FASS : A Flash-Aware Swap System

Dawoon Jung[*], Jin-Soo Kim[†], Seon-Yeong Park[‡], Jeong-Uk Kang[§] and Joonwon Lee[¶]
Division of Computer Science
Korea Advanced Institute of Science and Technology
373-1 Guseongdong, Yuseonggu, Daejeon 305-701, Korea
{dwjung[*], parksy[‡], ux[§]}@camars.kaist.ac.kr, {jinsoo[†], joon[¶]}@cs.kaist.ac.kr

Abstract

Laptop computers and tablet PCs currently exploit swap system with their second storage media as a cost effective solution to extend limited memory space. The rapidly evolving flash memory technology starts to replace the magnetic disks of these computers by flash memory due to its advantageous characteristics such as energy efficiency and mechanical shock resistance. Thus, we can imagine that the swap system running over flash memory will show up.

However, since the contents of flash memory cannot be overwritten before being erased, we need a flash translation layer (FTL) to use flash memory as a disk transparently. Although FTLs help to deploy flash memory-based storage easily, the kernel and FTL sometimes make bad decisions because FTL cannot access kernel-level information and the kernel is not aware of flash memory states.

In this paper, we mainly focus on the swap system running over flash memory. We discuss design and implementation of flash-aware swap system, called FASS, where the kernel manages flash memory-based swap space directly without FTL. This approach can utilize kernel information and flash memory states to optimize the system. On average, we show that our FASS reduces the number of flash read and write operations by 61% and 41.5%.

1 Introduction

Flash memory recently has become popular storage media of mobile systems due to its advantageous features such as non-volatility, mechanical shock resistance, and low power consumption. There are two major types of flash memory. One is NOR flash memory and the other is NAND flash memory. NOR flash memory is used in place of ROM (Read Only Memory) to store and execute application code [8]. NAND flash memory is usually employed as high capacity storage media since it provides high density with low cost.

As other semiconductor memory devices such as SRAMs and DRAMs, the capacity of NAND flash memory has been increased quite dramatically by aggressive scaling of the memory cell transistor and architectural innovations. Now it is easy to find NAND flash memory cards with more than 1GBytes of storage capacities in the market. Moore's law tells us that the exponential growth in the flash memory capacity will continue for a considerable time. Hence, we can expect that there will appear laptop computers and tablet PCs in the near future, which are equipped with tens of GBytes of NAND flash memory-based secondary storages instead of hard disks. M-Systems is already shipping NAND flash memory-based storages called FFDs (Fast Flash Disks), whose capacity ranges from 1GBytes to 128GBytes [6]. Replacing hard disks with NAND flash memory brings advantages in terms of size, weight, reliability, and energy use.

For decades, traditional operating systems have performed various optimizations for hard disks, assuming that hard disks are used as a backing storage for file systems and swap systems. Unfortunately, however, NAND flash memory shows very different characteristics compared to magnetic disks. Most notably, there is no seek time in NAND flash memory, while it has an access constraint that data cannot be overwritten before being erased. Therefore, it is necessary to revisit various operating system policies and mechanisms, and to optimize them for NAND flash memory-based secondary storages.

The main focus of this paper is the optimization of virtual memory system, especially when the swap space is created on NAND flash memory. Although there are several previous work on flash-aware file systems such as YAFFS [1] and JFFS2 [12], little attention has been paid to flash-aware swap systems. Almost every modern desktop operating system makes use of additional swap space in order to supplement the limited physical memory space. Moreover, recently it is shown that the page-level swapping with NAND flash memory can be an energy and cost efficient solution for ever-increasing memory requirement in mobile embedded applications running on cellular phones or PDAs [7]. Therefore, devising an efficient flash-aware swap management scheme is as important as developing a flash-aware file system.

One of common practices used for flash memory-based secondary storages is to employ a thin software layer called FTL (Flash Translation Layer). Since FTL emulates the conventional hard disks transparently, most systems do not require any modification to the kernel in order to build file systems or swap space on top of flash memory-based secondary storages. While this layered approach eases the deployment of flash memory-based storages, the kernel or FTL can sometimes make bad decisions because FTL has no access to kernel-level information and the kernel is not aware of flash memory states. (Details will be discussed in section 2.4.)

In this paper, we propose a novel flash-aware swap system called FASS (Flash-Aware Swap System), where the kernel manages NAND flash memory-based swap space directly without the use of an intermediate layer such as FTL. We have implemented a FASS prototype based on the Linux kernel and have shown that FASS is very effective in reducing the number of flash read/write/erase operations.

The rest of the paper is organized as follows. Section 2 describes the background and the motivation of our work. Section 3 presents the design and implementation of FASS. The performance of FASS is evaluated in section 4. Finally, section 5 concludes the paper.

2 Background and Motivation

2.1 NAND Flash Memory

A NAND flash memory consists of a fixed number of blocks, where each block has 32 pages and each page consists of 512bytes main data and 16bytes spare data [10]. A page is a basic unit of the read/write operation, while the erase operation is performed on a block basis. In flash memory, once a page is written, it should be erased before the subsequent write operation is performed on the same page. This characteristic is sometimes called erase-before-write.

Unlike hard disks or other semiconductor devices such as SRAMs and DRAMs, the write operation requires relatively long latency compared to the read operation. As the write operation usually accompanies the erase operation, the operational latency becomes even longer. Another limitation of NAND flash memory is that the number of updating a block is limited to about 100,000 times. Thus, the number of erase operation should be minimized to lengthen the lifetime of NAND flash memory.

Recently, a new type of NAND architecture called *large block NAND* flash has been introduced to provide a higher capacity [10]. In the large block NAND flash, a page in a single memory array is 2112bytes (2048bytes main data + 64bytes spare data) and a block consists of 64 pages resulting in a block size that is 8 times larger. Note that the large block NAND flash has another constraint in programming the flash memory; within a block, the pages must be programmed consecutively from page 0 to page 63. Random page address programming is strictly prohibited by the specification.

2.2 Flash Translation Layer

Since a page in flash memory cannot be overwritten before it is erased and it takes much longer time to erase a block, an intermediate software layer called FTL (Flash Translation Layer) is usually employed between the kernel and flash memory [3, 5]. The main role of FTL is to emulate the functionality of hard disks with flash memory, hiding the latency of erase operation as much as possible. FTL achieves this by redirecting each write request from the kernel to an empty location in flash memory that has been erased in advance, and by managing the mapping information internally. According to the granularity with which the mapping information is managed, FTL can be classified as page-mapped, block-mapped, or hybrid [5].

FTL also performs *garbage collection* to reclaim free pages by erasing appropriate blocks. Since the lifetime of flash memory is limited, FTL may utilize a *wear leveling* mechanism so that the block erase counts are distributed evenly across the media. In addition, FTL should deal with power-off recovery because there could be unexpected power-outages any time in

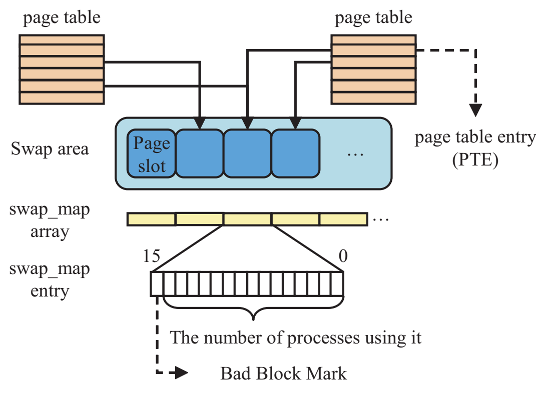

Figure 1. Linux swap system architecture

mobile embedded systems. Readers are encouraged to refer to [5] for further details on FTL.

2.3 Linux Swap System

Figure 1 illustrates the overall architecture of the swap system in the Linux kernel 2.4. When a page is swapped out, its location in the *swap area* is stored in the corresponding page table entry (PTE). Several different swap areas may be defined, and each swap area consists of a sequence of *page slots*: 4KBytes blocks used to contain a swapped-out page. The `swap_map` array is used to record the state of each page slot. If an entry in the *swap_map* is equals to 0, the page slot is free; if it has the value 32,768, the page slot is considered defective, and thus unusable. If an entry is positive, the page slot is filled with a swapped-out page. For the shared page which is pointed by multiple PTEs, the entry represents the number of processes sharing it [2]. Even if the number of processes sharing a page slot exceeds the maximum value[1], the kernel correctly works because the page slot is assigned to the corresponding memory page permanently and cannot be removed from the slot.

On a page fault, the kernel simply reads a page from the page slot pointed by PTE. After the kernel reads a page, it examines the swap area utilization. If the swap area is used more than 50%, the kernel attempts to free the page slot which is just swapped in. Otherwise, the page slot is not freed and gets reused when the same page is swapped out.

Swapping out a page consists of two phases, the allocation phase and the writing phase. At the allocation phase, the kernel assigns page slots to memory pages which are selected as victims. The position of the allocated page slot is stored in the PTE at this time. If a memory page already has an associated page slot, the kernel reuses it instead of finding a new free page slot. The actual writing of the victim page to the assigned page slot takes place in the writing phase, when the page is not recently accessed and every PTE sharing the page indicates the position of the page slot. Note that the writing phase does not follow the allocation phase immediately. This implies that although page slots are assigned to memory pages sequentially, the order in which page slots are written may not be sequential.

[1]The maximum value of Linux kernel 2.4 (*SWAP_MAP_MAX*) is 32,767.

2.4 Motivation and Contribution

Our work is primarily motivated by the observation that the traditional swap architecture, which regards flash memory as a conventional hard disk through the use of FTL, can be very inefficient since useful information is not shared between the kernel and FTL. For example, although there are many invalidated page slots in the swap area due to process termination, unmapped anonymous pages, and page slot release by the kernel, such information is not readily available to FTL. This will result in a situation where FTL uselessly copies those pages during garbage collection, considering them as valid.

Some functions of FTL are even unnecessary for swap system. For instance, the comprehensive power-off recovery is not necessary for swap system, since the contents of page slots are meaningless after the system restarts. For the same reason, it is not required to preserve mapping information managed by FTL.

This inefficiency can be avoided by turning off useless functions of FTL and providing communication channels between the kernel and FTL. However, this requires modification of both layers and incurs communication overhead. Moreover, the extra mapping performed inside FTL is redundant if the page table entry (PTE) points to the physical location of page slot directly.

In this paper, we propose a novel flash-aware swap system called FASS. To the authors's best knowledge, this is the first approach which directly manages NAND flash memory for swap system without the use of FTL. Our contributions can be summarized as follows.

- We introduce a NAND flash memory management scheme using page table entries and `swap_map` array, removing the extra mapping usually done inside FTL.
- We take advantage of kernel information to eliminate unnecessary page copy during garbage collection.
- We implement our approach based on the Linux kernel 2.4, minimizing additional data structures and removing several unnecessary optimizations performed for disks.
- We compare the proposed approach with the conventional swap system built on top of FTL. The evaluation results indicate that our approach is very effective in reducing the number of actual flash memory operations.
- Finally, we predict the lifetime of NAND flash memory when it is used as a backing storage for swap system.

3 Design and Implementation of FASS

3.1 Basic Design

The basic idea behind FASS is to use the existing PTEs to point to the *physical* location of the page slot in NAND flash memory, as shown in figure 2. By eliminating the extra logical-to-physical mapping which is usually performed inside FTL, we can save the space required to construct the mapping table. Moreover, the runtime overhead can be minimized, since otherwise FTL would try to synchronize the mapping table to NAND flash memory periodically for power-off recovery.

In the conventional swap system, a dirty memory page is swapped out to the same page slot, if the memory page still has an association with the previously allocated page slot. FASS, however, assigns a new empty page slot whenever a dirty memory page is swapped out in order to avoid overwriting the previous page slot.

Recall that each page slot may be in one of three states, *bad*, *used*, or *free*, and this information is recorded in `swap_map` array. FASS requires an additional state to identify invalidated page slots. We call this state *dirty* and assign a bit in a `swap_map` array as shown in figure 3. A dirty page slot is not allocated until FASS erases the block containing it.

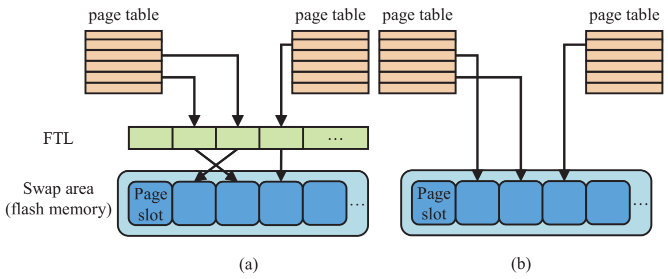

Figure 2. Conventional swap system (a) and FASS (b) running over flash memory

3.2 Shared Page Problems

The FASS architecture described in the previous section works well if the system has only private pages. However, there is a problem caused by shared pages as shown in the following scenario.

Let us assume that two processes, P_1 and P_2, are sharing a page Q, and Q is swapped out at page slot S_0. In this case, both corresponding PTEs of P_1 and P_2 will point to S_0. Now consider a situation where P_1 makes a write request to the page Q; the page Q will be swapped in into the physical memory and become dirty. If the page Q is swapped out later, FASS assigns a new page slot S_1 to avoid overwriting S_0. The PTE of P_1 is updated to point to S_1, but the PTE of P_2 will still indicate the old invalidated page slot S_0.

This problem can be solved easily if we can update the PTE of P_2 immediately when the page Q is swapped out to a new page slot S_1. Unfortunately, however, the Linux kernel 2.4 does not have a backward pointer from a memory page to the corresponding PTE. Thus, when a new page slot S_1 is assigned to the page Q, other PTEs that point to the same page cannot be discovered unless the whole page tables are searched. In the original Linux kernel, this is not a serious problem since the kernel simply overwrites the existing page slot without needing to allocate a new one.

FASS solves this problem by maintaining a hash table which records the fact that the latest copy of S_0 is available at S_1. We use another bit in the `swap_map` array to indicate the need for consulting the hash table (see figure 3). In the previous scenario, when the page Q is swapped out to a new page slot S_1, FASS marks a *hash table mapping* bit in the `swap_map` entry of S_0, and inserts an entry $S_0 \rightarrow S_1$ to the hash

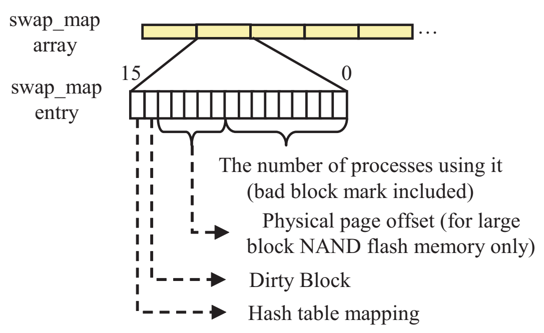
swap_map array
swap_map entry
15
0
The number of processes using it (bad block mark included)
Physical page offset (for large block NAND flash memory only)
Dirty Block
Hash table mapping

Figure 3. A swap_map array entry of FASS

table. Later, when the process P_2 makes a reference to the page Q, the kernel checks the hash table mapping bit in the `swap_map` entry of S_0, and if it is set, finds the valid copy at the page slot S_1 using the hash table information.

The kernel removes the hash table entry when the page slot S_0 is not used anymore by the processes sharing it. Then the page slot S_0 and S_1 can be reused without hash table mapping. Even though the hash table mapping $S_0 \rightarrow S_1$ still exists, the page slot S_0 can be reused through another hash table mapping.

Since the number of shared pages is usually small and there is very little chance where shared pages are swapped out, the hash table is a very effective way to deal with the shared page problem. The presence of the hash table hardly affects the overall performance.

3.3 Supporting the Large Block NAND Flash Memory

As described in section 2.1, the large block NAND flash memory has a restriction that the pages should be programmed sequentially within a block. This causes a serious problem in FASS because swapping out is handled by two separate phases, the allocation phase and the writing phase. Even if sequential page slots are allocated to victim pages during the allocation phase, there is no guarantee that the actual write to the page slot is issued sequentially within a block. It is not possible to predict the actual write order in the allocation phase either, because the moment a victim page is written to the page slot depends on many runtime factors.

We can imagine several solutions for this problem. The first is to assign sequential page slots to victim pages at the allocation phase, and then flush them immediately to NAND flash memory. Alternatively, when a victim page is written to the assigned page slot, we can find all the preceding pages inside the block and write them together. Although these methods can guarantee the sequential write order within a block, they tend to increase the swapping traffic by forcing some pages to be written unnecessarily.

FASS supports the large block NAND flash memory by constructing an intra-block mapping inside the entry of `swap_map` array. In the allocation phase, page slots are allocated to victim pages in a usual way. When a victim page is written to NAND flash memory, FASS writes it to the next available page slot in the block which does not violate the sequential write order, and records the page slot number within the original `swap_map` entry. Therefore, a victim page is always written to the block to which it is originally assigned, but the actual location within the block may vary depending on the write order of victim pages belonging to the same block. Because each block can contain 32 page slots in the large block NAND flash memory, we need to use 5bits inside the `swap_map` entry to record the physical page offset.

The resulting layout of the `swap_map` entry used in FASS is depicted in figure 3. Due to the assignment of several bits for handling flash memory, the remaining space is reduced to 9 bits which can represent the maximum 512 processes sharing a page slot. However, this has little influence on the system because personal computers or small system rarely exceed maximum value. Although the number of processes sharing a page slot becomes greater than 512, the kernel guarantees correct operation as the original Linux kernel.

3.4 Garbage Collection and Wear-Leveling

The primary role of garbage collector is to erase flash memory blocks and to reclaim clean pages without loss of valid data. In the swap system, many page slots become invalidated due to various reasons such as overwriting the same page slot, process termination, unmapped anonymous pages, and system restart. FASS makes use of these informations to eliminate unnecessary data copy during garbage collection. In FASS, the garbage collector is triggered when there are free pages lower then the predefined threshold. It uses a simple version of cost-benefit algorithm [4, 9] to select victim blocks. If there are valid page slots in the victim block, they are swapped in before the block is erased.

Since the lifetime of flash memory is limited, it is important to distribute erase operations evenly among the whole flash memory blocks. In particular, as the size of swap space is usually small and there may be frequent swap-out requests, the importance of wear leveling should be emphasized. In FASS, if the difference between erase counts of the victim block and the minimally-erased block exceeds a certain threshold, these two blocks are exchanged to achieve wear leveling.

4 Experimental Results

4.1 Evaluation Methodology

We have implemented a FASS prototype system based on the Linux kernel 2.4. Our machine consists of a Pentium III processor, 32MBytes RAM, and 32MBytes NAND flash memory[2]. In order to induce a modest number of garbage collections, we decided to use only 32MBytes flash memory. With this size we can exhaust whole flash memory area several times during the experiments. The performance of FASS is compared with that of the original Linux swap system running over the log-scheme FTL[5]. We turned off the swap read-ahead feature in both configurations to isolate the impact of different swap space management schemes.

In order to generate swap-in and swap-out operations, we run two Linux application sets which require more than 32MBytes RAM. The first application set consists of X server and Mozilla web browser, in which a user visits several web sites including Google, CNN, Amazon, and Hotmail during about 30 minutes. The second application set models a situation where a user plays a song with the xmms mp3 player while compiling the Linux kernel with gcc. The first application set is less heavily loaded than the second one because the first one is an interactive job.

4.2 Flash Memory Accesses

Figure 4 compares the number of swap-in and swap-out operations generated in each configuration. It is obvious in figure 4 that both configurations show almost the same number of swap-in/swap-out operations, since FASS is not intended to reduce the number of such operations.

Figure 5 and 6 illustrate the number of flash read/write and erase operations performed for NAND flash memory. Although the number of swap-in/swap-out operations is not much different, the experimental

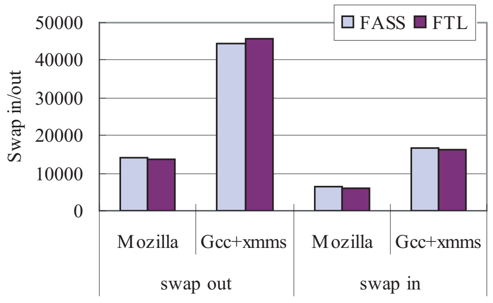

Figure 4. The number of swapping operations

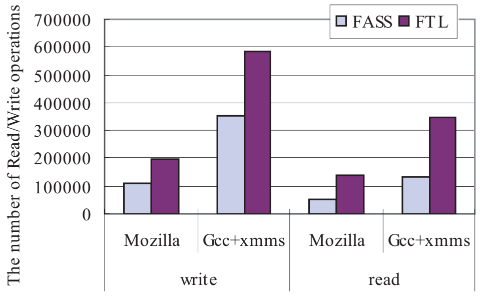

Figure 5. Comparison of flash read/write accesses

results indicate a significant benefit of FASS in terms of the number of actual flash memory operations. This is because FASS can fully take advantage of the various kernel informations in managing swap space, as explained in section 2.4.

On average, the number of flash read and write operations is decreased by 61% and 41.5%, respectively. FASS also reduces the number of erase operations by 84% with the assistance of the internal kernel information which allows to reclaim as many free pages as possible during garbage collection.

In this paper, we only measure the number of swapping and flash memory operations because RAM-emulated flash memory cannot provide actual timing of NAND flash memory. So we leave running time measurement and other evaluation related to this as a further work.

4.3 Wear-Leveling Impact

Since the result of FTL is much worse than FASS, we only compare the following two configurations: FASS with wear leveling and FASS without wear lev-

[2]The NAND flash memory is emulated with the spare RAM available on the system.

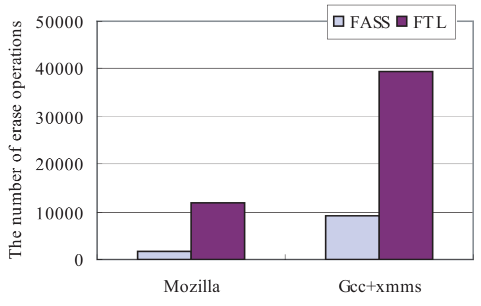

Figure 6. The number of erase operations

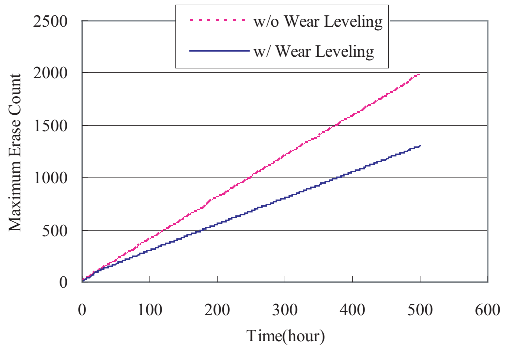

Figure 7. The growth of the maximum erase count with respect to running time

eling. In order to investigate the effectiveness of wear leveling and to estimate the lifetime of flash memory, we simulated the behavior of FASS with a two-hour long swap system trace. The trace is obtained from a system which runs various applications such as GNOME desktop manager, Mozilla, xmms, gcc, terminals, and vi editor.

Figure 7 compares the growth of the maximum erase count in two configurations. If we assume that the maximum erase count of a block is limited to 100,000 and the same trace is repeatedly given to the system, the estimated lifetime of flash memory is about 4.5 years with wear leveling and about 2.9 years without wear leveling. We can see that the wear leveling mechanism used in FASS improves the lifetime of flash memory about 55% longer. However, it does not come for free. Our evaluation shows that the number of erase operations is increased by 8.0% and the number of write operations by 7.1% due to the wear leveling. In addition, we should notice that this estimation would vary according to its workloads and running environments.

5 Conclusion and Future Work

In this paper, we have presented the design and implementation of flash-aware swap system (FASS), in which the kernel manages NAND flash memory directly without the use of intermediate software layers such as FTL. We have also shown the effectiveness of FASS by performing experiments with our prototype implementation running several real workloads. According to our experiments, FASS actually reduces the number of flash memory operations by about 39–87% by utilizing various kernel-level information in managing NAND flash memory. In addition, by simulating FASS with a swap system trace, we show that the wear leveling employed in FASS improves the total lifetime of flash memory by 55%.

In the near future, we plan to port FASS on the Linux kernel 2.6, which enables to exploit the reverse mapping from a memory page to the corresponding PTEs. In addition, more detailed analysis on the performance of FASS such as actual completion time and average access time with various configurations will be examined.

References

[1] *Yet Another Flash Filing System (YAFFS)*. Aleph One Company.

[2] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly, second edition, 2003.

[3] `http://www.intel.com/design/flcomp/applnots/29781602.pdf`.

[4] A. Kawaguchi, S. Nishioka, and H. Motoda. A flash-memory based file system. In *USENIX Winter*, pages 155–164, 1995.

[5] 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*, 48(2):366–375, May 2002.

[6] `http://www.m-systems.com/content/Products/FFDFamily.asp`.

[7] C. Park, J.-U. Kang, S.-Y. Park, and J.-S. Kim. Energy-aware demand paging on nand flash-based embedded storages. In *ISLPED '04: Proceedings of the 2004 international symposium on Low power electronics and design*, pages 338–343. ACM Press, 2004.

[8] C. Park, J. Seo, D. Seo, and B. Kim. Cost-efficient memory architecture design of nand flash memory embedded systems. In *ICCD '03: Proceedings of the 21st International Conference on Computer Design*, pages 474–489. IEEE, 2003.

[9] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, 1992.

[10] `http://www.samsung.com/Products/Semiconductor/Flash/TechnicalInfo/datasheets.htm`.

[11] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. John Wiley & Sons, sixth edition, 2003.

[12] D. Woodhouse. JFFS : The journalling flash file system. Ottawa Linux Symposium, 2001.