A Fast Start-Up Technique for Flash Memory Based Computing Systems

Keun Soo Yim, Jihong Kim, and Kern Koh
School of Computer Science and Engineering
Seoul National University, Seoul 151-742, Korea
{ksyim, kernkoh}@oslab.snu.ac.kr, jihong@davinci.snu.ac.kr

ABSTRACT

Flash memory based embedded computing systems are becoming increasingly prevalent. These systems typically have to provide an instant start-up time. However, we observe that mounting a file system for flash memory takes 1 to 25 seconds mainly depending on the flash capacity. Since the flash chip capacity is doubled in every year, this mounting time will soon become the most dominant reason of the delay of system start-up time. Therefore, in this paper, we present instant mounting techniques for flash file systems by storing the in-memory file system metadata to flash memory when unmounting the file system and reloading the stored metadata quickly when mounting the file system. These metadata snapshot techniques are specifically developed for NOR- and NAND-type flash memories, while at the same time, overcoming their physical constraints. The proposed techniques check the validity of the stored snapshot and use the proposed fast crash recovery techniques when the snapshot is invalid. Based on the experimental results, the proposed techniques can reduce the flash mounting time by about two orders of magnitude over the existing de facto standard flash file system. [*]

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; D.4.3. [**Operating Systems**]: File systems management; B.3.2 [**Memory Structures**]: Mass storage.

General Terms

Design, Management, Measurement, Performance.

Keywords

Fast booting, flash memory, fast mounting, and metadata snapshot.

1. INTRODUCTION

Embedded computing systems should be able to provide an instant start-up time [3]. In these systems, flash memory is typically used as a storage medium because of its small size, shock resistance, and low-power consumption. However, when file systems are mounted for flash memory, it takes a large fraction of the total device start-up time. For example, Figure 1 shows the start-up time of a Compaq iPAQ handheld device running Linux operating system. It shows that the flash mounting time takes over 4 seconds out of the total device start-up time of approximately 14 seconds when 16MB flash memory is used. We also found that the flash mounting time heavily depends on the flash capacity and stored data size. Since flash chip capacity is doubling every year [7], the flash mounting time will soon become the most dominant reason of the delay of system start-up time. In desktop computers, we also face the long flash mounting time when an USB flash device is plugged in. Thus, a fast mounting technique for flash file systems needs to be developed for embedded and desktop systems.

Basically this long mounting time is attributed to two physical constraints of write operation in flash memory [2]. First, since flash memory is a version of EEPROM, write operations should be preceded by an erase operation. Second, an erase operation, which can be performed in a larger granularity (i.e., block) than a write operation (i.e., page), takes relatively long time from a millisecond to a second. In order to hide the erase operation from upper-layer application programs, the existing flash file systems, e.g., JFFS2 [9] and YAFFS2 [1], use an out-place update method, which redirects an update request to a page which has been erased in advance. Since the location of updated page is not recorded in the original page, the file systems scan the entire flash space at mounting time in order to collect the location of lastly updated pages. The collected data are then reorganized in the main memory. This delays the flash mounting time significantly.

In this paper, we present instant mounting techniques for flash file systems. The proposed techniques store the in-memory file system metadata to flash memory when unmounting the file system and reload the stored metadata quickly when mounting the file system. The metadata snapshot techniques are specifically developed for NOR- and NAND-type flash memories by overcoming their physical constraints.

First, in flash memory, the number of erase operation that can be performed in a block is limited (e.g., from 100 thousands to 1

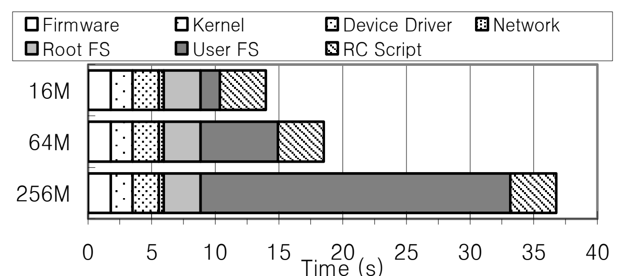

Fig. 1. A breakdown of boot-up time in a PDA running Linux.

[*] This research was supported by the Ubiquitous Autonomic Computing and Network Project, 21st Century Frontier R&D Program in Korea and was also supported by University IT Research Center Project in Korea.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'05, March 13-17, 2005, Santa Fe, New Mexico, USA.
Copyright 2005 ACM 1-58113-964-0/05/0003…$5.00.

million times). Thus, all blocks should be erased evenly in order to reach the erase limit simultaneously, namely the wear-leveling property. In order to ensure this, the proposed snapshot techniques select less frequently erased blocks and store a variable-size snapshot into them by using a linked list (Section 3.1 and 3.2). The proposed snapshot technique for NOR flash finds the last stored snapshot instantaneously as it reserves the first block as an ordered tree that manages pointers to the snapshot head blocks (Section 3.1). Second, this technique is further extended for NAND flash, which supports only page-based I/O [11] (Section 3.2). The proposed snapshot technique for NAND flash also ensures the wear-leveling property and finds the last stored snapshot in an instant.

Third, these snapshot techniques check the validity of the stored snapshot at mounting time. If the snapshot is invalid, three novel fast crash recovery techniques are used to rebuild the in-memory metadata quickly (Section 3.3). Fourth, a data compression technique is used to reduce the snapshot size and consequently the time used to store the snapshot to flash memory when unmounting the file system (Section 3.4).

The experimental results show that the proposed techniques ensure the wear-leveling property and also reduce the flash mounting time by about two orders of magnitude over the existing de facto standard flash file system [9]. For example, when the flash capacity is 128MB and the stored data size is 100MB, the mounting time is reduced from over 10 seconds to below 100 milliseconds. The results also show that the proposed fast recovery techniques significantly reduce the I/O time required for mounting the crashed file system. Although the proposed snapshot technique for NOR flash takes up a longer unmounting time than the existing file system, the data compression technique reduces the unmounting time of the proposed snapshot techniques by over 50%.

The rest of this paper is organized as follows. In Section 2, we describe the overall organization of the existing flash file systems and review the existing snapshot techniques as compared with the proposed techniques. The proposed techniques for NOR- and NAND-type flash memories are described in Section 3, while the evaluation results are given in Section 4. In Section 5, we conclude this paper with a summary.

2. RELATED WORK

In embedded computing devices, flash memory has significant merits such as light-weight, shock-resistance, and low-power consumption. There are two types of flash memories: NOR and NAND. NOR flash is usually used as a code storage medium because it supports word-unit I/O and provides faster read speed than NAND flash. Conversely, as NAND flash supports only page-based I/O (e.g., 512B or 2KB) and provides faster write speed, it is widely used as a large-scale data storage medium.

In flash memory, write operations have to be preceded by an erase operation, and the number of erase that can be performed in a block is limited. In order to hide the erase while ensuring the wear-leveling property, the flash translation layer (FTL) [4, 5, 10] has been developed to be able to translate the logical addresses generated by a host system to physical addresses of flash memory. Since FTL provides an identical abstraction to hard disks, a disk file system (e.g., FAT and ext2 [8]) is used in the host to control the FTL-based flash memory. However, the use of FTL is restricted by international patents, and it is known that the performance of FTL can be seriously degraded if the host file system generates write operations frequently.

Therefore, recent embedded computing devices tend to directly use flash file systems (e.g., JFFS2 [9] and YAFFS2 [1]) which are based on a data journaling technique. Unfortunately, the journaling technique incurs long mounting time for the flash file systems as exemplified in Figure 2 where the root directory contains a file which is updated by a write operation. In step I, the root directory has an inode and a directory entry [8] which tell that it has the file *A* whose inode number is 2. The file also has an inode which means that its version is 1 and the attached data range is 1 to 4 pages. In step II, when a write operation is performed on the file, the journaling technique makes a new inode for the file rather than directly updating the data pages attached to the existing inode. In this new inode, the version is increased by 1 and the range exactly specifies the updated data pages. Then, the data pages 2 and 3 attached to the old inode become invalid. If the empty block ratio of flash memory is lower than the specified threshold value (e.g., 20%), a garbage collector copies the valid pages in the block 2, which has the smallest number of valid pages, to an empty block as shown in step III. The garbage collector finally erases the block 2 as shown in step IV. In this manner, the existing flash file systems logically hide erase operations from application programs. Furthermore, these file systems ensure the wear-leveling property as the performed erase count is considered when selecting blocks for new inode and data pages.

In order to directly find an inode which has the latest copy of a requested page, at mounting time these file systems scan entire flash memory space and build an in-memory metadata as shown in Figure 3. For example, in order to access the second page of the

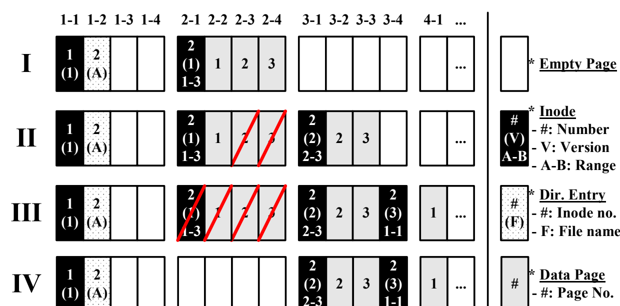

Fig. 2. Data management in the journaling flash file systems.

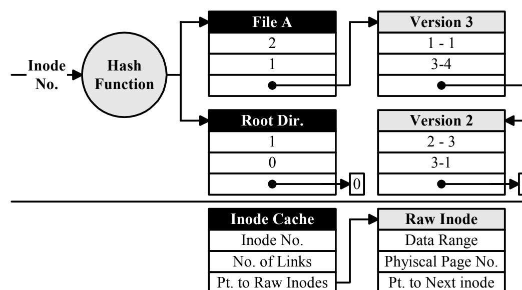

Fig. 3. An in-memory metadata of the journaling flash file systems.

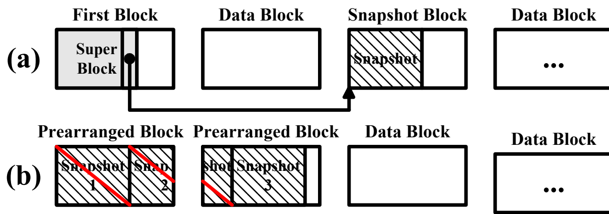

Fig. 4. Existing snapshot techniques in LFS(a) and FTL(b).

file, these systems first finds its inode cache data in the in-memory metadata by using a hash function with its inode number. They then traverse the linked raw inodes until they reach the raw inode whose data range includes the second page. Finally the physical page number stored in the raw inode is used to access the requested page in flash memory.

However, scanning flash space and building an in-memory metadata require long I/O and computation times, respectively. This in turn delays the mounting time of the existing flash file systems significantly, especially when both the flash capacity and stored data size are large.

The existing flash file systems borrowed the journaling idea from a disk file system of the log-structured file system (LFS) [6]. In LFS, a metadata snapshot technique is used to reduce the long mount time as well as the crash recovery time. Specifically, LFS periodically stores a metadata snapshot to a hard disk and updates the super block placed in the first sector of the disk for directing the stored snapshot as shown in Figure 4(a). However, this technique is not directly applicable to flash memory due to the following two reasons. First, since the update operation requires that the super block be erased and the original super block data be partially restored, the data can be lost if the power goes down unexpectedly during the restoration, causing a data integrity problem. Second, the super block reaches the erase limit quickly, thereby breaking the wear-leveling property.

Kim *et al.* [5] adopted a metadata snapshot technique for FTL where the snapshot size is small (e.g. 512B) and fixed. The technique stores a metadata snapshot to dedicated areas in a round-robin manner and reloads the latest snapshot at mounting time from the areas as shown in Figure 4(b). Thus, the technique avoids the integrity problem and keeps the wear-leveling property. However, the technique is not applicable to flash file systems where the snapshot size varies between below 100KB and over 1MB depending on the stored data size because the technique breaks the wear-leveling property. For example, if the snapshot size is relatively large as compared with the size of dedicated areas, the areas reach the erase limit quickly, and conversely, if the snapshot size is small, the remaining flash memory space reaches the limit quickly. Furthermore, the technique stores the snapshot whenever it is changed. However, since the snapshot size of flash file systems is quite large, this can degrade the I/O performance significantly in the file systems. The snapshot techniques for flash file systems which will help overcome these technical obstacles are discussed in the following section.

3. INSTANT MOUNTING TECHNIQUES

This section describes the proposed snapshot techniques for NOR and NAND flashes with three fast crash recovery techniques and a fast unmounting technique.

3.1 Snapshot Technique for NOR Flash

Our design goal is to provide an instant lookup method for the lastly stored snapshot, while ensuring the wear-leveling property. In order to achieve these goals, we store snapshots to variable-size areas managed by linked lists and sequentially record the location of the stored snapshots to prearranged areas by using an ordered tree data structure.

In Figure 5, it can be seen that the first block of flash memory is reserved as a root block which sequentially stores pointers to snapshot header blocks. And since the block size (B_{size}) is typically 128KB in NOR flash and the size of a pointer to block is 2B (P_{size}), the maximum number of pointers that can be stored in the root block is B_{size} / P_{size} (=2^{16}). Pointers are sequentially stored from the first word in the root block. If the root block is full, the block is erased before writing a new pointer to the block. At mounting time, we can find the last stored pointer quickly using sequential or binary search algorithms. The sequential searching reads the stored pointes from the first one until it reads a null pointer. Its time complexity is O(B_{size} / P_{size}). The binary searching divides the root block into two sub-blocks and reads the boundary pointer of these sub-blocks. If the pointer is null, this searching selects the left sub-block. Otherwise, the other one is selected. With the selected sub-block, the above procedure is repeated until the last stored pointer is found. This search algorithm has a better time complexity of O(lg(B_{size} / P_{size})).

Each pointer in the root block directs the corresponding header block as shown in Figure 5. The header blocks and their related data (marked by dotted boxes), with the exception of the last one, could be erased in advance, while the last one is always maintained in a valid state. A header block contains several snapshot headers, and a snapshot header consists of a mounting flag (*M*), an unmounting flag (*U*), and a pointer to the snapshot data block (*Pt*). Since the snapshot header size (H_{size}) is typically 4B, the maximum number of snapshot headers that can be stored in a header block is B_{size} / H_{size} (=2^{15}). The snapshot headers are sequentially stored onto the header block, and one of the two searching algorithms is used to find the last stored snapshot header in the same way as done by the root block. The pointer stored in the latest snapshot header is used to access the snapshot data that were last stored.

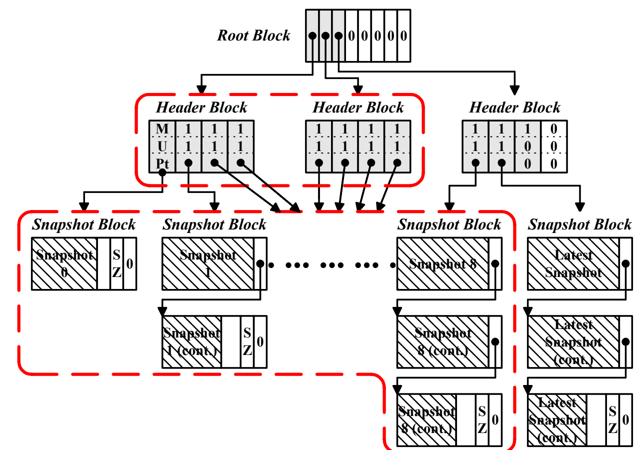

Fig. 5. Snapshot management in the proposed snapshot technique for NOR flash.

As the snapshot data size varies, the snapshot data is stored in several snapshot blocks which are managed by a linked list. These blocks are selected by considering the performed erase count in order to ensure the wear-leveling property. All snapshot blocks has a pointer to the next block, and this pointer value is zero if the block is the last one. The last snapshot block also has a size field (*SZ*) which specifies the size of the snapshot data stored in the last snapshot block as shown in Figure 5.

In summary, this technique only reads $\lg(B_{size} / P_{size})$ x P_{size} + $\lg(B_{size} / H_{size})$ x H_{size} (=92) bytes in an average case to fine the location of the last stored snapshot, providing an instant lookup time. Furthermore, this ensures the wear-leveling property due to the following two reasons. First, the root block is extremely and rarely erased in every B_{size} / P_{size} x B_{size} / H_{size} (=2^{31}) snapshot writing operations. Second, the header and snapshot blocks are selected by considering the number of performed erase operations.

After mounting the file system, when the in-memory metadata is firstly changed due to a write, we set the *M* flag of the next one of the latest snapshot header to be 1. At unmounting time, if the *M* flag is set, we store the metadata to flash memory, record the location of the stored snapshot in the snapshot header, and finally set the *U* flag to be 1. By doing this, we can check the validity of the latest snapshot by testing whether the *U* flag is set or not.

3.2 Snapshot Technique for NAND Flash

The snapshot technique that is developed for NOR flash can not be applied to NAND flash because NAND flash supports only page-based I/O. For example, when we store a pointer in the root block or a snapshot header to a header block, an internal fragmentation occurs in the page since the size of both a pointer and a snapshot header is smaller than the page size of NAND flash and thereby wasting the flash memory space significantly [11]. Note that without erasing a block which embeds the page, it is practically impossible to write data to the unused space of the page. Thus, we must specifically develop another space-efficient snapshot technique for NAND flash.

The proposed snapshot technique for NAND flash reserves the first *N* blocks in flash memory as header blocks. As shown in Figure 6, each header block is used to store the snapshot header, which consists of a mounting page and an unmounting page, and the snapshot data. When the in-memory metadata is changed due to a write operation, we select a header block in a round-robin manner and erase both the selected block and the data blocks linked by the selected block. The mounting page of the selected block keeps a special symbol (*SS*), parameter *N*, and a mounting count (*M*), which is increased by 1 at every snapshot writing operation. The special symbol means that this is a used header block and is actually stored in a spare area of the mounting page. A spare area is a small out-of-band area (e.g. 16 or 64 bytes) which is attached to each page in NAND flash.

When unmounting the file system, we store both the snapshot size (*SZ*) and data to the selected block. If the snapshot size is larger than the size of free pages in the selected block, we store the snapshot to multiple data blocks by using a linked list. A pointer (*Pt*) is used to direct the next block, and the pointer value is zero if the block is the last one. Finally, an unmounting count (*U*), which is equal to the mounting count, is stored onto the unmounting page of the selected block.

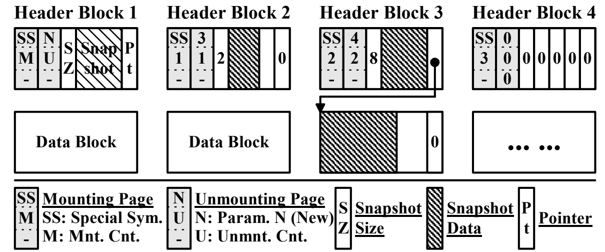

Fig. 6. Snapshot management in the proposed snapshot technique for NAND flash when the header block count is 4.

This means that the internal fragmentation occurs only on the two pages of the mounting and unmounting pages. Since the snapshot size, snapshot data, and pointer field are stored onto pages at one time, no internal fragmentation space is incurred for them.

Only one header block is erased at each snapshot writing operation despite of the snapshot size. Thus, the proposed technique ensures wear-leveling property for the reserved header blocks. However, the erase rate of the reserved header blocks can be different from that of the remaining data blocks. Thus, we adaptively control the parameter *N* by using Eq. 1 where B_{size} means the flash block size, F_{size} means flash memory capacity, and E_{count} means the total number of performed erase operations in the data blocks during the mounting time. At unmounting time after storing the snapshot in flash memory, Eq. 1 is used to calculate *N'*, and the calculated *N'* is used to control *N*. If *N'* is larger than *N*, we increase *N* by 1. Otherwise, *N* is decreased by 1. We set the range of *N* to be 2 to 10% of total block count. The modified *N* is stored onto the unmounting page of the header block, which means that the first *N* blocks are reserved as header blocks. In this way, the proposed technique for NAND flash adaptively changes the reserved area size in order to ensure the wear-leveling property.

$$\frac{1}{N'} = \frac{E_{count}}{F_{size}/B_{size} - N'} \Rightarrow N' = \left\lceil \frac{F_{size}}{B_{size}(E_{count}+1)} \right\rceil \quad (1)$$

At mounting time, we read the mounting count of all reserved header blocks. Since we do not know the *N*, we sequentially read header block from the first one and check its special symbol until we read $\max\{N\}$ = 0.1 x F_{size} / B_{size} blocks or the special symbol is not valid. We then select a block which has the highest mounting count and check whether the mounting count is equal to the unmounting count. If these counts are the same, as this means that the block has a valid snapshot, we directly use its snapshot data as an in-memory metadata. In summary, we only need to read $\max\{N\}$ x P_{size} + P_{size} bytes to find the latest snapshot in NAND flash.

3.3 Fast Mounting for Crashed File System

In the proposed snapshot techniques, if the latest snapshot is not valid, this means that the file system crashed due to power failure or operating system fault. Although recently updated data are not written to flash memory, the data stored in the flash memory are consistent because the existing flash file systems write the updated data to flash memory one by one. Thus, we can build an in-memory metadata in the same way as the existing flash file systems perform mounting operation. However, this takes up long I/O and computation times, and thus, in order to resolve this problem we develop three fast crash recovery techniques.

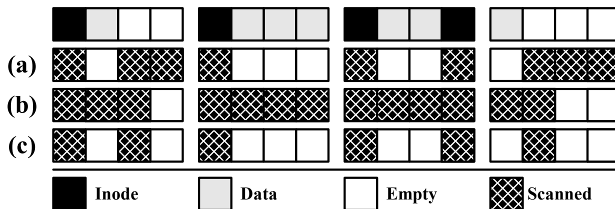

Fig. 7. Example of the proposed fast crash recovery techniques.

First, we do not scan data pages in flash memory. Since each inode embeds the location and size of the attached data pages, we can skip the scanning for the attached data pages as shown in Figure 7(a). Irrespective of this, the existing flash file systems can still read the data pages for checking CRC errors in the pages. The proposed technique performs this error checking when the data is actually accessed. Second, we do not scan empty pages in flash memory by using a sequential writing policy. Since we sequentially write data to a block from the first page, we can stop scanning for the rest of the block when we read an empty page. With this technique, we read only inode pages, data pages, and the first page of empty areas as shown in Figure 7(b). Third, combining these two techniques as shown in Figure 7(c), we only need to read the inode pages and the first page of the empty areas in flash memory in order to directly build an in-memory metadata. This significantly reduces the crash recovery time.

3.4 Fast Unmouting by Compressed Snapshot

Since at unmounting time the proposed snapshot techniques store the metadata snapshot to flash memory which provides slow write speed, the unmounting time of the proposed techniques can be delayed especially when the flash write speed is slow and the snapshot size is large. In order to reduce this unmounting time, we use a data compression technique [11] in such as way that the snapshot is compressed before storing it into flash memory. Since the time required for compressing the snapshot is quite shorter than the time required for writing the snapshot to flash memory, this snapshot compression technique can reduce unmounting time of the proposed snapshot techniques significantly. Furthermore, this compression technique reduces the flash memory space used for storing the snapshot. At mounting, the compressed snapshot is decompressed on the fly.

4. EXPERIMENTAL RESULTS

We evaluate the mounting time of the proposed techniques over an existing flash file systems of JFFS2 on an Intel machine (Pentium III 850MHz) running the latest patch version of Linux kernel 2.4. We modeled both NOR and NAND flash devices whose I/O characteristics are shown in Table 1. Based on this model, we developed virtual flash device drivers which hold CPU during the time required to perform a requested I/O operation in a real non-DMA flash device. The virtual drivers provide a common flash driver interface for the flash file systems and record the numbers of performed I/O, which are further used to analyze the I/O characteristics of the file systems. We also measured the time spent to process the flash file systems on CPU.

First, we measured the mounting time of the existing file system. As shown in Figure 8, the mounting time is directly proportional to the flash memory capacity. This is because the existing file

Table I. I/O Characteristics of Flash Memory.

Type	Model	Read		Write		Erase
NOR	Unit	2B	512B	2B	512B	128KB
	Speed	75ns	19.2μs	14μs	3.58ms	1.2s
NAND	Unit	512B		512B		16KB
	Speed	35.9μs		226μs		2ms

system scans 103% and 125% of the entire flash memory space for NOR and NAND flash memories, respectively, during the mounting. The file system scans over 100% because it reuses some pages which have been evicted from the memory cache. Also in NAND flash the ratio of scanned space is higher than that in NOR flash because NAND flash only supports page-based I/O, reading useless data.

The mounting time of NAND flash is longer than that of NOR flash because the read operation in NAND flash is 2 to 3 times slower than that in NOR flash. Even though the I/O speed of flash memory is continuously improved due to the enhancement of semiconductor technology, we believe that the I/O time required to scan the entire flash chip space will continuously take about 5 to 20 seconds because the flash chip capacity is continuously increasing by 100% every year. We also observed that the time consumed for computation, which involves classifying block type, building an inode tree, and counting nlinks value for all inodes, is less than 1 second when the stored data size is less than 1MB [8].

Then, we measured the mounting time depending on the stored data size. We set the flash capacity to be 128MB and used the Linux kernel source tree as a stored data. As shown in Figure 9, a larger stored data size incurs longer computation time, while it seldom effects the I/O time. We also found that the computation time is not dependent on the flash capacity. For example, if the flash capacity is 256MB, the computation time is almost equal to that shown in Figure 9. However, the computation time depends heavily on the computing power of CPU. As we used Pentium III in this experiment and embedded systems are typically equipped by a low-power embedded CPU, the computation time required for mounting the file system on embedded systems will be much longer than that presented in this experiment. In addition, we believe that the computation time can be partly interleaved with the I/O time if DMA I/O operations are used.

Thus, in the existing flash file system, during mounting procedure, the I/O time heavily depends on the flash memory capacity, while the computation time is greatly influenced by the stored data size. The proposed fast mounting techniques reduce the I/O time by

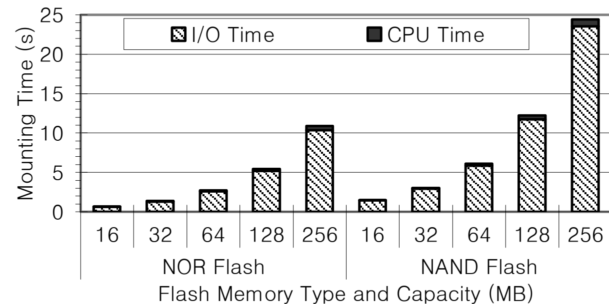

Fig. 8. Mounting time of an existing flash file system as a function of the flash memory type and capacity.

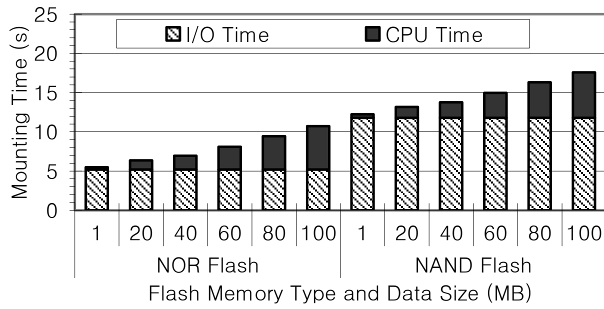

Fig. 9. Mounting time of an existing flash file system as a function of the stored data size.

simply reading the stored snapshot and the computation time by directly using the stored snapshot as an in-memory metadata. On the other hand, the existing fast mounting technique reduces the I/O time only [4].

We then measured the in-memory metadata snapshot size of the existing flash file systems in order to analyze the mounting time precisely. Figure 10 shows the size of snapshot of a 128MB NOR flash depending on the use of the snapshot compression technique. The snapshot consists of inode caches, raw inodes, and erase blocks. It shows that a larger stored data size incurs bigger physical inode and inode cache sizes. For example, when the stored data size is 100MB, the snapshot size is about 700KB. We also observed that the size of erase block metadata is directly proportional to the flash memory capacity. Similar features and size are observed in NAND flash. Fortunately, in this latter case, the snapshot size is reduced by less than half of its original when we use the *zlib* compression algorithm, used in JFFS2 and UNIX.

Second, we analyzed the mounting time of the proposed snapshot techniques with a 128MB NOR flash when a valid snapshot exists. Figure 11 shows that both I/O and computation time is reduced by over two orders of magnitude as compared with the existing system because the proposed techniques do not scan the entire flash space nor construct any in-memory metadata. Rather it directly uses the stored snapshot. The result also shows that a larger stored data size results in longer I/O time due to the snapshot size. In fact, the I/O time can be further reduced if the compression technique is used. The drawback is that, it takes a longer computation time for decompressing the snapshot. We also applied the proposed technique to a NAND flash and observed slightly longer I/O time but similar performance features.

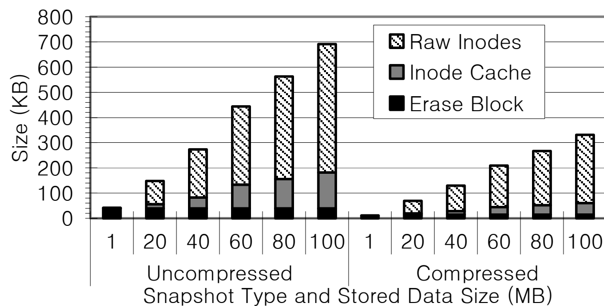

Fig. 10. Snapshot size as a function of the stored data size and the use of compression technique.

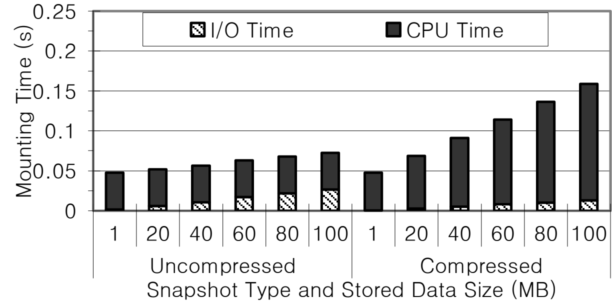

Fig. 11. Mounting time of the proposed snapshot technique in NOR flash.

Conversely, the proposed technique has a longer unmounting time than the existing system. As shown in Figure 12, the unmouting time of the proposed one is directly proportional to the stored data size because the snapshot writing operation takes a dominant part of the total unmounting time. When the snapshot is compressed, the I/O time is reduced by less than half, while the computation time is slightly increased. For example, when a 128MB NOR flash is used with 100MB of stored data, the unmounting time takes about 2.5 seconds, while that of the existing system is less than 0.5 second. Fortunately, since NOR flash is used as a code storage, its snapshot is seldom changed. Thus, it does not need to store the snapshot for a majority of the unmounting time. Even if it does need to store the snapshot, we believe that the unmouting time can be interleaved with the shutdown procedure of operating systems. Furthermore, we measured the unmounting time on a NAND flash and observed over 10 times shorter unmounting time and similar performance features because the write speed of NAND flash was over 15 times faster than that of NOR flash. Thus, in NAND flash, the unmounting time of the proposed technique is usually less than 1 second.

Third, we analyzed the wear-leveling property of the proposed snapshot techniques over the existing snapshot techniques used in LFS and FTL. In LFS, the super block reaches to the erase limit quite quickly regardless of the write traffic as shown in Figure 13(a). Figure 13(b) shows that in FTL the erase count of reserved area (marked by white background color) is different from that of the remaining data area because the size of reserved area is fixed. For example, with the light write traffic, the reserved area reaches the limit quickly. Moreover, we observed that in FTL if the stored data size is large, the reserved area reaches the limit quickly as the snapshot size is heavily dependent on the stored data size.

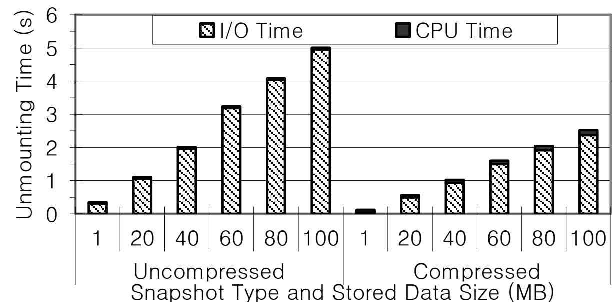

Fig. 12. Unmounting time of the proposed snapshot technique in NOR flash.

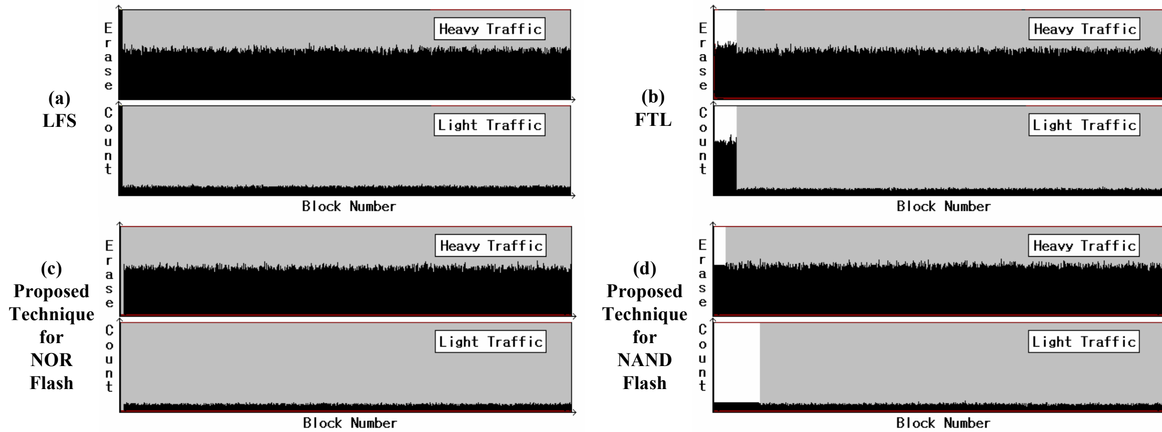

Fig. 13. Analysis of the wear-leveling property under heavy and light write traffics.

On the other hand, the proposed technique for NOR flash ensures the wear-leveling property even though the erase count of the first block is smaller than that of remaining blocks. Also the proposed technique for NAND flash ensures this property by adaptively controlling the size of reserved area depending on the amount of write traffic. When the write traffic is heavy, the reserved area size is set to be smaller, and conversely when the traffic is light, the area size is set to be larger in order to balance the wear-levels of the reserved and the remaining areas.

Finally, we evaluated the effectiveness of the proposed fast crash recovery techniques. Figure 14 shows the ratio of scanned flash area of the recovery techniques as a function of the stored data size on a 128MB NOR flash. With the data skipping technique as shown in Figure 7(a), we can skip scanning the data area, whose size is only about 30% of the originally stored data size because they are stored in a compressed form. By using the sequential writing policy as shown in Figure 7(b), we can skip scanning the empty areas that takes up a large portion of the flash space when the stored data size is small. Finally, the hybrid technique as shown in Figure 7(c) only read metadata and the first page of empty areas which usually form less than 5% of the entire flash space. The metadata consists of inodes, directory entries, and node headers. This means that when the file system crashes, the I/O time used for mounting can be reduced by more than 95%. However, these recovery techniques do not reduce the computation time, while the proposed snapshot techniques do. We also observed slightly higher scanned area ratio in NAND flash because it only provides page-based I/O.

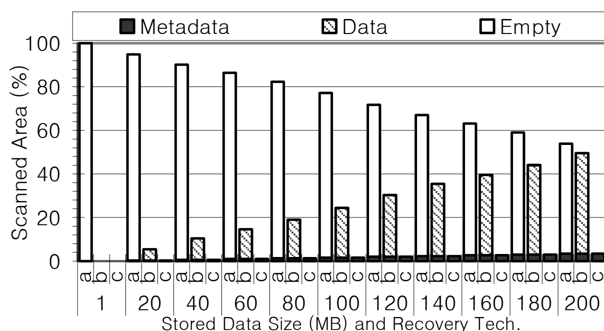

Fig. 14. Scanned area ratio of the crash recovery techniques.

5. CONCULSION

In this paper, we have presented instant mounting techniques for both NOR and NAND flash file systems. The proposed snapshot techniques store a metadata snapshot at unmounting time and reload the snapshot quickly at mounting time by overcoming the physical constraints. These techniques are intended to check the validity of the snapshot. If it is invalid, we use three fast crash recovery techniques. The experimental results have shown that the proposed techniques reduce the mounting time by over two orders of magnitude because they reduce both the I/O time (they only read a stored snapshot) and the computation time (they directly use the snapshot as an in-memory metadata). Therefore, we expect that the proposed techniques will be effective in reducing the mounting time of flash file systems and consequently the boot-up time of flash memory based computing devices.

REFERENCES

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

[2] R. Bez, E. Camerlenghi, A. Modelli, and A. Visconti, "Introduction to Flash Memory," *In Proc. of the IEEE*, Vol. 91, No. 4, pp. 489-502, April 2003.

[3] T. R. Bird, "Methods to Improve Bootup Time in Linux," *In Proc. of the Ottawa Linux Symposium (OLS)*, Sony Electronics, 2004.

[4] L.-P. Chang and T.-W. Kuo, "An Efficient Management Scheme for Large-Scale Flash-Memory Storage Systems," *In Proc. of the ACM Sym. on Applied Computing (SAC)*, pp. 862-868, 2004.

[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 Trans. on Consumer Electronics*, Vol. 48, No. 2, pp.366-375, 2002.

[6] M. Rosenblum and J.K. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Trans. on Computer Systems*, Vol. 10, No. 1, pp. 26-52, 1992.

[7] Samsung Electronics, "Advantages of SLC NAND Flash Memory," http://www.samsungelectronics.com/.

[8] U. Vahalia, *UNIX Internals*, *The New Frontiers*, Ch. 8-9, Prentice Hall Inc., 1996.

[9] D. Woodhouse, "JFFS: The Journaling Flash File System," *In Proc. of the Ottawa Linux Symposium (OLS)*, RedHat Inc., 2001.

[10] M. Wu and W. Zwaenepoel, "eNVy: A Non-Volatile, Main Memory Stroage System," *In Proc. of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 86-97, 1994.

[11] K. S. Yim, H. Bahn, and K. Koh, "A Flash Compression Layer for SmartMedia Card Systems," *IEEE Trans. on Consumer Electronics*, Vol. 50, No. 1, pp. 192-197, 2004.