International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:2, No:3, 2008

Performance Trade-Off of File System between Overwriting and Dynamic Relocation on a Solid State Drive

Choulseung Hyun, Hunki Kwon, Jaeho Kim, Eujoon Byun, Jongmoo Choi, Donghee Lee, and Sam H. Noh

***Abstract*—**Most file systems overwrite modified file data and metadata in their original locations, while the Log-structured File System (LFS) dynamically relocates them to other locations. We design and implement the Evergreen file system that can select between overwriting or relocation for each block of a file or metadata. Therefore, the Evergreen file system can achieve superior write performance by sequentializing write requests (similar to LFS-style relocation) when space utilization is low and overwriting when utilization is high. Another challenging issue is identifying performance benefits of LFS-style relocation over overwriting on a newly introduced SSD (Solid State Drive) which has only Flash-memory chips and control circuits without mechanical parts. Our experimental results measured on a SSD show that relocation outperforms overwriting when space utilization is below 80% and vice versa.

***Keywords*—**Evergreen File System, Overwrite, Relocation, Solid State Drive.

I. Introduction

IN most file systems including FFS (Fast File System), Ext3, FAT, and NTFS, locations of files and metadata are fixed, and when data is modified they are always overwritten in their original location. We will refer to this type of scheme as the *overwrite scheme*. With the overwrite scheme, if geometrically dispersed data are modified within the same time span, seek and rotational delay substantially influences disk I/O time resulting in underutilization of disk I/O bandwidth. In order to fully utilize disk bandwidth for write requests, the LFS (Log-structured File System) was introduced, where all modified data are collected in memory chunks, then written to disk together [1]. In other words, LFS relocates every modified block in order to sequentialize the write requests. We will refer to this type of scheme such as LFS where modified data are relocated to a different location, the *relocation scheme*. LFS, however, has the disadvantage that it must reclaim contiguous disk space called *segments* for further writes, and the efficiency of reclamation (*cleaning* in many literatures) determines the performance of LFS [2-4]. It has been shown that due to this cleaning overhead, the performance of LFS degrades when the file system utilization rises over a certain point, which is determined by characteristics of the storage device and fragmentation degree of the free space [5, 6]. In order to attain advantages of both the overwrite and relocation schemes, Wang et al. propose a hybrid approach called HyLog model that relocates hot data and overwrites cold data [6]. They also predict that disk technologies in the future will favor the LFS-style relocation scheme.

Though the LFS-style sequential writing scheme has potentials for superior write performance, there have been numerous debates concerning the performance superiority between the overwrite and relocation schemes. File systems today make use of either of the schemes. In fact, even though LFS use the same inode structure as UNIX file systems, no implementation of LFS overwrites any part of the files. Though the HyLog file system proposes to exploit the advantages of both relocation and overwrite, it was limited to a simulation and analytical study.

In this paper, we present the design and implementation of a new file system that we call the Evergreen file system. The central idea of the Evergreen file system is that it is able to choose the modification scheme, that is, choose between the overwrite and relocation schemes for each modified block of a file and metadata. Currently, the Evergreen file system can apply a different modification scheme for each category of metadata and file data. Another issue challenged in this paper is identifying whether newly introduced SSDs show similar performance characteristics to magnetic disks or not. Specifically, we focus on the performance benefits of LFS-style relocations scheme over overwriting according to file system utilization on a SSD which has only Flash-memory chips and control circuits without mechanical parts such as moving heads and rotating platters.

Main topics of this paper are in the design and implementation of the Evergreen file system and also in the performance trade-offs of relocation and overwriting on a SSD.

Manuscript received March 31, 2008.

Choulseung Hyun, Hunki Kwon, Jaeho Kim, Eujoon Byun, and Donghee Lee are with the School of Computer Science, University of Seoul, Seoul, Korea (e-mail: cshyun@uos.ac.kr, kwonhunki@uos.ac.kr, kjhnet@gmail.com, smilejoon@uos.ac.kr, and dhl_express@uos.ac.kr). Donghee Lee is a corresponding author.

Jongmoo Choi is with the Division of Information and Computer Science, Dankook University, Youngin, Korea (e-mail: choijm@dankook.ac.kr).

Sam H. Noh is with the School of Computer and Information Engineering, Hongik University, Seoul, Korea (e-mail: samhnoh@hongik.ac.kr).

In our presentation, we will focus on the relocation features of the Evergreen file system rather than on the basic file system operations. Also, our experiments will concentrate on SSDs as the benefits of LFS-style relocation on HDDs are well known. Surprisingly, the performance benefit of overwriting on an SSD is similar to that of HDDs in our experiments.

This paper is organized as follows. In Section 2, we describe the motivations of the Evergreen file system and related works. In Section 3, we explain the design and structure of the Evergreen file system. Section 4 shows experimental results measured on a SSD and we conclude this paper in Section 5.

II. Related Works

The Log-structured File System (LFS) collects modified file data and metadata in memory and writes them sequentially to a segment on disk. Even though LFS uses the same inode structure as conventional UNIX file systems that overwrite modified data, no implementation of LFS supports overwriting of file data and metadata. Unlike LFS, the Evergreen file system can selectively relocate each file or metadata block by modifying only the relevant file system structure for that block. The performance gain of LFS and extent-based allocation schemes in real environments has been debated for a long time [5, 7] and many performance models were defined for LFS-style relocation [1-4, 6]. The HyLog file system was proposed along with its performance models for overwriting and relocation. However, previous models did not consider Solid State Disks (SSDs) as this technology is relatively new. Currently, performance characteristics of SSDs are challenging issues.

File systems that dynamically relocate modified data do exist. For example, JFFS2 [8] and YAFFS [9] relocate modified data to write them sequentially on Flash-memory media. Also, they recycle Flash-memory space by a garbage collection scheme that is similar to cleaning in LFS. However, as these systems directly control the raw Flash-memory chips, they cannot run on storage devices that provide block device interfaces such as magnetic disks, SSDs, and USB drives.

A major issue that must be considered in file system design is the recovery scheme. Creation/deletion of a file/directory is accompanied with multiple block updates and those modified blocks need to be updated altogether. If not, the file system falls into an inconsistent state. To recover from file system inconsistency, some file systems rely on file system check and recovery utilities (ex, *fsck* in UNIX system). However, recovery time increases proportionally to disk capacity and often takes too much time to be acceptable in commercial environments. To reduce recovery time, some file systems such as Ext3 and NTFS use journaling mechanisms [10, 11] that write logging information before modifying the file system structure. Another approach called Soft Update [12, 13] has been introduced and applied in the Sun file system.

The WAFL (Write Anywhere File Layout) file system relocates modified file data and metadata blocks to new locations and the relocated data are confirmed only after a checkpoint [14]. Therefore, the file system can recover file system consistency immediately after system crash and also it can provide multiple versions of file system state. Moreover, the WAFL file system has some chances to increase performance by relocating file and metadata blocks to somewhere near current disk head position. In some aspects, the design of the WAFL shares some common things with the Evergreen file system. However, design and implementation of the WAFL file system focuses on providing multiple versions of file system state while the Evergreen file system focuses on performance trade-offs between overwriting and fine-grained relocation. For example, the Evergreen can overwrite existing file and metadata for performance reasons and the dynamic selection of overwriting and relocation is the main difference of the WAFL and Evergreen file systems.

III. Design of the Evergreen File System

In this section, we explain design of the Evergreen file system. In our presentation, we will use conventional terms such as inode of UNIX file systems and ifile of LFS even though our code uses different names. Also, we will focus on the relocation feature of the Evergreen file system rather than on other basic file system operations. Therefore, we assume in the following descriptions that the Evergreen file system always relocates every file data and metadata block when it is modified.

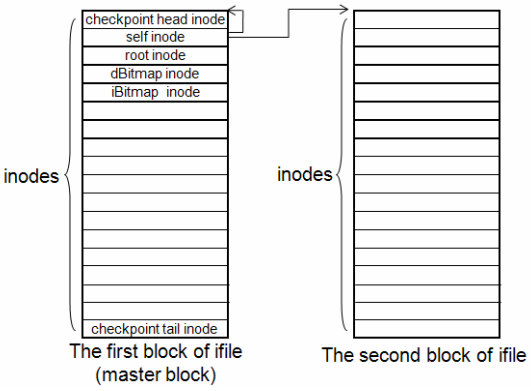


Fig. 1 ifile and master blocks

The Evergreen file system has an ifile that was originally introduced in LFS. In the original UNIX file system, all inodes have their own fixed locations. However, LFS packages inodes into an ifile in order to relocate them on demand. Like LFS, the Evergreen file system uses the ifile to package inodes. As a result, an inode can move to a new location by relocating a part of the ifile. We will call the first block of the ifile the *master block*, where the self inode, root inode, dBitmap inode, and iBitmap inode reside (Fig. 1). The self inode (inode 1) represents the ifile itself and keeps block numbers of the ifile itself. By convention, the root directory uses the third inode (inode 2) and its role is the same as that in other UNIX file systems. The dBitmap inode (inode 3) represents the dBitmap file, which contains a bit-array for all the blocks in the file

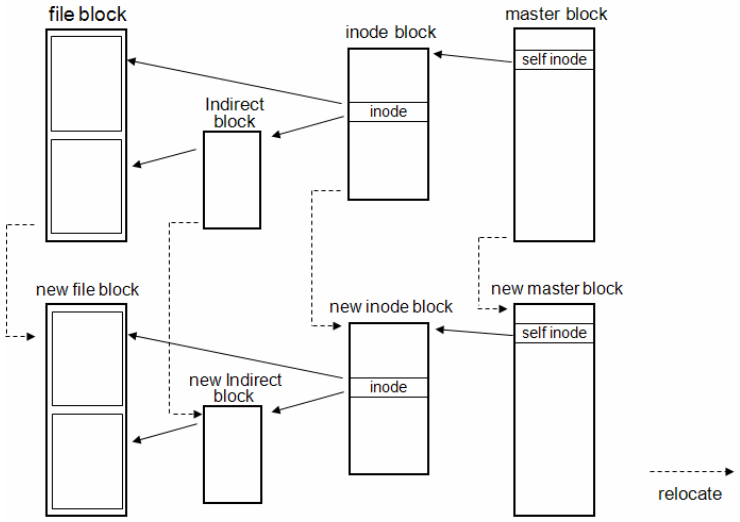


Fig. 2 Cascading relocations in Evergreen file system

system, with each bit indicating whether the block is used or not. The iBitmap file represented by the iBitmap inode (inode 4) contains a bit-array indicating whether an inode is being used or not. The first and the last inodes in the master block are used to store check-point information. Let us assume that 16 inodes reside in the master block. Then, the check-point information is stored in inodes 0 and 15, which we call the head and tail check-point inodes, respectively. The check-point information consists of a 64-bit sequence number, logging information (described later), and a checksum of the check-point information itself. As all file system structures starts from the master block, modifications of file system structures are committed by writing the master block. On the contrary, modifications are aborted if the system crashes before writing the master block.

Fig. 2 shows cascading relocations of relevant metadata when a file/directory data block is modified. When a file data block is modified in the buffer cache, the file system allocates a new location and relocates the modified block to the new location (relocation is done by just changing the block number of the cached block in the buffer cache). If an indirect block is used to address the data block, a new location is allocated again and the indirect block is also relocated. Then, the file system modifies the inode of the file to point to the new location or to the new indirect block. Modification of the inode requires relocation of a part of the ifile and the modified block of the ifile is also relocated to a new location. As block numbers of the ifile are kept in the self inode, relocation of the ifile block is accomplished by registering the address of the new location in the self inode. In all cases, block relocation requires allocating a new empty block and freeing an old block, and these all modify the contents of the dBitmap file (not shown in Fig. 2). If the contents of dBitmap are changed, the modified dBitmap blocks must also be moved to new locations and their locations will be registered in the dBitmap inode. As the self inode and dBitmap inode exist in the master block, all cascading modifications end in the master block.

Relocation of the Evergreen file system seems to be quite wasteful because it modifies many relevant file system structures to relocate file data or metadata blocks. However, a file/metadata block moves to a new location only when it is modified in the buffer cache for the first time. In other words, the relocation occurs when the state of the cached block is changed from CLEAN to DIRTY. Modification of an already DIRTY block, however, does not trigger relocation. Therefore, because of locality, overall number of relocations is typically not very large for most file system operations if the buffer cache is sufficient. Also, when compared to the overwrite scheme, relocation does not demand excessive overhead. Consider a file system using the journaling mechanism. In this file system, modification of a file/metadata block incurs modification of the inode of that file and also requires additional writing of logging information to the journaling area. Therefore, though relocation has possibilities to incur additional writes when synchronous writes are requested, it is not as inefficient as it appears in case of normal file system operations.

The Evergreen file system calls *commit()* in two cases; when *sync()* is called periodically by the operating system or sporadically by applications and when internal resources of the file system are consumed. The commit procedure writes all DIRTY blocks in the buffer cache except for the master block. When writing DIRTY blocks, it memorizes the last-written block number. Then, it writes the master block to one of the blocks reserved for the master block. As shown in Fig. 3, some blocks are reserved for master blocks and the master block must be written to one of these reserved blocks. Specifically, the Evergreen file system chooses the nearest one to the

last-written block (as it remembers the last-written block number) among the reserved blocks except for one case. If the written to a location closer to the last-written data block. Reversely, higher *n* decreases booting time and available space.

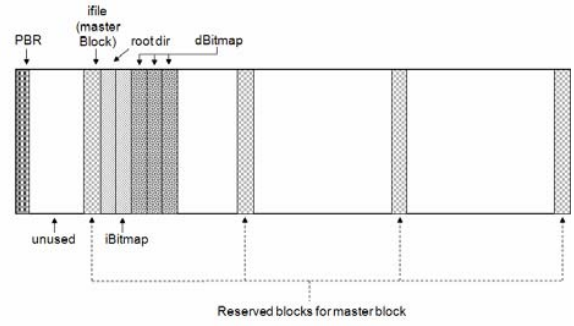


Fig. 3 Layout of Evergreen file system after format

master block was already written to the chosen reserved block at the previous commit, then the Evergreen file system selects the second-to-nearest block. Before writing the master block, the sequence numbers of the head and tail check-point inodes are incremented so as to be the largest among previously written master blocks.

Fig. 3 shows the overall layout of the Evergreen file system right after format. The PBR (Partition Boot Record) resides at the starting sector of the partition and some unused sectors follow. Then, the master block is located at the first block of the file system. As the master block contains self inode, root inode, dBitmap inode, and iBitmap inode, all files and directories can be found and relocated by modifying structures starting from the master block.

LFS divides the entire storage space into a number of segments and scans them to find the last-written one at boot time. As the segment structure is predefined, LFS can easily determine the location of the segment summary block within a segment. However, the Evergreen file system has no segment structure; rather, it has the master block from which all files and metadata stems. Assume that the master block can reside at any block of the file system. Then, the file system must scan all blocks to search the last-written master block. Also, it needs a proper method to discriminate the master block from other data blocks. In order to reduce the scanning delay and to avoid the confusion, the Evergreen file system reserves some blocks for the master block and it scans them to find the last-written master block at boot time. Specifically, the first block, the last block, and the every n^{th} block are reserved for the master block. Currently, the maximal *n* is *1024*, and it can be adjusted to a smaller value according to the capacity of the storage device. The Evergreen file system scans the first, the last, and the every n^{th} block to find the last-written master block, and the boot procedure finishes if the last one is found. Therefore, the boot procedure of Evergreen is similar to LFS in that they both scan to find the last-written segment or master block. There exists trade-offs between performance and boot time and between performance and available space. The lower the *n* value, better performance is expected because the master block can be Considering those trade-offs, we set *n* to *1024* in our experiments, but further investigation as to the effect of this value need to be done. With *n=1024*, the available file system space decreases by about 0.1%.

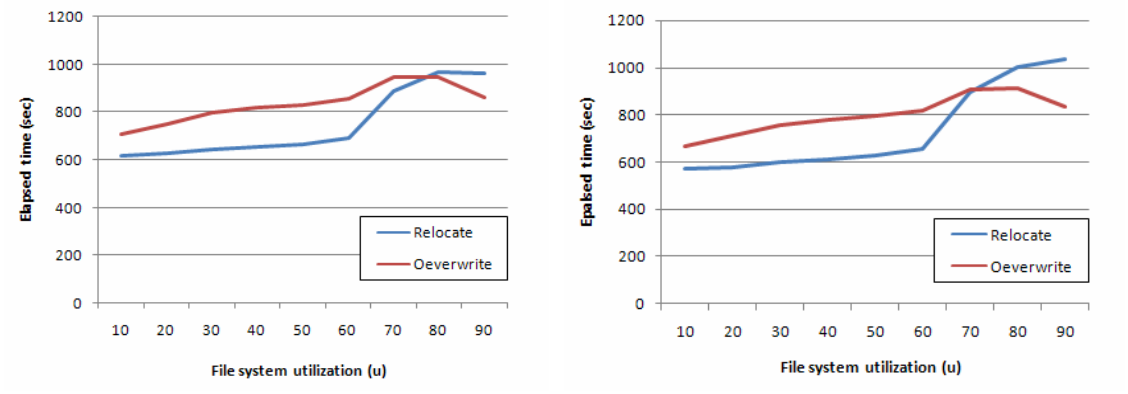
IV. Experimental Results on a SSD

We implemented the Evergreen file system in the Windows CE operating system and measured the performance with the Postmark benchmark [15] on CE/PC. The Evergreen file system has two operation modes; synchronous and asynchronous. In the synchronous mode, the Evergreen file system commits (flushes the buffer cache) after every file system operation such as creating/writing/deleting a file/directory. In the asynchronous mode, the Evergreen file system commits only when the Windows CE operating system requests to flush the buffer cache or when all internal resources of the file system are consumed.

The Windows CE operating system comes with a FAT file system. Because the FAT file system uses the overwrite scheme, it is a good reference to compare performance with the Evergreen file system. In the experiments on an SSD, indeed, the Evergreen file system runs multiple times faster than the FAT file system. However, structural overheads of the two file systems could be different and, thus, a direct performance comparison would be unfair. Therefore, we will focus on the trade-off of the overwrite and relocation schemes of the Evergreen file system.

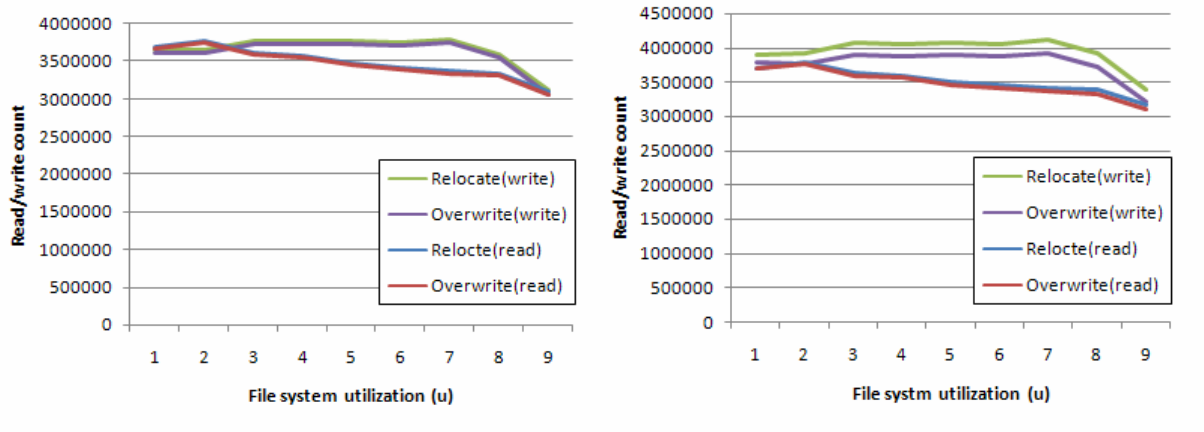
We used a Samsung SSD MCAQE32G5APP-0XA (32G) in our experiments. For consistency of experiments, we made a 1 GB partition and formatted it with the desired file system and run an arbitrary Postmark benchmark to fill the file system up to a desired utilization before the experiments. In the experiments, the buffer cache size of the Evergreen file system is set to 64 blocks.

Fig. 4 shows the elapsed times of the Postmark benchmark program running on the SSD as the file system utilization is varied. In Fig 4(a) and (b), the Evergreen file system operates in synchronous mode and in asynchronous mode, respectively. In order to compare the performance of the relocation and



(a) Asynchronous mode **(b) Synchronous mode**

Fig. 4 Performance of Postmark benchmark program on the Evergreen file system



(a) Asynchronous mode **(b) Synchronous mode**

Fig. 5 Number of read and write requests on the Evergreen file system

overwrite schemes, we did not use adaptive scheme, but set the scheme to either one. In both figures, we can observe that the relocation scheme outperforms the overwrite scheme when file system utilization is low, but performance crosses over at some point between 70% and 80%, beyond which, the overwrite scheme shows better performance the gap increasing as utilization increases. Currently, the Evergreen file system freezes block allocation when file system utilization exceeds 95%. Therefore, the Postmark benchmark program sometimes fails to create files when it starts to execute with 90% utilization (and less frequently 80% utilization). As a result, the experimental results underestimates the elapsed times at utilization 80% and 90% and, if we take this into account, the performances of relocation and overwriting schemes matches well to the models previously proposed for magnetic disks. Specifically, many models for magnetic disks predicted the existence of cross-over point and, even on a SSD, there exists the cross-over point between 70% and 80% of utilization.

Fig. 5 shows the number of read and write requests issued for each of the schemes. In Fig. 5(a), which shows the results executed in asynchronous mode, we observe that the number of relocation write requests is almost the same as that of overwriting. In the asynchronous mode where the buffer cache is fully utilized to delay actual writes, overhead of relocation is minimized because the already dirtied blocks do not incur relocation. Also, there is no notable difference in read request counts between the overwrite and relocation schemes. In the synchronous mode (Fig. 5(b)), relocation has about 5% more write requests than overwriting, which is actually more efficient than we expected. This efficiency can be explained in two ways. First, the Postmark benchmark simulates small file read/write workloads and a small file does not heavily use indirect blocks to address its data blocks. The relocation overhead of small file is small because it requires only changing block addresses in the inode of the file and the inode has to be modified for other reasons such as changing time and size. However, if a large file is modified, then relocation is expected to see higher overhead. Another overhead of relocation is frequent modification of the dBitmap file to allocate and to free blocks. However, buffering and locality seem to hide the overhead successfully. The second reason behind efficient relocation is that the Evergreen file system

efficiently localizes the core structures that must be modified for relocation into a small spot. In the Evergreen file system, cascading modifications converge to the master block and many of modified blocks were to be modified for other reasons anyway. For these reasons, the dynamic relocation of modified blocks can be implemented without significant overheads.

V. Conclusion

The Evergreen file system can choose between the relocation and overwrite schemes as its modification scheme for each modified file data and metadata block. As a result, the Evergreen file system can apply different schemes for each categories of metadata and file data. Recently developed SSDs raise questions about whether they show similar trade-offs to magnetic disks between sequential and random writes and between overwrite and LFS-style relocation schemes. The experimental results of the Evergreen file system measured on a SSD show that LFS-style relocation outperforms overwriting when space utilization is low and vice versa. These results confirm that the SSD has similar performance characteristics to conventional magnetic disks though it has no mechanical parts.

References

[1] M. Rosenblum and J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System," ACM Transactions on Computer Systems, vol. 10, pp. 26-52, 1992.
[2] T. Blackwell, J. Harris, and M. I. Seltzer, "Heuristic Cleaning Algorithms in Log-Structured File Systems," in Proceedings of the 1995 USENIX Technical Conference, New Orleans, Louisiana, USA, 1995, pp. 277-288.
[3] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson, "Improving the performance of log-structured file systems with adaptive methods," ACM Operating Systems Review, vol. 31, pp. 238-251, December 1997.
[4] J. Wang and Y. Hu, "WOLF-A Novel Reordering Write Buffer to Boost the Performance of Log-Structured File Systems," in 1st Conference on File and Storage Technologies, 2002, pp. 47-60.
[5] M. Seltzer, K. A. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan, "File System Logging versus Clustering: A Performance Comparison," in USENIX Annual Technical Conference, 1995, pp. 249-264.
[6] W. Wang, Y. Zhao, and R. Bunt, "HyLog: A High Performance Approach to Managing Disk Layout," in Proceedings of the 3rd USENIX Conference on File and Storage Technologies, San Francisco, CA 2004.
[7] L. W. McVoy and S. R. Kleiman, "Extent-like Performance from a UNIX File System," in Proceedings of the USENIX Winter 1991 Technical Conference, Dallas, TX, USA, 1991, pp. 33-43.
[8] D. Woodhouse, "JFFS: The Journaling Flash File System," in Ottawa Linux Symposium 2001, 2001.
[9] "YAFFS (Yet Another Flash File System) Specification Version 0.3," http://www.aleph1.co.uk/yaffs/, 2002.
[10] R. Hagmann, "Reimplementing the Cedar File System Using Logging and Group Commit," ACM Operating Systems Review, vol. 21, pp. 155-162, 1987.
[11] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis and evolution of journaling file systems," in Proceedings of the USENIX Annual Technical Conference 2005 Anaheim, CA
[12] G. R. Ganger and Y. N. Patt, "Metadata Update Performance in File Systems," in Proceedings of the USENIX 1994 Symposium on Operating Systems Design and Implementation, Monterey, CA, USA, 1994, pp. 49-60.
[13] G. R. Ganger, M. K. McKusick, C. A. N. Soules, and Y. N. Patt, "Soft updates: a solution to the metadata update problem in file systems," ACM Transactions on Computer Systems, vol. 18, pp. 127-153, 2000.
[14] D. Hitz, J. Lau, and M. Malcolm, "File System Design for an NFS File Server Appliance," in Proceedings of the USENIX Winter 1994 Technical Conference, San Fransisco, CA, USA, 1994, pp. 235-246.
[15] D. Katcher, "PostMark: A New File System Benchmark," Network Applicance Inc. 1997.

Choulseung Hyun received the BS degree in telecommunication and computer engineering from Cheju National University in 2001 and the MS degree in computer science from University of Seoul in 2007. He is now a PhD student in the University of Seoul, Korea. His research interests include operating systems, file systems, and performance modeling

Hunki Kwon received the BS and MS degrees in computer science from University of Seoul in 2006 and 2008, respectively. He is now a PhD student in the University of Seoul, Korea. His research interests include operating systems, embedded systems, and flash-memory storage.

Jaeho Kim received the BS degree in information and telecommunication engineering from Inje University in 2004. He is now an MS student in University of Seoul, Korea. His research interests include file systems and cache management algorithms.

Eujune Byun received the BS degree in computer engineering from Daejin University in 2007. He is now an MS student in University of Seoul, Korea. His research interests include embedded systems and flash-memory storage.

Jongmoo Choi received the BS degree in oceanography from Seoul National University, Korea, in 1993 and the MS and PhD degrees in computer engineering from Seoul National University in 1995 and 2001, respectively. He is an assistant professor in the division of information and computer science, Dankook University, Korea. Previously, he was a senior engineer at Ubiquix Company, Korea. He held a visiting faculty position at the University of California, Santa Cruz from 2005 to 2006. His research interests include micro-kernels, file systems, flash memory, and embedded systems.

Donghee Lee received the MS and PhD degrees in computer engineering, both from Seoul National University, Korea, in 1991 and 1998, respectively. He is currently an associate professor in the School of Computer Science, University of Seoul, Korea. Previously, he was a senior engineer at Samsung Electronics Company, Korea, in 1998. His research interests include embedded systems, cache algorithms, and flash-memory storage.

Sam H. Noh received the BS degree in computer engineering from Seoul National University, Korea, in 1986, and the PhD degree from the University of Maryland at College Park in 1993. He held a visiting faculty position at George Washington University from 1993 to 1994 before joining Hongik University in Seoul Korea, where he is now a professor in the School of Information and Computer Engineering. His current research interests include parallel and distributed systems, I/O issues in operating systems, and real-time systems. Dr. Noh is a member of the IEEE, the IEEE computer Society, and the ACM.