A FLASH RESIDENT FILE SYSTEM FOR EMBEDDED SENSOR NETWORKS

Scott Fazackerley, Ramon Lawrence[*]

University of British Columbia Okanagan
scott.fazackerley@alumni.ubc.ca, ramon.lawrence@ubc.ca

ABSTRACT

Many embedded devices, especially those designed for environmental sensor logging, have extremely limited RAM, often less than several kilobytes. Logged data is stored on flash memory and needs to be easily managed at low energy cost. A file system is required to efficiently manage the device, specifically dealing with wear leveling of the flash memory. Previous flash file systems, even those designed for small memory devices, still consume a reasonable amount of RAM (1K or more). In this paper, we present a flash file system that supports record level consistency with the entire file system and address mapping functionality stored on flash memory. Although this results in a marginally higher read cost, RAM utilization is less than 150 bytes and the read cost in terms of energy usage is less. The key idea is that NOR flash used on these devices supports direct byte reads not supported by NAND memory which allows page translation and data storage to require less memory and consume less energy.

Index Terms— wireless sensor network, flash memory, file system, microprocessor

1. INTRODUCTION

Wireless sensor networks are an efficient way to gather sensed data from a large physical area without the need for hard wired infrastructure. Wireless sensor networks consist of a series of wireless sensing devices that have the ability to measure parameters regarding their physical environment. Devices are typically small 8-bit devices [1] that are constrained in terms of power, persistent storage and runtime memory [2]. Numerous hardware platform choices are available with the majority of research being conducted on the Telos, Btnode and MicaZ platforms [3]. Devices intercommunicate using a wireless link [1]. Although there are many different communication paradigms, IEEE 802.15.4 [3, 4] has emerged as the dominate choice. It allows for low data rate, low power ad hoc communications between devices.

The management of data sampling and transmission in a sensor network can be handled by a custom application or by one of the over 37 different operating systems available for wireless sensing devices [5]. In the majority of existing systems local data storage has not been a focus. Sensed data are pushed unprocessed back to the sink or collection point for analysis. With the increased availability of low cost flash ($0.003 per kbyte) as well as the computation vs. communications trade off [6], it is now possible for nodes to maintain results locally. Significant savings in energy can be realized by storing results locally for processing and aggregation in contrast to sending all data across the network [7][8].

Recent work has investigated how to efficiently store data in flash memory [9] on low power, memory constrained devices. Researchers have investigated using flash specific operating systems to abstract the physical storage away from the application while taking advantage of the architecture of flash memory [2][10][11]. Utilization of flash memory without the use of wear leveling or other erase/write normalization strategies can accelerate device failure.

While these file systems present novel and efficient methods for storing the data, they all rely on extensive sensor processor resources in terms of RAM or EEPROM. Some have trouble scaling to large flash memories or many files. To address these challenges we propose FlaReFS, a **Fla**sh **Re**sident **F**ile **S**ystem. FlaReFS has minimal RAM usage and all file system data structures are maintained in flash memory. It uses direct reads on NOR flash and on-chip buffers to reduce energy costs and improve performance. This strategy allows FlaReFS to be used even with the smallest embedded devices.

The contributions of this paper are:

- A novel flash resident file system with a minimal RAM footprint for direct read, dual buffered NOR flash
- A time and energy analysis of direct memory read versus buffered page read
- A dynamic wear leveling algorithm for flash memory

The paper outline is as follows. In Section 2, we overview previous flash file system approaches and discuss the layout and limitations of flash memory devices. An energy analysis is conducted in Section 3 comparing direct versus buffered page reads. Our file system is presented in Section 4. Simulation results are presented in Section 5. The paper closes with future work and conclusions.

[*]The authors gratefully acknowledge the support of the Natural Sciences and Engineering Research Council of Canada.

2. BACKGROUND

With the use of flash memory in wireless sensor networks, work has focused on developing file systems that can accommodate its different performance characteristics. With the inability of flash to re-write data without first erasing a page, research has examined the use of the log based file systems [12][13]. While suitable for complex processors, a direct implementation of these systems on a microprocessor is infeasible due to lack of system resources [2]. Custom designed file systems such as JFFS, JFFS2 [14] and YAFFs [15] are not suitable for use in resource constrained systems, as they maintain data structures in RAM for file system control. These flash file systems are designed to work on systems with large amounts of RAM and processing power.

Smaller systems have evolved from these that are more suitable for wireless sensor platforms that are constrained in terms of power and resources. Matchbox [16], a byte structured file system implemented in TinyOS, supports basic wear leveling as well as multiple files. It is designed for logging applications but supports only append operations. Modification of existing files is not possible. Additionally, its RAM footprint will grow with the number of files in the system [2].

In [2], the authors present ELF which is a log based file system. It relies on the host microprocessor EEPROM to store the directory structure for the system. Unlike traditional log based file systems, it caches multiple writes to reduce the write overhead. It maintains log entries on separate pages to improve fault tolerance. It also maintains a garbage collector to reclaim pages and relies on a page write counter stored in the metadata of each page. Garbage collection is only triggered when the number of free pages drops below a given threshold which is stored as a bitmap in RAM.

Microhash [17] provides a primitive framework for storing and indexing temporal data based on page chaining. It maintains numerous data structures in RAM and utilizes a naive garbage collection strategy. In practice, the runtime RAM requirements make it infeasible for small sensor devices [11].

Capsule is a cross device file system that uses object abstractions [11] to store data. If offers a wide selection of data objects such as stacks, streams and queues to store data but requires a large RAM footprint. It relies on per object buffering in microprocessor RAM and is based on a logging file system. Similar to other systems, it supports a garbage collector which is triggered when the amount of available space in the system falls below a certain threshold. It also supports check pointing and rollback of objects.

2.1. Flash Memory

Flash memory offers advantages over other types of solid state devices in terms of increased capacity, speed and energy usage [8][17] which makes it particularly attractive for energy constrained devices such as wireless sensor nodes. Flash is available in two different formats. NAND flash is the most commonly found format [18], and is characterized by fast access for sequential byte access, high density and durability, but it exhibits high latency in startup and can be prone to bit errors. Data is only accessible in a page format [8][17] which limits the types of read and write actions. NAND flash is typically accessed in a parallel fashion, requiring a high pin count commitment from the host processor which may make it unsuitable for small, low pin count devices.

NOR flash is typically less dense and less energy efficient but is available in either a parallel or serial access format [8]. It offers other advantages in terms of device organization as it may be either word or page orientated. One common characteristic in all flash devices is that data cannot be overwritten without first being erased. Erase operations are slower than other operations on the device and typically must occur in physical blocks or sectors, which can pose a memory management problem for small microprocessors. Each page in the device has a limited number of erase cycles (typically 100,000) before the device starts to fail. A leveling algorithm [19] can be used to amortize the cost of the write/erase cycle across all pages to extend the effective life of the device.

For our application, we are using the Atmel ATD45DB161 dataflash device (www.atmel.com) with two on board SRAM buffers, which is a larger capacity version of the device found on the Mica motes [8]. This is a unique NOR flash device that has a page orientated design with a high speed serial interface. Data can be accessed both in a page-wise fashion through buffers or read directly in a byte-wise fashion bypassing all buffers similar to serial EEPROM, through a four wire SPI interface. The device consists of 4096 x 528 byte pages organized into eight page erase blocks. Blocks are organized into 32 sectors. The device also supports individual page erase (at a higher energy cost) as well as sector and chip erase functionality.

3. ENERGY COSTS FOR DIRECT READS

The following energy cost calculations are for the AT45DB161 family of flash memory and demonstrate the significant advantage of using direct reads over buffered page reads. Algorithms using these chips should adapt their operation accordingly. This device supports different direct memory read methods which allows the data to bypass internal buffers, leaving buffered data unchanged. The Low Frequency Continuous Array read, which support bus speeds up to 33 Mhz, requires four setup bytes followed by one byte for each sequential data to be clocked onto the SPI bus. In this analysis we exclude edge transition times as they are considerably smaller than device setup times. The manufacturers nominal values are used for timing and power analysis. The byte cost for accessing is

$$bytes_{total} = 4 \textit{ setup bytes} + n \textit{ data bytes} \tag{1}$$

where n is the number of sequential bytes to be read from a page. It then follows that the total time to transfer n bytes of data off the flash device on an SPI bus is

$$\begin{aligned} t_{DR} &= \frac{bytes_{total}}{SPI_{clk}/8} \\ &= \frac{4 + n \textit{ bytes}}{SPI_{clk}/8} \end{aligned} \tag{2}$$

where SPI_{clk} is the SPI clock rate in Mhz, and t_{DR} is the time in seconds to read n sequential bytes. The total energy per byte for a direct read can be expressed as

$$E_{DR} = I_{Read} * V * t_{DR} \tag{3}$$

where I_{Read} is the current draw for the flash memory device for a read operation, V is the operating voltage, and E_{DR} is the energy cost per byte in Joules.

For a series of bytes to be read from the device using a buffered read, the page containing the bytes of interest must be first read into one of the two SRAM buffers and then transfered across the SPI bus. Similar to the direct read, there is a four byte setup cost to initiate the page transfer to buffer in addition to a 200 mircosecond delay while the page is being transfered to the buffer. Once the page is loaded into the buffer the data can then be read requiring five setup bytes and then one clock byte for each data byte required. The total time for a buffered read is

$$\begin{aligned} t_{BR} &= \frac{4 \textit{ setup bytes}}{SPI_{clk}/8} + 200\mu s + \\ &\quad \frac{5 \textit{ setup bytes} + n \textit{ data bytes}}{SPI_{CLK}/8} \\ &= \frac{9 + n \textit{ bytes}}{SPI_{clk}/8} + 200\mu s \end{aligned} \tag{4}$$

where n is the number of sequential bytes to be read from a page, SPI_{clk} is the SPI bus speed in Mhz and t_{BR} is the byte time is seconds for a buffered read. It then follows from Equation (4) that the energy to read n bytes with a buffered page read is

$$E_{BR} = I_{Read} * V * t_{BR} \tag{5}$$

where E_{BR} is the energy cost per byte in Joules.

Consider the following example for accessing a one byte value from the flash memory using direct read where the operating voltage V=3 volts, the nominal read current I_{Read}=0.007 amp, and the SPI bus speed is 4 Mhz. From Equation (2) the total access time is $10\mu s$. The total energy cost for a byte read from Equation (3) is $0.21\mu J$.

With a buffered page read where the operating voltage V=3 volts, the dataflash nominal read current I_{Read}= 0.007 amp, and the SPI bus speed is 4 Mhz using Equation (4). From Equation (2) the access time for a one byte read is 220 μs and the energy cost from Equation (5) is 4.62 μJ.

In comparing the two different read methods for this example, the direct read has an setup overhead of 0.168 μJ while that buffered page read has an overhead of 4.578 μJ regardless of length. Thus, the direct read is favored over buffered page read regardless of size.

4. FLASH RESIDENT FILE SYSTEM

The goal of the Flash Resident File System (FlaReFS) is to produce a file system that is transactionally robust with a small RAM footprint that can run on a sensor node. Unlike other systems [2][10][11][17], FlaReFS maintains almost the entire structure of the file system in external flash memory. The lowest level of the file system provides physical to logical page translation as well as control blocks for monitoring page status and wear leveling. By using the ability to do direct reads from the flash memory, on chip buffers are used strictly for writing data. As a result, no page data is buffered in microprocessor RAM, leaving memory free for other application components. A buffer manager is responsible for controlling the allocation of the buffers on the device. Unlike other flash file systems, no separate garbage collection is needed.

The core idea is that the logical to physical page translation is encoded in flash memory not in RAM. The logical address space, shown at the top of Figure 1 consists of N pages numbered 0 to $N - 1$. The first page, labeled **M**, is the Master Table Translation Page (MTTP) followed by fifteen Secondary Table Translation Pages (STTPs)(A system with 4096 x 528 byte pages will map to one MTTP and 15 STTPs). Each Table Translation Page (TTP) stores physical addresses of logical pages in the system. The MTTP stores the physical addresses for the first 255 logical pages including the 15 STTPs.

By exploiting direct reads and the physical to logical mapping structure, the physical address of any page in the system can be determined in at most 2 direct reads of 2 bytes each. To lookup the physical address of a logical page, the TTP is determined as:

$$TTP_{number} = LPN/(\textit{number of pages per TTP}) \tag{6}$$

and the position of the Logical to Physical Page translation in a given TTP is determined as:

$$LP_{index} = LPN \bmod \textit{number of pages per TTP}. \tag{7}$$

If the logical page is located in the MTTP, then the physical address of the logical page can be accessed directly. If not, then the address of the STTP is read from the MTTP using the TTP_{number} as the lookup index and then the physical address read from the LP_{index} position in the corresponding STTP. By using direct memory reads, page address resolution lookups can be accomplished orders of magnitude faster in terms of energy and time and without having to use flash buffers or RAM to store translation information.

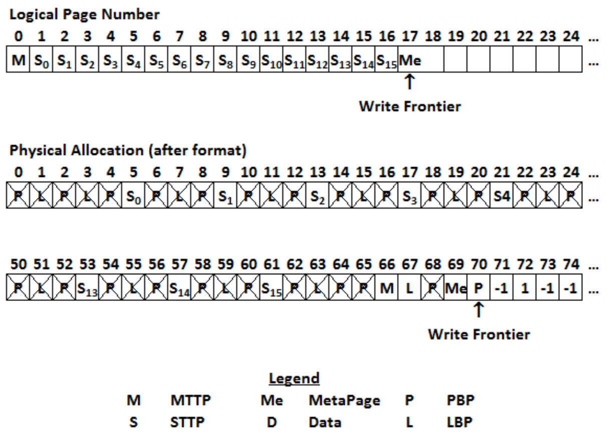


Fig. 1. Memory Allocation Overview

In Figure 1 is the structure of memory after first format. After the MTTP and STTPs is the MetaTable Page, which encodes information regarding the files on the system. Each file entry has an identifier, structure information, and logical page number of its first data page. File data are stored in Data Pages. Each MetaTable page stores multiple metadata records on files and multiple pages may be used if there are many files.

Two special pages are used to control the allocation and status of physical and logical pages in the system. The Physical Busy Page (PBP) encodes the status of each physical page in the system while the Logical Busy Page (LBP) encodes the status of each logical page. A single 528 byte page can be used to encode the status of all pages in the system as a bit vector. In the PBP, each byte segment maps to a physical 8 page block. If a page is free then the corresponding bit is set to 0; if it is busy the bit is set to 1. The same mapping applies to the LBP. Unlike other designs, our system does not record or care about the erase state of a page. This is due to the nature of how the Wear Leveler and Flash Translation Layer function.

All pages except the PBP and LBP contain an out-of-band data section. For a 528 byte page, 512 bytes are used for data while the remaining 16 bytes are used to store information. Each page has an identifier and reverse logical page pointer which identifies which logical page is assigned to a specific physical page. This information is used by the Wear Leveler. For Metatable Pages and Data pages, fields are present for the next logical page in the file as well as the number of records in a given page.

4.1. Frontier Advance Wear Leveling

Similar to [11], pages are allocated by a frontier sweeping through the memory in an increasing fashion which helps to spread erase/writes across the entire device and thus extend its life. Whenever a new page is allocated or an existing page is updated with new data, it is written to the page pointed to by the frontier. Pages located at the frontier are clean as they have been guaranteed to have been erased by the Wear Leveler.

Unlike other systems, there is no separate garbage collection process to reclaim dirty pages. Instead, a dynamic wear leveling and cleaning strategy is used called Frontier Advance Wear Leveling. It uses a greedy approach to keep a region in front of the write frontier continually clean and available for use.

In advance of the write frontier, the algorithm keeps a minimum of eight blocks (64 pages) free. As data is written to the device, the write frontier advances into the clear space, ensuring that no old data has been inadvertently overwritten.

Located at the eight block boundary in front of the write frontier is the sweep frontier. Blocks immediately in front of the sweep frontier will be swept clean based on their bit state in the PBP. Live or parked data pages will be copied to the write frontier as the sweep frontier advances through the page space. As live pages are moved, each page's reverse logical pointer is noted and stored along with the new physical address. Once all live blocks are moved, the reverse logical pointers are used to update the secondary table translation pages and physical busy page. To complete the transaction, the master table translation page is written as a keystone. If the transaction fails before completion, the blocks in front of the sweep frontier are undisturbed and accessible from the last known good keystone page. Upon a successful write of the master table translation page, the eight blocks in front of the sweep frontier are erased.

4.2. Inserting Records

When inserting a record in a page, the physical address of the logical page is determined from the MTTP and a STTP. This logical to physical address mapping costs two direct byte reads of two bytes each. The PBP is used to get new physical pages for the data page to be written out, PBP, and the MTTP as all three pages will be updated and cannot be overwritten. Figure 2 shows how pages are allocated. The pages are written out in the order: data page, PBP, and then the MTTP. In this way, the MTTP correctly points to the most up to date version of the logical data page. Due to the order of writing out the blocks, if a failure occurs before writing the MTTP the previous version of the MTTP still points to the previous data and will be used at restart. Only after the MTTP write has occurred will the file system reflect the changes. All MTTPs have an associated count such that on a restart the most current master block can be located. By controlling how and when blocks are erased in relation to the write frontier, the system guarantees that live data pages being updated will never be erased until the transaction is complete.

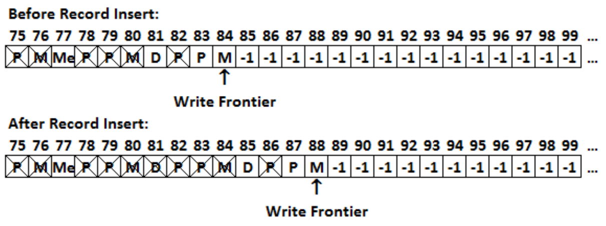


Fig. 2. Record Insert Example

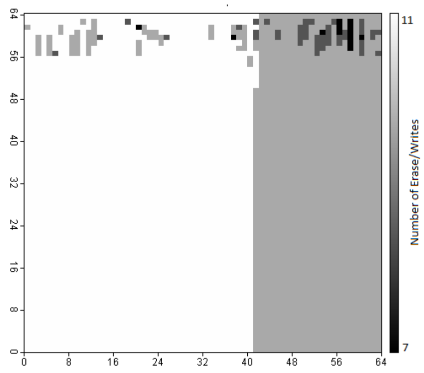
5. EXPERIMENTAL RESULTS

To profile the different aspects of this algorithm, a simulator was developed that allows for extensive visualization of page utilization and device timing and energy consumption. The experiments tested the basic operations of FLaReFS by creating a file and performing 10 consecutive 80 byte writes. Data was then read back from the file and the energy consumption and bus time determined. The results are presented in Table 1 and were then compared to the results presented by [11]. It is assumed that the energy results presented in [11] is for the flash chip only. In general, FlaReFS compares well in terms of energy efficiency as it exploits direct reads and does not transfer buffered data off the device unless it is needed. In terms of latency, it is on average twice as slow as other systems as all mapping structure is flash-resident, but that it is still reasonable when considering the time period of data sampling in environmental applications (which may be on the order of minutes to days). For systems that rely heavily on data reads, FlaReFS significantly outperforms the other systems due to its ability to exploit direct reads. It should be noted that the results presented for FlaReFS does not include energy and latency for wear leveling.

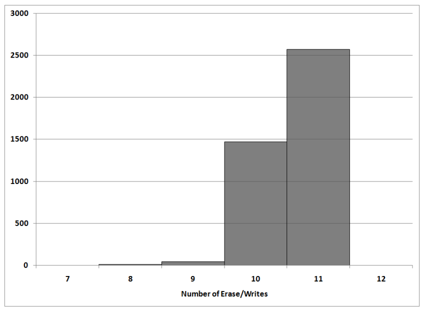
	FLaRe		Capsule		Matchbox	
	Energy (mJ)	Latency (ms)	Energy (mJ)	Latency (ms)	Energy (mJ)	Latency (ms)
Create	1.25	36.07	1.79	19.16	1.03	14.16
Write (80b x 10)	5.65	168.217	8.83	85.6	10.57	91.6
Open	n/a	n/a	0.0098	0.184	0.093	1.382
Read (80b x 10)	0.574	27	1.2	18.44	1.12	16.52
Total (c+w,o+r)	7.474	231.287	11.83	123.4	12.82	123.7
RAM footprint	134 bytes		1.5Kbytes		0.9Kbytes	

Table 1. Energy Consumption and Latency for FlaReFS, Capsule and Matchbox

The wear leveling policy presents extremely uniform wear across the device. A file was created with a record length of 10 bytes which is typical of what would be measured from a 12-bit device including a time stamp. To simulate a large collection period of 100 days where samples were taken every 15 minutes, 10,000 records were written to a single file. The results of the wear leveling policy is seen in Figure 3. After over 40,000 erase/write cycles data has been uniformly worn across the entire device with a very small variance.



(a) Wear Heat Map



(b) Erase Cycle Histogram

Fig. 3. Results of the Wear Leveling Policy

6. CONCLUSION

In this paper we present FlaReFS, a flash resident file system. Unlike other systems, all support structures for the file system are maintained in flash with only key pointers being maintained in RAM. The system is transactionally robust due to keystoning and has the ability to recover quickly on restart. Experimental evaluation has shown that the file system is suitable for use in long term monitoring applications. Due to the ability to support direct reads, the system is particularly suited to applications requiring extensive on device data reads and analysis. Additionally, the wear leveling strategy employed ensures uniform wear across the device which results in maximum lifetime. Future work will focus on the porting of the application to a transportable library.

7. REFERENCES

[1] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless Sensor Networks: A Survey," *Computer Networks*, vol. 38, no. 4, pp. 393 – 422, 2002.

[2] Hui Dai, Michael Neufeld, and Richard Han, "ELF: An Efficient Log-structured Flash File System for Micro Sensor Nodes," in *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, New York, NY, USA, 2004, pp. 176–187, ACM.

[3] Paolo Baronti, Prashant Pillai, Vince W.C. Chook, Stefano Chessa, Alberto Gotta, and Y. Fun Hu, "Wireless Sensor Networks: A Survey on the State of the Art and the 802.15.4 and ZigBee Standards," *Computer Communications*, vol. 30, no. 7, pp. 1655 – 1695, 2007, Wired/Wireless Internet Communications.

[4] IEEE Computer Society, "IEEE Std. 802.15.4-2007," August 2007, Available: http://standards.ieee.org/getieee802/download/802.15.4a-2007.pdf.

[5] A. K. Dwivedi, M. K. Tiwari, and O. P. Vyas, "Operating Systems for Tiny Networked Sensors: A Survey," *International Journal of Recent Trends in Engineering (IJRTE), Issue on Computer Science*, vol. 1, no. 2, pp. 152–157, May 2009.

[6] G. J. Pottie and W. J. Kaiser, "Wireless Integrated Network Sensors," *Commun. ACM*, vol. 43, pp. 51–58, May 2000.

[7] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister, "System Architecture Directions for Networked Sensors," *SIGPLAN Not.*, vol. 35, pp. 93–104, November 2000.

[8] Gaurav Mathur, Peter Desnoyers, Deepak Ganesan, and Prashant Shenoy, "Ultra-low Power Data Storage for Sensor Networks," in *Proceedings of the 5th International Conference on Information Processing in Sensor Networks*, New York, NY, USA, 2006, IPSN '06, pp. 374–381, ACM.

[9] Eran Gal and Sivan Toledo, "Algorithms and data structures for flash memories," *ACM Comput. Surv.*, vol. 37, no. 2, pp. 138–163, 2005.

[10] Eran Gal and Sivan Toledo, "A Transactional Flash File System for Microcontrollers," in *ATEC '05: Proceedings of the Annual Conference on USENIX Annual Technical Conference*, Berkeley, CA, USA, 2005, pp. 7–7, USENIX Association.

[11] Gaurav Mathur, Peter Desnoyers, Paul Chukiu, Deepak Ganesan, and Prashant Shenoy, "Ultra-low Power Data Storage for Sensor Networks," *ACM Trans. Sen. Netw.*, vol. 5, pp. 33:1–33:34, November 2009.

[12] Mendel Rosenblum and John K. Ousterhout, "The Design and Implementation of a Log-structured File System," *ACM Trans. Comput. Syst.*, vol. 10, pp. 26–52, February 1992.

[13] Margo Seltzer, Keith Bostic, Marshall Kirk Mckusick, and Carl Staelin, "An Implementation of a Log-structured File System for UNIX," in *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, Berkeley, CA, USA, 1993, pp. 3–3, USENIX Association.

[14] David Woodhouse, "JFFS: The Journaling Flash File System," in *Proceedings of the Ottawa Linux Symposium*, 2001.

[15] Seung-Ho Lim and Kyu-Ho Park, "An Efficient NAND Flash File System for Flash Memory Storage," *Computers, IEEE Transactions on*, vol. 55, no. 7, pp. 906 – 912, july 2006.

[16] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler, "The nesC Language: A Holistic Approach to Networked Embedded Systems," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, New York, NY, USA, 2003, PLDI '03, pp. 1–11, ACM.

[17] Demetrios Zeinalipour-Yazti, Song Lin, Vana Kalogeraki, Dimitrios Gunopulos, and Walid A. Najjar, "Microhash: An Efficient Index Structure for Flash-based Sensor Devices," in *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies - Volume 4*, Berkeley, CA, USA, 2005, pp. 3–3, USENIX Association.

[18] Aviad Zuck, Ohad Barzilay, and Sivan Toledo, "NANDFS: A Flexible Flash File System for RAM-constrained Systems," in *Proceedings of the seventh ACM international conference on Embedded software*, New York, NY, USA, 2009, EMSOFT '09, pp. 285–294, ACM.

[19] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda, "A flash-memory Based File System," in *Proceedings of the USENIX 1995 Technical Conference Proceedings*, Berkeley, CA, USA, 1995, TCON'95, pp. 13–13, USENIX Association.