Q1	35551.75	38052
Q2 (Median)	37991	38086
Q3	40587	38119
IQR	5035.25	67
R.M.S. Mean	38191.48	38081.77
Maximum	43227	38180
Minimum	33227	37969
Range	10000	211
σ	2893.14	43.81

employs cyclic sequential allocation to achieve uniform wear distribution across the storage medium. Power failure analysis reveals an inherent bias in allocation probability toward lower-addressed blocks, as demonstrated in Figure 6 [16]. To mitigate this uneven wear pattern, the allocator incorporates a randomization mechanism during initialization, selecting a random starting offset within the address space. Figure 6 illustrates the improved distribution achieved through this approach. Table I and Figure 6 reveal the smaller standard deviation.

VI. FLUSH

The file system flush operation is initiated either when the journal or the master blocks are full. In both cases, the journal gets emptied when the flush operation is finished. The entire file system tree is updated in a postfix-like manner. Following this, a new master node is written as the root is finally updated, and the journal is moved. The journal moves after every flush in order to prevent a skew in wear in the journal's location. Normally, this results in a time complexity of $O(mn)$ where m refers to the number of logs in the journal, and n the number of items in the FS tree. However, by bounding the journal to hold a fixed number of logs, and thus having a fixed size, this can be reduced to $O(n)$.

VII. CONCLUSION

Thus mnemofs, a file system built for increasing storage efficiency and responsiveness for sensor data, fulfills its task. Appending data to a file has a time complexity of $O(1)$, while writing in the middle has $O(log_2 n)$ time complexity. Similarly, it is $O(d)$ for traversing a directory, where d is the number of elements in the directory, and $O(d \cdot log_2 n)$ for updating a directory or performing any directory operations. Further, flush operation takes a $O(k)$ time complexity, where k is the number of items in the file system tree.

REFERENCES

[1] C. Monzio Compagnoni, A. Goda, A. S. Spinelli, P. Feeley, A. L. Lacaita, and A. Visconti, "Reviewing the evolution of the nand flash technology," *Proceedings of the IEEE*, vol. 105, no. 9, pp. 1609–1633, 2017.

[2] P. Desnoyers, "Empirical evaluation of nand flash memory performance," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 1, p. 50–54, Mar. 2010. [Online]. Available: http://dx.doi.org/10.1145/1740390.1740402

[3] S. Choi, D. Kim, S. Choi, B. Kim, S. Jung, K. Chun, N. Kim, W. Lee, T. Shin, H. Jin, H. Cho, S. Ahn, Y. Hong, I. Yang, B. Kim, P. Yoo, Y. Jung, J. Lee, J. Shin, T. Kim, K. Park, and J. Kim, "19.2 a 93.4mm2 64gb mlc nand-flash memory with 16nm cmos technology," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2014, pp. 328–329.

[4] M. Helm, J.-K. Park, A. Ghalam, J. Guo, C. wan Ha, C. Hu, H. Kim, K. Kavalipurapu, E. Lee, A. Mohammadzadeh, D. Nguyen, V. Patel, T. Pekny, B. Saiki, D. Song, J. Tsai, V. Viajedor, L. Vu, T. Wong, J. H. Yun, R. Ghodsi, A. D'Alessandro, D. Di Cicco, and V. Moschiano, "19.1 a 128gb mlc nand-flash device using 16nm planar cell," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2014, pp. 326–327.

[5] H. Zhao, X. Li, L. Chang, and X. Zang, "Fat file system design and research," in *2015 International Conference on Network and Information Systems for Computers*, 2015, pp. 568–571.

[6] R. Card, T. Ts'o, and S. Tweedie, "Design and Implementation of the Second Extended Filesystem — web.mit.edu," http://web.mit.edu/tytso/www/linux/ext2intro.html, [Accessed 06-07-2024].

[7] D. Woodhouse, "Jffs: The journalling flash file system," in *Ottawa linux symposium*, vol. 2001. Citeseer, 2001.

[8] C. Manning, "How yaffs works," *Retrieved April*, vol. 6, p. 2011, 2010.

[9] P. Andersson, "Spiffs: A wear-leveling embedded file system for spi flash," Open Source Community, Tech. Rep., 2015. [Online]. Available: https://github.com/pellepl/spiffs

[10] M. Cao, T. Y. Tso, B. Pulavarty, S. Bhattacharya, A. Dilger, and A. Tomas, "State of the art: Where we are with the ext3 filesystem," in *Proceedings of the Ottawa Linux Symposium (OLS)*. Citeseer, 2005, pp. 69–96.

[11] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: current status and future plans," in *Proceedings of the Linux symposium*, vol. 2. Citeseer, 2007, pp. 21–33.

[12] R. Russon and Y. Fledel, "Ntfs documentation," *Recuperado el*, vol. 1, p. 2, 2004.

[13] A. Hunter, "linux-mtd.infradead.org," http://www.linux-mtd.infradead.org/doc/ubifs_whitepaper.pdf, 2008, [Accessed 06-07-2024].

[14] I. Corporation, "Tzcnt — count the number of trailing zero bits," accessed April 5, 2025. [Online]. Available: https://www.felixcloutier.com/x86/tzcnt

[15] A. Limited, "Ctz: Count trailing zeros." accessed April 5, 2025. [Online]. Available: https://developer.arm.com/documentation/ddi0602/latest/Base-Instructions/CTZ--Count-trailing-zeros-

[16] C. Haster, "littlefs," accessed April 5, 2025. [Online]. Available: https://github.com/littlefs-project/littlefs/blob/master/DESIGN.md