Exploiting Non-Volatile RAM to Enhance Flash File System Performance

In Hwan Doh
Department of Computer Engineering
Hongik University
Seoul 121-791, Korea
ihdoh@cs.hongik.ac.kr

Jongmoo Choi
Division of Information and Computer Science
Dankook University
Seoul 140-714, Korea
choijm@dankook.ac.kr

Donghee Lee
School of Computer Science
University of Seoul
Seoul 130-743, Korea
dhl_express@uos.ac.kr

Sam H. Noh
School of Information and Computer Engineering
Hongik University
Seoul 121-791, Korea
samhnoh@hongik.ac.kr

ABSTRACT
Non-volatile RAM (NVRAM) such as PRAM (Phase-change RAM), FeRAM (Ferroelectric RAM), and MRAM (Magnetoresistive RAM) has characteristics of both non-volatile storage and random access memory (RAM). These forms of NVRAM are currently being developed by major semiconductor companies and are expected to be an everyday component in the near future. The advent of NVRAM may possibly bring about drastic changes to the system software landscape. In this work, we develop a new Flash memory based file system that exploits NVRAM in order to improve system performance. Specifically, we discuss the initial design and implementation of a file system that stores all metadata in NVRAM, while storing all file data in Flash memory. In so doing, we make two contributions in this work. First, we present a model that analyzes the amount of NVRAM that is needed for specific Flash memory storage capacity. Experimentally, we verify that this model represents the exact NVRAM usage in the realistic environment. Second, we present quantitative experimental results that show how much performance gains are possible by exploiting NVRAM. Compared to YAFFS, a popular Flash memory based file system, we show that this file system requires only minimal time for mounting and that the execution time improves by a maximum of 600% and an average of 437% for the realistic workloads that we considered.

Categories and Subject Descriptors
C.3 [**Computer Systems Organization**]: Special-Purpose And Application-Based Systems – *Real-time and embedded systems*; D.4.3 [**Operating Systems**]: File System Management – *Access methods, directory structures*;

EMSOFT'07, September 30-October 3, 2007, Salzburg, Austria.

General Terms
Design, Experimentation, Measurement, Performance

Keywords
Non-Volatile RAM, Flash memory, File system, Metadata, Experimental evaluation

1. INTRODUCTION
NVRAM (Non-volatile Random Access Memory) is a form of next generation memory that retains the access characteristics of conventional RAM such as DRAM and SRAM, and yet, retaining the non-volatile characteristics of secondary storage such as Flash memory or disks. NVRAM in its various forms, such as PRAM (Phase-change RAM), FeRAM (Ferroelectric RAM), and MRAM (Magnetoresistive RAM), are being developed by major semiconductor companies such as Texas Instruments, IBM, Samsung, Fujitsu, Motorola, etc [5, 16]. As semiconductor technology continue to make progress, we can anticipate NVRAM to become a common component of embedded systems as well as commodity computers. In fact, Samsung recently announced its PRAM development plans as shown in Figure 1 and is expected to ship 512Mbit PRAM commercial samples by the end of 2007 [21].

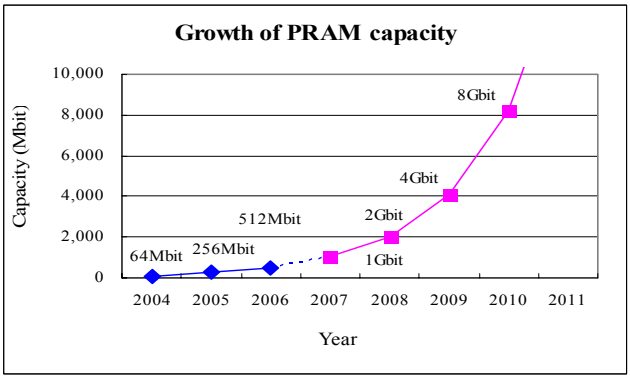


Figure 1: Anticipated trend for growth of PRAM capacity

Though not yet as prevalent as Flash memory, NVRAM has specific advantages over Flash memory. First of all, its access time is faster than Flash memory, especially for write access. While write access time for Flash memory is in the hundreds of microsecond range, writes in NVRAM is in the hundred nanosecond range [6, 17, 19]. Furthermore, the poor write access time in Flash memory can be exacerbated into the millisecond range if erasure operations must be invoked before the write. This may occur because physically overwriting within Flash memory is not possible. To overwrite existing data, the new data must generally be written to a different location and the old data invalidated. This incurs complications in managing old and new data [8, 12, 13]. The erasure operation is also the key factor that disfavors Flash memory compared to NVRAM. As erasures must be done in block units, copying of valid information within the block to be cleaned must occur [8]. This type of overhead for writes are not necessary with NVRAM as writes are done simply like DRAM. Erasures also bring about another critical limitation with Flash memory, that is, of endurance. Block erasures are physically limited to around 10,000 to 100,000 times for Flash memory, while there is no such limitation for NVRAM [6, 17, 19]. This makes NVRAM suitable for situations where writes frequently occur.

The question, then, is how to exploit these beneficial characteristics that NVRAM possesses, and to understand how much of it is needed to make such use possible. It seems the plan for now is simply to "replace NOR flash memory" [21]. Specifically, from a system software viewpoint, one of the options may be to use it as a write buffer cache [2]. In this paper, we take a different direction and explore the possibility of using NVRAM as a metadata store and examine what effects it has on the Flash file system. We choose to design a file system that puts all metadata of the file system in NVRAM as maintaining metadata safely, and doing it efficiently, is essential to keeping the file system consistent. All the file data, on the other hand, are stored in Flash memory. We name this new file system, the MiNV (Metadata in Non-Volatile ram) file system.

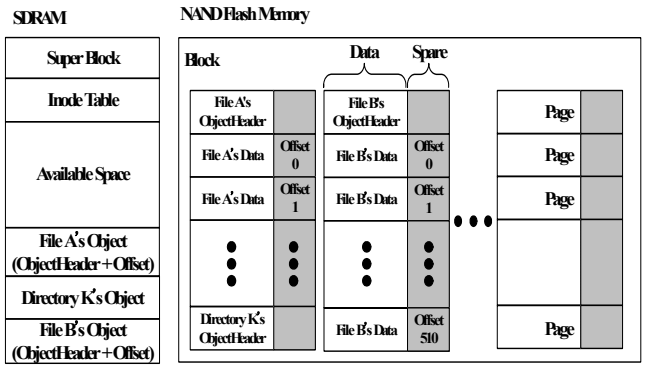
To assess the amount of NVRAM that would be required in this new design we derive a model of the NVRAM space requirement that is needed to store all the metadata. This model is validated via a real implementation in an embedded board. We also conduct other experiments and measure the performance difference between the MiNV file system and YAFFS, which is a typical Flash memory based file system [1]. Our measurements show that by using NVRAM as a metadata store the performance benefits are substantial. Mount time is reduce to the minimum and for realistic workloads that we considered the execution time is improved by as much as 600%.

The remainder of the paper is organized as follows. In the next section, we describe the design of the MiNV file system in detail. Then, in Section 3, we derive the NVRAM space requirement model and analyze the space requirement to deploy the MiNV file system in various settings. In Section 4, we present the experimental results. We first describe the experimental setup including the software and hardware platforms that we use. Then, we conduct various experiments and discuss their results. We briefly review some of the previous works related to this study in Section 5, and then, conclude with a summary and directions for future research in Section 6.

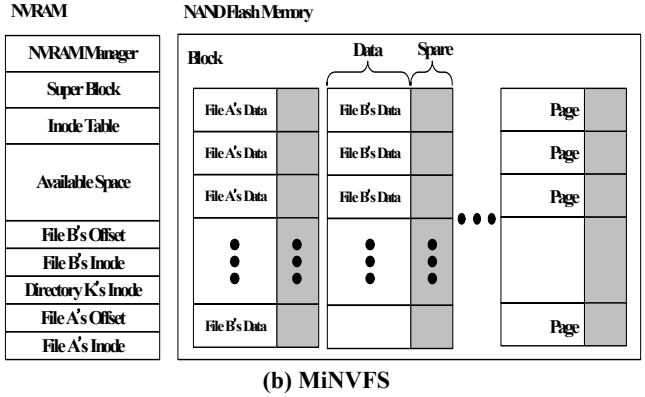
2. DESIGN OF THE MiNV FILE SYSTEM

In this section, we describe the design of the MiNV file system (MiNVFS) in detail. The overall structure of the design is similar to that of YAFFS, a popular file system designed for Flash memory currently used in operating systems such as Linux and WinCE [1]. The main difference in MiNVFS is that all its metadata are stored in NVRAM. Hence, we describe our design in contrast to that of YAFFS. Then, we explain how NVRAM is dynamically managed and how it maintains the corresponding metadata of the files and directories as their hierarchy changes.

Figure 2 illustrates the state of the metadata and file data as maintained in RAM, Flash memory, and NVRAM for YAFFS and MiNVFS. As depicted in Figure 2(a), YAFFS stores the metadata including the file offset information as well as the file data in Flash memory. YAFFS allocates a page from Flash memory for the `yaffs_ObjectHeader` structure, which contains the inode information of the file, for each file (and directory). The contents of the file data are stored in page units, and for each page the corresponding file offset information is recorded in the spare area. This information is used for recovery from system failure as well as at mount time to construct the data structures for metadata management in SDRAM that is needed for normal operation. The SDRAM depicted in Figure 2(a) is how it would look like after YAFFS is mounted.

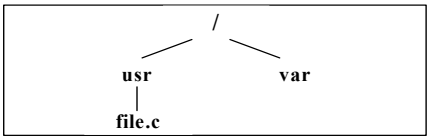


(a) YAFFS

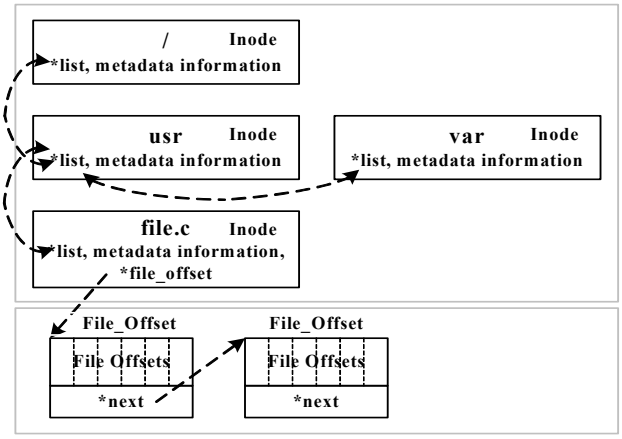


(b) MiNVFS

Figure 2: Metadata and file data maintained by (a) YAFFS and (b) MiNVFS



(a) Logical hierarchy of directories and a file



(b) Corresponding metadata structures in MiNVFS

Figure 3: Management of metadata in MiNVFS

Similarly to YAFFS, MiNVFS maintains data structures for metadata management for normal operation, but in NVRAM instead of SDRAM, as depicted in Figure 2(b). As can be noted from the figure, instead of the `yaffs_ObjectHeader` structure MiNVFS maintains a different structure referred to as the `NVRAM_Manager`, along with other structures such as `Inode` and `File_Offset`. Note also that all these structures are maintained once and for all in NVRAM and that these structures do not have to be reconstructed at mount time. Also, none of this information exists in Flash memory. Only file data is stored in Flash memory and the spare area is also no longer used to store metadata.

Let us now describe each of the data structures in more detail in the order they are laid out in NVRAM. At the top of the NVRAM, there is the `NVRAM_Manager` structure that contains information that is used to dynamically manage the NVRAM. Due to its simplicity, the BGET memory allocator is used for this purpose [22]. The `Superblock` structure comes next. This structure contains the current status information of each of the blocks, which consists of 32 pages, in Flash memory. In our implementation of MiNVFS, we associate 16 bytes for each block in Flash memory to maintain information such as its current allocation status and the number of currently allocated pages in the block. Within this information, we associate 2 bits for each page comprising the block to specify whether the page is available or not and whether it is clean or dirty. Next comes the `Inode_Table`. This is the hash table that links all existing `Inode` structures, which we describe shortly. By maintaining this table in NVRAM, MiNVFS has instant access to all the files immediately after booting.

Finally, the information for managing each of the files and directories are contained in the `Inode` structure. This structure is allocated from the bottom of the NVRAM. This structure contains information typically maintained in any file system such as ownership, access permission, creation time, modification time, access time of either a file or directory, and so on. When the structure is associated with a data file, the `Inode` structure contains a pointer to the `File_Offset` structure that contains the mapping information between the file data offset and the page (or possibly, block) address in Flash memory. As an example, Figure 3(a) shows a logical hierarchy of directories and a file and, in Figure 3(b), how this would be represented with the `Inode` and `File_Offset` metadata structures in MiNVFS. As depicted in Figure 3(a), there is the root directory, 2 subdirectories, and a file, and this is represented as an `Inode` structure for each of the three directories and the one file. These `Inode` structures are linked to each other to form the directory hierarchy. Different from the `Inode` structure for the directory, the `Inode` structure for the file has a pointer to the structure `File_Offset` that contains information mapping the file offset to the address of the page in Flash memory. The `File_Offset` structure has a fixed size that is equivalent to the size of the `Inode` structure to keep NVRAM management simple and minimize external fragmentation. By using a fixed size `File_Offset` structure, the size of a file that can be represented may be limited. To represent large files, we allow `File_Offset` structures to be linked to each other as shown in Figure 3(b).

3. NVRAM SPACE REQUIREMENT FOR MiNVFS

MiNVFS uses NVRAM as an extension of storage to maintain all the metadata of the file system. To assess the feasibility of deploying MiNVFS in real embedded systems, we analyze the amount of NVRAM that is needed to deploy MiNVFS in theory as well as in practice. We first present an analytical model that assesses the amount of NVRAM required for MiNVFS. Then based on the model, we analyze the amount of NVRAM needed for MiNVFS to be deployed in the real world by considering various Flash memory capacities, number of files, and sizes of files.

3.1 NVRAM Space Requirement Model

In MiNVFS, the data structures for the metadata that are to be stored in NVRAM consists of the `NVRAM_Manager`, `Superblock`, `Inode_Table`, `Inode`, and `File_Offset` structures. Space for the first three of these data structures are allocated and initialized in NVRAM when the Flash memory partition is formatted by MiNVFS. The space needed here is determined based on the capacity of the Flash memory partition that MiNVFS manages and is fixed once determined. On the other hand, space needed for structures `Inode` and `File_Offset` is dynamically allocated and freed as files are created and deleted. Space required by these data structures is all that is required of by MiNVFS. Hence, to analyze the NVRAM capacity requirement, we now need to model the space used by these structures.

Let us now model the space required by these data structures. Let the Flash memory size be S_{NAND}, the total number of files (including directories) managed by MiNVFS be n, with each file size denoted as S_k $(k = 0,1,2,\ldots,n-1)$. Files are allocated space in Flash memory in either block units or page units. We will denote this unit in which files are allocated C_{Offset}. Then, the relation $0 \le n \le \frac{S_{NAND}}{C_{Offset}}$ and $\sum_{k=0}^{n-1} S_k \le S_{NAND}$ holds.

Let us now represent the amount of NVRAM required by MiNVFS as function $R(S_{NAND}, n, S_k)$. Then, $R(S_{NAND}, n, S_k)$ (Equation (1)) can be represented as a sum of function $f(S_{NAND})$, which is the fixed space required by structures `NVRAM_Manager`, `Superblock`, and `Inode_Table` determined at initialization, function $g(n)$, the space needed for the `Inode` structures, and function $h(n, S_k)$, the space needed for the `File_Offset` structures. Each of the functions $f(S_{NAND})$, $g(n)$, and $h(n, S_k)$ can be represented as Equations (2)-(4) as follows and are described in detail below.

$$R(S_{NAND}, n, S_k) = f(S_{NAND}) + g(n) + h(n, S_k) \quad \text{---------- (1)}$$

$$f(S_{NAND}) = C_{NVRAMmm} + \left(\frac{S_{NAND}}{C_{BlockSize}} \times C_{BlockInfoSize}\right) + C_{InodeTable} + C_{InodeSize} \quad \text{--------- (2)}$$

$$g(n) = C_{InodeSize} \cdot n \quad \text{---------- (3)}$$

$$h(n, S_k) = \sum_{k=1}^{n-1} \left\lceil \frac{S_k}{C_{Offset}} \right\rceil \Big/ C_{NumOff} \times C_{OffSize} \quad \text{---------- (4)}$$

$C_{NVRAMmm}$ and $C_{InodeTable}$ in Equation (2) are constants that represents the fixed space required by the `NVRAM_Manager` and the `Inode_Table` structures, respectively. The second element of this equation is the space needed for the `Superblock` structure. As the `Superblock` maintains a fixed sized status information for each of the blocks, denoted as $C_{BlockInfoSize}$, the space requirement in NVRAM is the number of blocks, that is, the full Flash memory capacity, S_{NAND}, divided by the block size, $C_{BlockSize}$, of the Flash memory multiplied by constant $C_{BlockInfoSize}$. Finally, the last element of Equation (2) is $C_{InodeSize}$, which denotes the space needed for the one `Inode` structure for the root directory.

Equation (3) represents the space needed for the `Inode` structures. The number of files that are maintained by MiNVFS determines this space requirement, and hence, is simply the number of files, n, multiplied by the size of the `Inode` structure, $C_{InodeSize}$. Finally, Equation (4) represents the space needed for the `File_Offset` structures. The number of files and the size of each of these files determine this space. The larger a file, the more `File_Offset` structures are needed. We denote the size, in bytes, of the `File_Offset` structure as $C_{OffSize}$ and the number of mapping information, that is the number of page or block pointers it can hold, within the `File_Offset` structure as C_{NumOff}. Then, Equation (4) sums all the `File_Offset` structures consumed by each file multiplied by the size of the `File_Offset` structure. Table 1 summarizes the parameters used in the model.

3.2 Analysis Based on the Model

There are many parameters that are used to model the NVRAM space requirement for MiNVFS. These parameters can be categorized into three groups. First, there are parameters determined by the hardware platform. The full Flash memory capacity, S_{NAND}, and the block size, $C_{BlockSize}$, are such parameters. The second group of parameters is those that are determined by the specific implementation details and choices made for MiNVFS. $C_{NVRAMmm}$, $C_{BlockInfoSize}$, $C_{InodeTable}$, $C_{InodeSize}$, C_{Offset}, C_{NumOff}, and $C_{OffSize}$ are such parameters. Finally, the last group of parameters is determined by the workload. The number of files, *n*, and the size of each of these files, S_k, are such parameters.

Table 1: Summary of the parameters in the model

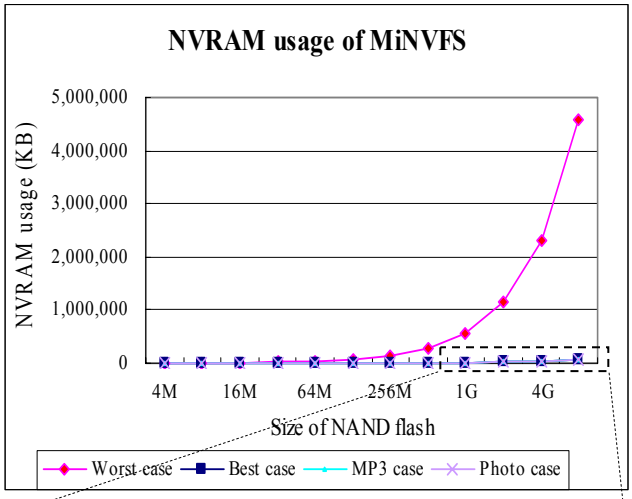
Parameters		Descriptions
Dependency	**Name**	
Hardware	S_{NAND}	Flash memory capacity
	$C_{BlockSize}$	Block size in Flash memory
Implementation	$C_{NVRAMmm}$	Size of `NVRAM_Manager` structure (bytes)
	$C_{BlockInfoSize}$	Size of status information for each block (bytes)
	$C_{InodeTable}$	Size of `Inode_Table` structure (bytes)
	$C_{InodeSize}$	Size of `Inode` structure (bytes)
	C_{Offset}	Size of file data allocation unit (bytes)
	C_{NumOff}	Number of pointers within `File_Offset` structure
	$C_{OffSize}$	Size of `File_Offset` structure (bytes)
Workload	n	Number of files
	S_k	Size of each file

Table 2: NVRAM space requirement in MiNVFS for 32MB NAND Flash memory (unit: KB)

		Number of Files								
		1	**4**	**16**	**64**	**256**	**1,024**	**4,096**	**16,384**	**65,536**
File Size	**0.5KB**	35	36	40	53	105	315	1,155	4,515	17,955
	2KB	35	36	40	53	105	315	1,155	4,515	-
	8KB	35	36	40	53	105	315	1,155	-	-
	32KB	36	37	42	61	140	455	-	-	-
	128KB	36	40	55	114	350	-	-	-	-
	512KB	40	53	107	324	-	-	-	-	-
	2MB	53	106	317	-	-	-	-	-	-
	8MB	105	316	-	-	-	-	-	-	-
	32MB	315	-	-	-	-	-	-	-	-

In the following analyses, we set the following parameters to constant values: $C_{BlockSize}$ and C_{Offset} to 16KB and 512B, respectively, as it is the block size and page size, respectively, of the Flash memory that we consider, and $C_{NVRAMmm}$, $C_{BlockInfoSize}$, $C_{InodeTable}$, $C_{InodeSize}$, C_{NumOff}, and $C_{OffSize}$ to values 28B, 16B, 3072B, 140B, 32, and 140B, respectively, as these are the numbers that were used in our specific implementation of MiNVFS. To consider various hardware platforms and workloads we vary the parameters S_{NAND}, n, and S_k, that is, the Flash memory size, the number of files, and the size of these files, respectively. However, owing to the fact that it is practically impossible to exhaustively consider all possible numbers of files and their sizes, we initially assume that all the files that are considered in a particular setting to be of the same size. We relax this assumption somewhat in later analyses.

Table 2 shows the NVRAM space requirement obtained from the model to operate MiNVFS in a system with 32MB NAND Flash memory. Each row represents the particular file size of interest and each column represents the number of files that exists in MiNVFS for that particular file size. The number corresponding to the particular row and column refers to the NVRAM space that is required for the number of files of that particular file size. For example, for 'File Size' row 2MB and 'Number of Files' column 16, we find the value 317. This means for MiNVFS to maintain sixteen 2MB sized files the NVRAM space requirement is 317KB.



	NAND Flash Size			
	1G	**2G**	**4G**	**8G**
Best case	9,987	19,971	39,939	79,875
MP3 case	10,022	20,041	40,079	80,155
Photo case	10,584	21,166	42,328	84,654

Figure 4: MiNVFS NVRAM usage for various Flash memory capacities

Note that the numbers in the diagonal represents the maximum NVRAM space requirement that is needed when the 32MB Flash memory space is fully utilized with files of the particular size. From these results, we can see that the NVRAM space requirement increases proportionally to the number of files, even though the total size of all the files stored in Flash memory is identically 32MB. In the best case, MiNVFS needs only 315KB of NVRAM when a single 32MB file fills up the 32MB Flash memory. In the worst case, however, the NVRAM requirement is around 18MB when MiNVFS fills Flash memory with 65,536 files that are 512B in size. Hereafter, we will refer to the case when we store the minimum number of files, thus requiring minimum NVRAM space as the "Best case". Conversely, the case when we store the most number of files and thus require the maximum NVRAM space will be referred to as the "Worst case".

Figure 4 shows the MiNVFS NVRAM usage when the underlying Flash memory capacity ranges from 4MB to 8GBs. Note that the latest MP3 players, such as iPod nano, are being shipped with a maximum of 8GB NAND Flash memory. The "Worst case" and "Best case" results, again, show the maximum and minimum NVRAM space requirements for each Flash memory capacity, respectively. In this figure, we also show graphs denoted as "MP3 case" and "Photo case". A typical MP3 file takes up approximately 4MBs, while a typical size of photo takes up approximately 330KBs. (Note that the iPod nano advertises that the 8GB version can hold 2000 tunes or 25,000 photos.) Hence, each of the "MP3 case" and the "Photo case" show the amount of NVRAM needed to deploy MiNVFS when the maximum number of 4MB and 330KB sized files, respectively, are stored in Flash memory of the specified capacity.

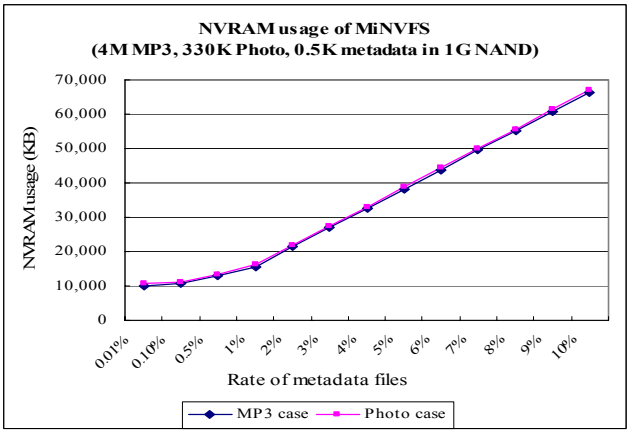


Figure 5: MiNVFS NVRAM usage for various rates of metadata files that consume Flash memory

The NVRAM space requirement increases linearly, for all four cases, as the Flash memory capacity increases. (Note that the x-axis is represented in log-scale.) We observe that the results for both the "MP3 case" and "Photo case" are very similar to that of the "Best case". For clarity, we show the results of the "Best case", "MP3 case", and "Photo case" for 1GB and above Flash memory capacity in table form in Figure 4. Considering 1GB Flash memory, NVRAM usage is approximately 10MBs for "MP3 case", "Photo case", and "Best case"; specifically, 10,022KB for the "MP3 case", 10,584KB for the "Photo case", and 9,987KB for the "Best case". In contrast, the NVRAM space required in the "Worst case" is more than half the Flash memory capacity, which may be impractical. However, in real-world products, such as MP3 players, mobile phones, and digital cameras, this "Worst case" scenario will rarely happen, if ever. Though the results for both the "MP3 case" and "Photo case" are promising, in real life, systems may retain metadata files along with the content files such as music and digital image files. For example, it is possible that with each music file, information such as the performer or composer may be stored in a separate play list or database metadata file. To consider this scenario, we examine the NVRAM space requirement when there is a mix of content and metadata files. We assume that each content file is of 4MBs for MP3 files, 330KBs for photo files and that each metadata file is of 0.5KBs. Figure 5 shows the NVRAM usage when MiNVFS manages a device with 1GB Flash memory capacity for various rates of content and metadata files. The *x*-axis represents the proportion of the whole Flash memory capacity that is consumed for all the metadata files. For example, consider the somewhat realistic case where metadata files occupy 1% of the Flash memory capacity and the 4MB content files take up the rest. With 1GB Flash memory, this means that we are storing approximately 20,000 metadata files along with 250 content files. For the 330KB photo content files and metadata files, this is like storing 20,000 metadata files along with around 3,000 content files. Our results show that in this case, we require 15,667KB and 16,223KB of NVRAM space for the 4MB content files and the 330KB content files, respectively. In the common case, we believe that the number of metadata files will not exceed a few hundred. Hence, the results in Figure 5 show that MiNVFS may be safely deployed with 10-20MBs of NVRAM for embedded consumer products of today.

4. PERFORMANCE EVALUATION

We implement MiNVFS in Linux and evaluate its performance by deploying it in a real embedded system board. In this section, first, we describe briefly the implementation and the experimental setup. Then, we present results of the performance evaluation conducted on this setup.

4.1 Experimental Setup

We implement MiNVFS in Linux 2.4.x according to the design that we described in Section 2. The Linux kernel loads MiNVFS as a kernel module implemented in between the VFS (Virtual File System) and MTD (Memory Technology Device) layers. Our implementation includes the file system formatting tool and supports all the common basic file system operations.

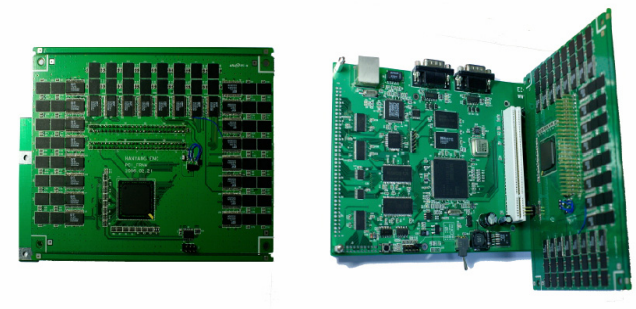
Figure 6 shows the actual NVRAM daughter board (Figure 6(a)) and the embedded development board with the NVRAM daughter board attached to the motherboard through the PCI interface (Figure 6(b)). The NVRAM daughter board was developed in-house and has 12MB of FeRAM. The NVRAM appears in physical memory address space and can be accessed via memory mapped addressing. The embedded development board that we use is the EZ-M28 board, which supports PCI interfaces, developed by Falinux [7]. The EZ-M28 board has a S3C2800 processor, which is based on the ARM920T, 32MB SDRAM, and 64MB NAND Flash memory.

4.2 Performance Evaluation Results

In this section, we present the experimental results. We first present results that validate the correctness of the NVRAM space requirement model presented in Section 3. Then, quantitative comparisons of experimental results when using MiNVFS and YAFFS are presented.

4.2.1 Correctness of the Model

In Section 3, we presented a model calculating the NVRAM space used by MiNVFS. In this subsection, we validate the model using the experimental setup. We use 32MBs of the 64MB Flash memory available on the experimental board and generate as many equal sized files as possible, the sizes ranging from 1KB to 32MB. (Since we have only 12MB of FeRAM in our experimental setup, we were not able to verify the results for files of size 0.5KB.)



(a) 12MB FeRAM board **(b) EZ-M28 Embedded board**

Figure 6: System environment for the experiments

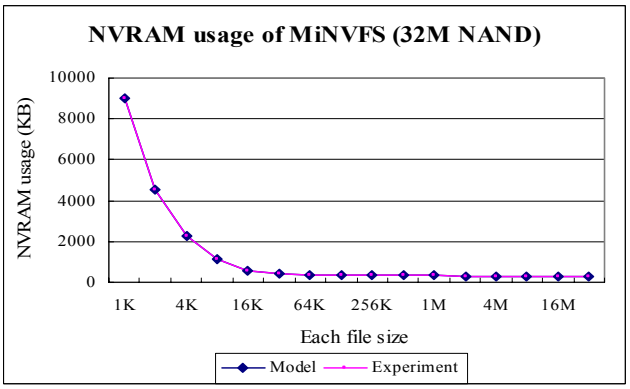


Figure 7: NVRAM space requirement as estimated by model and as measured through experiments

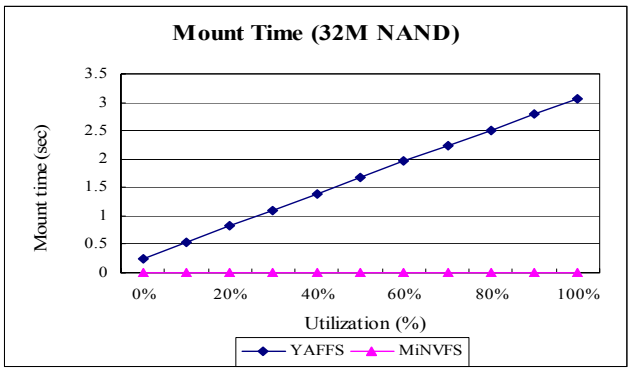


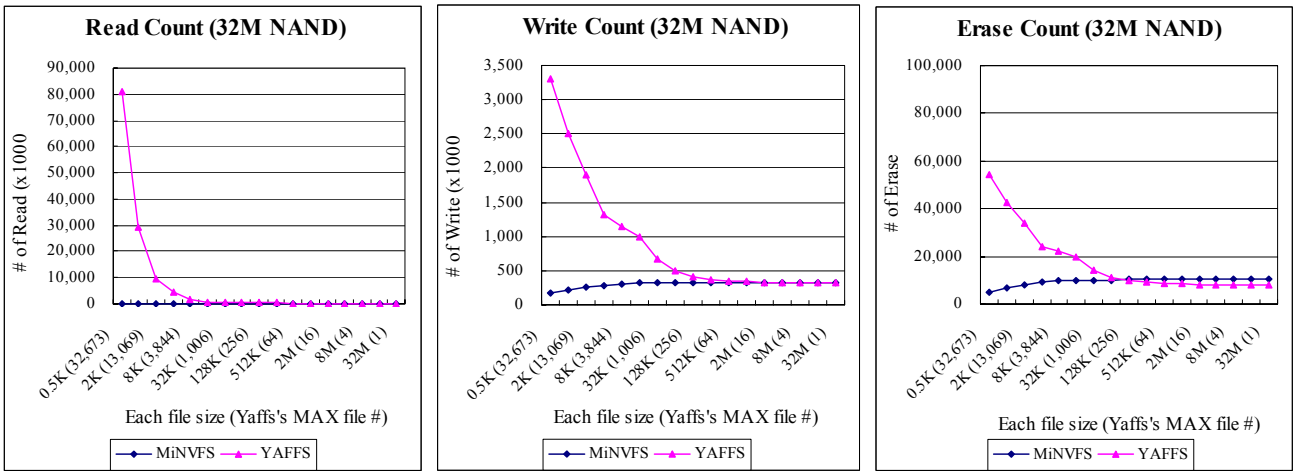
Figure 8: Mount time for YAFFS and MiNVFS

Figure 7 shows both the NVRAM space requirement calculated through the model, denoted "Model", and the actual maximum amount of NVRAM that was used that was measured through the experiments, denoted "Experiment". In fact, the two lines overlap each other and cannot be distinguished implying that the model is accurate.

4.2.2 Mount Time

In the experiments presented in this subsection, we compare the mount time for YAFFS and MiNVFS. With YAFFS, the metadata of the file system is scattered in Flash memory space. The scattered metadata are collected and organized in SDRAM for use, as explained in Section 2. This implies that as the Flash memory size grows, mount time will proportionally increase. In MiNVFS, however, all metadata is in NVRAM; hence, mount time should be constant and almost instantaneous. This is verified through the experiments.

Figure 8 shows the mount time for YAFFS and MiNVFS. The *x*-axis of this graph is the utilization of Flash memory ranging from 0% to 100%. We see that the mount time for MiNVFS is constant at 67 microseconds for all cases. For YAFFS, mount time increases linearly with utilization peaking at slightly over 3 seconds when Flash memory is fully utilized. The reason behind the linear increase is that the Flash memory space that YAFFS has to scan grows. Moreover, the linear increase observed is optimistic and it is a result of the way the experiments were conducted. For each measurement, we erased all the blocks and wrote out files to consume the designated percentage of Flash



(a) Read operation count **(b) Write operation count** **(a) Erase operation count**

Figure 9: Number of read, write, and erase operations performed in Flash memory for the synthetic workload

memory. By so doing, YAFFS actually used only those blocks at the lower end of the Flash memory space leaving all the other blocks clean, because YAFFS consumes blocks, and the pages within the blocks, in sequential order. Hence, when scanning the blocks to obtain the metadata, blocks with a clean first page are all skipped. This results in considerable savings in scanning time. In reality, however, as files are manipulated, this optimal layout will generally not occur increasing the scan time, hence the mount time.

Note that as Flash memory capacity grows, YAFFS type of scanning for metadata when mounting can become a significant overhead. Today's handheld devices commonly have gigabyte capacity Flash memory, and its capacity is growing at a fast pace. With such large capacity Flash memory devices mounting could take 10's to 100's of seconds making such devices intolerable. Additionally, if a YAFFS type file system is employed as the root file system, its long mount time could cause significant delays during system boot time. In this respect, NVRAM hardware and file systems that make use of this hardware such as MiNVFS, which incurs extremely short mount time, can be quite an attractive alternative.

4.2.3 Performance Evaluation with Synthetic Workloads

In this subsection, we compare the performance results of MiNVFS and YAFFS using a synthetic workload. The workload comprises sequentially creating files and then deleting these files over and over five consecutive times. No updates on these files occur. The files created are all of the same particular size for each experiment. Table 3 shows the size of the file and the number of files that are created for each particular file size as used in the experiments. The number of files is determined by the size of the data file and the extra space that is needed for the metadata in YAFFS to fill up the 32MB Flash memory space. For example, for file size of 0.5KB, we can only create 32,768 files as the rest of the Flash memory space is needed to store the metadata information of these files. Note also that for particular file sizes, the number presented in Table 3 is not exact. For example, for a 32MB file, this file alone will consume all of the Flash memory space, not leaving any space for the metadata. Hence, we reduce the actual file size used in the experiments just enough to provide space for the metadata. However, since the file size is close to 32MBs, we simply represent it as such, for simplicity. With this experimental setup MiNVFS will have extra space in Flash memory that is not being used. Also, it may seem like MiNVFS is being given an unfair advantage with the extra NVRAM resource. However, since MiNVFS does not need SDRAM space for the metadata as in YAFFS, NVRAM is, in fact, replacing SDRAM resource, not adding extra resources, plus it saves on the Flash memory resource.

Figure 9 shows the number of read, write, and erase operations performed in Flash memory for the synthetic workload. The *x*-axis for the three graphs is the file sizes ranging from 0.5KB to 32MB, in log scale. The value within the parentheses is the number of files created and is the same as those listed in the second row of Table 3. Figure 9(a) shows that MiNVFS has no read operation regardless of the number of files in contrast to that of YAFFS. Although the synthetic workload has no read request for files, YAFFS executes Flash memory page read operations proportional to the number of files. The reason behind this is that YAFFS has to execute read operations in order to update the metadata and also to copy pages to other blocks during garbage collection that occasionally occurs. (Garbage collection occurs because of the way YAFFS creates files. It first creates and writes metadata for the file to be created, then writes out the file data, and then, creates and writes a newly updated metadata, and then, invalidates the old metadata, all in Flash memory. This incurs 3 extra writes to manipulate metadata and consumes two pages for the metadata. This incurs garbage collection, as eventually clean pages will run out in Flash memory.) MiNVFS, on the other hand, writes all metadata on NVRAM and does not incur any garbage collection. Hence, we do not observe any read operations in Flash memory with MiNVFS.

Table 3: Number of files created for each particular file size

	Each file size (Bytes)								
	0.5K	**2K**	**8K**	**32K**	**128K**	**512K**	**2M**	**8M**	**32M**
# of files	32,673	13,069	3,844	1,006	256	64	16	4	1

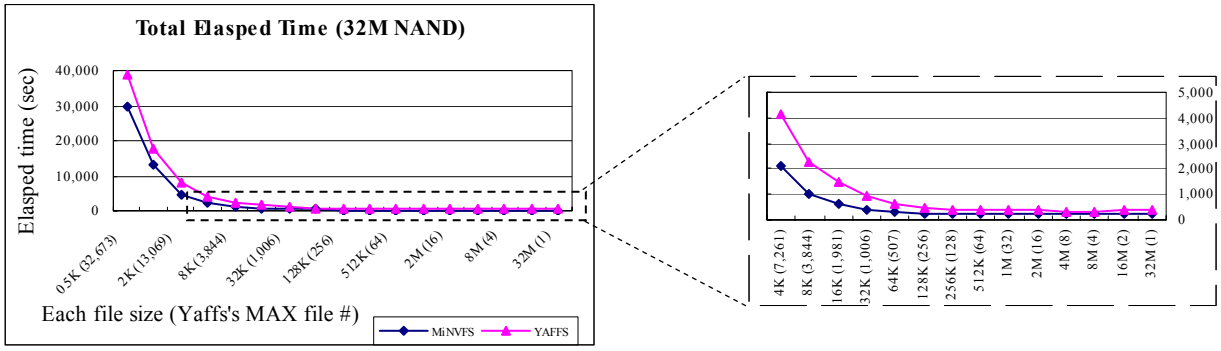


Figure 10: Total time elapsed executing the synthetic workload

Figure 9(b) shows the number of page write operations incurred for MiNVFS and YAFFS. For all cases, MiNVFS incurs less writes than YAFFS. Since both file systems are being driven by the exact same workload, they incur the same number of write operations for the file data writes. However, as described in the previous paragraph, YAFFS incurs 3 additional write operations to manage the metadata when creating a file, with additional writes being incurred during garbage collection. With YAFFS, hence, the number of write operations increases proportionally to the number of files that are created.

The number of erase operations incurred by MiNVFS and YAFFS is shown in Figure 9(c). From this graph, we again observe that the number of erase operations increases as the number of files increase. YAFFS has many more garbage blocks that need to be collected because of the frequently updated metadata that pollutes the pages in Flash memory.

A peculiar observation in Figure 9(c) is that the number of erase operations incurred for MiNVFS is greater than that of YAFFS when the file is 128KB and greater. The reason behind this is that while MiNVFS actually cleans, that is, erases blocks when files are deleted YAFFS tends to simply mark the file as deleted and delays the actual block erase operations for as long as possible. The consequence of this is that, since the last set of operations conducted in our experiments is to delete all the files, after each experiment is over, YAFFS leaves behind around 2,000 blocks that still needs to be erased, while MiNVFS leaves none such blocks behind. Hence, even though the results reported in Figure 9(c) appear to favor YAFFS, if we consider the blocks that still need to be erased before reuse, the erase count is actually almost identical.

Finally, Figure 10 shows how the difference in read, write, and erase operation counts are reflected in the execution time. Figure 10(a) shows the total time elapsed executing the synthetic workload. As can be observed from the figure MiNVFS takes shorter time than YAFFS in all cases. MiNVFS outperforms YAFFS by a maximum of 156% and an average of 81%. Figure 10(b) gives a closer look at the results for file sizes 4KB to 32MB. It is noted that MiNVFS has shorter elapsed time than YAFFS for file sizes greater or equal to 128KB even though MiNVFS has a larger erase count for this file size range. This is due to the implementation details regarding YAFFS and MiNVFS, with MiNVFS being implemented much more efficiently. We do not go into the details of this, as this is more an issue of code optimization rather than file system design.

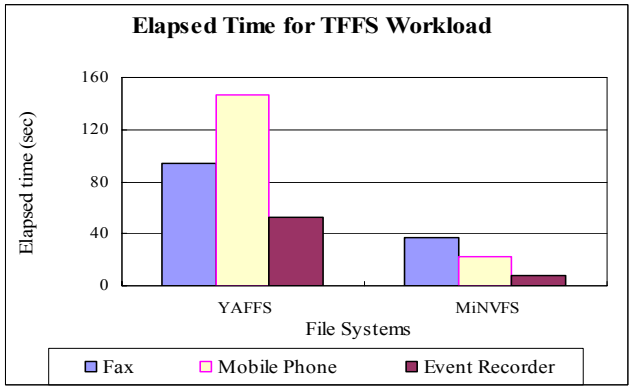


Figure 11: Total elapsed time executing the TTFS workload

4.2.4 Performance Evaluation with Realistic Workloads

In this subsection, we evaluate MiNVFS for more realistic workloads. To the best of our knowledge, there is no de facto "realistic" file system workload that reflects the characteristics of embedded systems using Flash memory that is accepted by the research community. Hence, for this study, we reproduce the TFFS benchmark program from the workload scenario used to evaluate the performance of the TFFS file system proposed by Gal and Toledo [8].

The TFFS benchmark generates the Fax, Mobile Phone, and Event Recorder file system workloads. The Fax workload can be said to represent the operations of not just the Fax machine, but also of devices that manage relatively large files such as answering machines and music players. The Mobile Phone workload represents the operations of devices that manage small files such as beepers as well as mobile phones. The Event Recorder workload represents the operations of devices that create record files and update logs in the record files.

Figure 11 shows the total elapsed time for both MiNVFS and YAFFS for the three workloads. Compared to YAFFS, MiNVFS outperforms YAFFS by 152% for the Fax workload, 559% for Mobile Phone workload, and 600% for the Event Recorder workload, the average of these three being 437%. These results imply that MiNVFS is especially efficient for workloads where file sizes are relatively small and where these files are frequently updated.

5. RELATED WORKS

Considerable research has already been conducted on the issues related to NVRAM and/or file systems for Flash memory. We summarize some of these works in this section.

File systems for storage systems based on Flash memory has recently been a popular area of research. The research conducted in this area can be divided into two directions. One is on efficiently supporting legacy file systems on Flash memory devices. Much of the work in this direction has been in developing an efficient software layer called the Flash Translation Layer (FTL), which is a layer that makes the Flash memory device look like a disk device. Hence, the file system need not be aware of the difference in the storage media. The Flash memory driver proposed by Kawaguchi et al. is one of the first FTLs to be proposed [11]. A few improvements have been suggested since then. Specifically, the FTL proposed by Kim et al. provides high performance by using log blocks to support hybrid mapping, which combines page-based mapping and block-based mapping [12]. More recently, Lee et al. propose a new scheme called FAST that improves on the log block scheme by making more efficient use of the log blocks [13].

The other direction of research on file systems for Flash memory is the development of file systems that are aware of the underlying Flash storage device. A majority of these file systems are based on the design philosophy of the Log-structured File System (LFS), but adapted to support NAND or NOR Flash memory [18]. Well known among these are YAFFS and JFFS2 [1, 23]. These file systems, however, have a significant drawback in that they require long mount times as they generally have to scan the whole Flash memory space. To avoid this scanning process, Yim et al. suggest that they copy the metadata information maintained in RAM to a reserved portion of Flash memory before unmounting the file system [26]. On a similar issue, Wu et al. suggests logging additional information for future fast mount as well as for recovery after failure [24]. Both these schemes require additional Flash memory and additional operations for fast mounting. Gal and Toledo also provide a nice survey of popular file systems and works related to Flash memory [9].

Research on the issue related to NVRAM is not new. In the early 1990s, research on making use of NVRAM in general purpose computer systems were conducted. However, NVRAM being considered at the time was mostly battery-backed RAM. Specifically, Baker et al. showed that write traffic can be significantly reduced with the help of NVRAM in a network file system environment [2]. Chen et al. proposed the Rio file cache that supports fault tolerance in file systems without degrading performance by using NVRAM [4, 14].

There are also previous studies that consider using NVRAM as an extension of storage for file systems and thus, maintain metadata in this part of storage. These studies are closely related to our work. The HeRMES file system proposed by Miller et al. makes use of MRAM to store metadata, while storing file data in disk [15]. Another file system, MRAMFS, uses a similar approach as HeRMES, but it utilizes compression for the metadata to conserve NVRAM space [20]. Conquest is another file system developed with NVRAM in mind [25]. In these works NVRAM is considered in conjunction with the hard disk drive, which serves as the main storage, whereas in MiNVFS we are considering an embedded platform with the Flash memory media being the main storage.

There are also studies on NVRAM based file systems, in particular, PRAMFS and the NEB file system, that consider NVRAM as the main storage [3, 10]. Our study differs from these studies in that we consider NVRAM as a supplement to the main Flash memory storage, whereas these studies consider NVRAM to be the main storage.

6. SUMMARY AND FUTURE WORK

NVRAM technology is becoming a reality. NVRAM of considerable capacity is soon to become available for use in embedded systems. In this study, we presented a design and implementation of the MiNV file system that exploits NVRAM to store all of the file system's metadata. The design itself is in line with the YAFFS file system. We model and analyze the NVRAM space required to deploy the MiNV file system. For applications of today such as MP3 players or digital image retainers, the amount of NVRAM required is in the 10's of megabytes.

We conduct a series of experiments on a real embedded board that has 12MBs of FeRAM, a form of NVRAM. The performance results show that mount time is drastically reduced as Flash memory need not be scanned. For synthetic and realistic workloads, the performance improvement is significant with the MiNV file system execution time improving as much as 600% compared to YAFFS for the workloads that we considered.

There is still much work that needs to be done. The use of NVRAM will certainly have an effect on the energy consumption of the system. As Flash memory operations are reduced, overall energy consumption will likely be reduced. Also, as writes to Flash memory is reduced, the issue of wear-leveling in Flash memory may be simplified. These are some immediate issues that we are currently taking a look at. Also, further optimizations in our design and implementation are also being contemplated. Finally, how the manner in which NVRAM was used in this study can benefit embedded systems that employ legacy file systems should be considered as well.

7. ACKNOWLEDGMENTS

This work was partly supported by the IT R&D program of MIC/IITA [2006-S-040-01, Development of Flash Memory-based Embedded Multimedia Software]] and supported in part by grant No. R01-2004-000-10188-0 from the Basic Research Program of the Korea Science & Engineering Foundation.

8. REFERENCES

[1] Aleph One Company, YAFFS (Yet Another Flash File System), http://www.aleph1.co.uk/yaffs/yaffs.html/.

[2] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer, Non-Volatile Memory for Fast, Reliable File System, In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pp. 10-22, Oct. 1992.

[3] CE Linux Public, PramFs, http://www.celinuxforum.org/CelfPubWiki/PramFs/.

[4] P. M. Chen, W. T. Ng, S. Chandra, and D. E. Lowell, The Rio File Cache: Surviving Operating System Crashes, In *Proceedings of the 7th International Conference on*

Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII), pp. 74-83, Oct. 1996.

[5] Freescale Semiconductor, http://www.freescale.com/.

[6] Freescale Semiconductor, MR2A16A MRAM Datasheet, http://www.freescale.com/files/microcontrollers/doc/data_sheet/MR2A16A.pdf.

[7] FALINUX, EZ-M28, http://falinux.com/zproducts/ez-m28.php/.

[8] E. Gal and S. Toledo, A Transactional Flash File System for Microcontrollers, In *Proceedings of the USENIX Annual Technical Conference (USENIX 2005)*, pp. 89-104, Apr. 2005.

[9] E. Gal and S. Toledo, Algorithms and Data Structures for Flash Memories, *ACM Computing Surveys (CSUR), 37,* 2 (Jun. 2005), 138-163.

[10] C. Hyun, S. Baek, S. Ahn, J. Choi, and D. Lee, Experimental Evaluation of an Extent-based File System for Nonvolatile-RAM, In *Proceedings of the 2nd International Workshop on Software Support for Portable Storage (IWSSPS 2006)*, pp. 18-24, Oct. 2006.

[11] A. Kawaguchi, S. Nishioka, and H. Motoda, A Flash-Memory Based File System, In *Proceedings of the 1995 USENIX Winter 1995 Technical Conference*, pp. 155-164, Jan. 1995.

[12] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho, A Space-efficient Flash Translation Layer for CompactFlash Systems, *IEEE Transactions on Consumer Electronics, 28,* 2 (May 2002), 366-375.

[13] S. Lee, D. Park, T. Chung, D. Lee, S. Park, and H. Song, A Log Buffer based Flash Translation Layer using Fully Associative Sector Translation, To appear in *ACM Transactions on Embedded Computer Systems*.

[14] D. E. Lowell and P. M. Chen, Free Transactions with Rio Vista, In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI 2000)*, pp. 119-134, Oct. 2000.

[15] E. L. Miller, S. A. Brandt, and D. D. E. Long, HeRMES: High Performance Reliable MRAM-Enabled Storage, In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pp. 83-87, 2001.

[16] Ramtron International – Nonvolatile Memory, Integrated Memory and Microcontrollers, http://www.ramtron.com/.

[17] Ramtron International, FM24C512 FRAM Datasheet, http://www.ramtron.com/lib/literature/datasheets/FM24C512ds_r1.0.pdf.

[18] M. Rosenblum and J. K. Ousterhout, The design and implementation of a log-structured file system, *ACM Transactions on Computer Systems, 10,* 1 (Feb. 1992), 26-52.

[19] Samsung Electronics, NAND Flash Memory Datasheet, http://www.samsung.com/Products/Semiconductor/NANDFlash/SLC_LargeBlock/64Gbit/K9NCG08U5M/ds_k9xxg08uxm_rev10.pdf.

[20] A. B. Szczurowska, MRAM-preliminary analysis for file system design, Master's thesis, University of California, Santa Cruz, Mar. 2002.

[21] Tech-On News, http://techon.nikkeibp.co.jp/english/NEWS_EN/20070226/128173/.

[22] The BGET Memory Allocator, http://www.fourmilab.ch/bget/.

[23] D. Woodhouse, JFFS: The Journaling Flash File System, Ottawa Linux Symposium, 2001.

[24] C. H. Wu, T. W. Kuo and L. P. Chang, Efficient Initialization and Crash Recovery for Log-based File Systems over Flash Memory, In *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC 2006)*, pp. 896-900, Apr. 2006.

[25] A. A. Wang, P. Reiher, G. J. Popek, and G. H. Kuenning, Conquest: Better Performance Through a Disk/Persistent-RAM Hybrid File System, In *Proceedings of the USENIX Annual Technical Conference (USENIX 2002)*, pp. 15-28, Jun. 2002.

[26] K. S. Yim, J. Kim, and K. Koh, A Fast Start-Up Technique for Flash Memory Based Computing Systems, In *Proceedings of the 2005 ACM Symposium on Applied Computing (SAC 2005)*, pp. 843-849, Mar. 2005.