Memory Management Scheme for Cost-Effective Disk-On-Modules in Consumer Electronics Devices

Young-Sik Lee, Dawoon Jung, *Student Member*, IEEE, Jin-Soo Kim, *Member*, IEEE, and Seungryoul Maeng

**Abstract — ***A Disk-On-Module (DOM) is a NAND flash memory-based device with legacy I/O interface which does not require any special device driver due to the presence of flash translation layer (FTL). FTL is an intermediate software layer which makes DOMs look like conventional hard disk drives. Since DOMs are usually used in mass-market consumer electronics devices, they are extremely cost-sensitive; hence FTL should be able to run in a severely resource-constrained environment.*

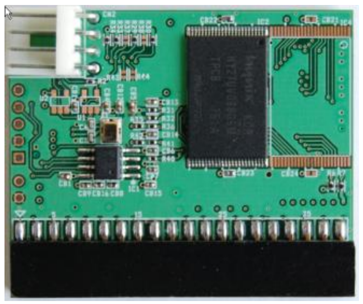
In this paper, we propose TinyFTL, a new FTL which employs an efficient memory management scheme for DOMs with a very small amount of memory. TinyFTL divides the mapping information into multiple levels and caches only recently-accessed mapping information in memory. According to experimental evaluation, TinyFTL shows the performance comparable to or better than the existing FTLs with only 4.3-6.2% of memory requirement (12KB) for 16 GB NAND flash memory. [1]

Index Terms — disk-on-module, flash translation layer, NAND flash memory, cache management

I. INTRODUCTION

NAND flash memory is being used as a primary storage medium for many consumer electronics devices such as PDAs, GPS navigation devices, MP3 players, PMPs, digital still cameras, and digital TVs. This is mainly because NAND flash memory is a small and lightweight non-volatile device which consumes low power and provides high reliability against mechanical shock, vibration, and operating temperature.

Specifically, it becomes increasingly popular for many consumer electronics devices to incorporate NAND flash memory in the form of Disk-On-Modules. A *Disk-On-Module (DOM)* is a flash drive with either USB or IDE/SATA interface to be used as a computer hard disk drive (HDD) within embedded computing systems [1]. For instance, DOMs can be used for storing maps in GPS navigation devices or saving favorite channel lists in digital TVs. Fig. 1 shows internal organizations of DOMs with ATA and USB interface. DOMs are directly plugged into the interface slot (IDE, SATA, or USB) or integrated into a target system with other on-board components.


(a) ATA-DOM

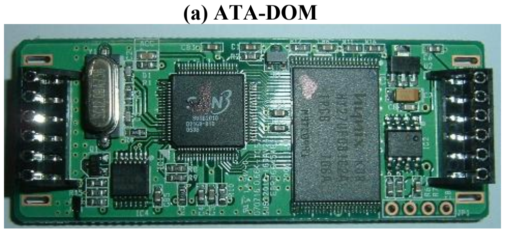

(b) USB-DOM

Fig. 1. DOMs with ATA and USB interface.

DOMs are similar to Solid State Drives (SSDs), but have a smaller form factor with less storage capacity. While SSDs are optimized for performance to be used in laptops and server systems, DOMs are mostly an integral part of mass-market consumer electronics devices, making them more cost-sensitive. Due to this reason, DOMs tend to be enormously resource-constrained in terms of processing power and memory capacity.

An important advantage of using DOMs in consumer electronics devices is that an additional or specific device driver is not needed to interface with DOMs. Instead, traditional device drivers for HDDs/CDROMs can be used for DOMs as well without any modification. This is usually achieved by an intermediate software layer called flash translation layer (FTL).

NAND flash memory has many characteristics which are different from conventional memory or HDDs. Most notably, NAND flash memory requires an erase operation to rewrite data and there is a limitation in the number of erase operations that can be performed. Moreover, the unit size of an erase operation is far larger than that of a read/write operation. Hence, NAND flash-based systems, including DOMs and SSDs, employ the flash translation layer to support legacy

[1] This work was supported by the MKE(Ministry of Knowledge Economy), Korea, under the ITRC(Information Technology Research Center) Support program supervised by the IITA(Institute of Information Technology Advancement)" (IITA-2008-C1090-0801-0020).

Young-Sik Lee, Dawoon Jung, and Seungryoul Maeng are with the Computer Science Department, Korea Advanced Institute of Science and Technology (KAIST), Daejeon 305-701, Korea (e-mail: {yslee, dwjung, maeng}@camars.kaist.ac.kr)

Jin-Soo Kim (corresponding author) is with the School of Information and Communication Engineering, Sungkyunkwan University, Suwon 440-746, Korea (e-mail: jinsookim@skku.edu)

Contributed Paper
Manuscript received March 17, 2008 0098 3063/08/$20.00 © 2008 IEEE

block device interfaces while providing higher performance. The main role of FTL is to redirect each incoming write request to a previously-erased area and to manage the internal mapping information to access the latest data on NAND flash memory. In most cases, this mapping information is kept in memory. It should be noted that the performance and the reliability of NAND flash-based systems are greatly affected by the mapping algorithm implemented in FTL.

When designing FTL, we should take hardware restriction into account as well as software performance. An FTL scheme which offers superior performance at the expense of higher resource consumption (such as memory) is not necessarily a better scheme as there is always some trade-off between performance and cost. This is especially true for DOMs which are used in mass-market consumer electronics devices. Typical DOMs are equipped with a very small amount of RAM due to cost pressure. FTL should utilize the small RAM space to store not only firmware data but also the mapping information used by FTL. Although many innovative FTL schemes are proposed in the past few years, none of them has focused on the environment where the amount of memory usable by FTL is extremely small.

In this paper, we propose a new FTL called TinyFTL which employs an efficient memory management scheme for resource-constrained DOMs. In TinyFTL, we divide the mapping information into multi-level translation units and cache only the recently-accessed translation units in RAM to reduce the memory footprint of FTL. Since we do not allocate the RAM space permanently to keep track of all the mapping information, the required RAM size is not proportional to the total NAND flash memory size. According to our experimental evaluations, TinyFTL shows the performance comparable to or better than the existing FTLs by making use of only 4.3-6.2% of memory (12 KB) for 16 GB NAND flash memory.

The rest of this paper is organized as follows. In Section II, we overview DOMs, MLC NAND flash memory, and flash translation layers. Section III presents the related work and Section IV discusses the motivation of our work. Section V describes the proposed FTL and memory management scheme in detail. In Section VI, we compare experimental results between the proposed FTL and the previous work. Finally, we conclude in Section VII.

II. BACKGROUND

A. *Disk-On-Modules (DOMs)*

Fig. 2 illustrates the block diagram of a Disk-On-Module. A DOM consists of flash controller, ROM, RAM, I/O interface controller, and one or more NAND flash memory chips. The flash controller is a programmable microprocessor whose codes are stored in ROM. The primary task of the flash controller is to run the FTL code so that the DOM can be viewed as a hard disk drive to the host system. The run-time data and the mapping information used by FTL are maintained in RAM. The DOM is interfaced with the host system via the I/O interface controller which handles various protocols such as USB, ATA, and SATA.

Since the DOM is typically used as a storage medium for mass-market products, its hardware is severely restricted. It tends to be constructed with a low-end CPU core and a tiny amount of RAM. In particular, the size of RAM is directly related to the manufacturing cost and the energy consumption, and, in many cases, less than 32 KB.

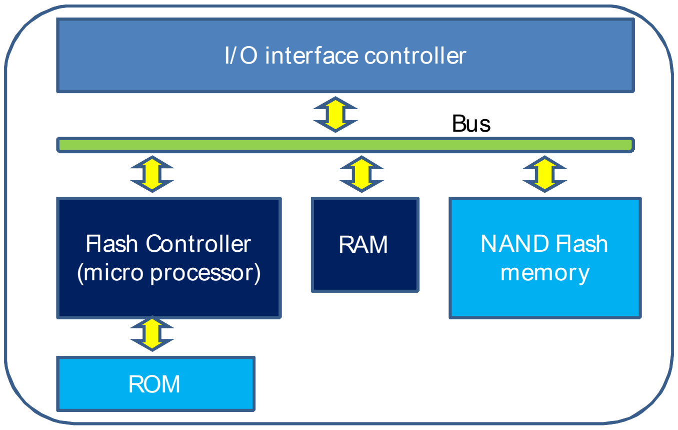


Fig. 2. The block diagram of a Disk-On-Module (DOM).

B. *MLC NAND flash memory*

NAND flash memory is a non-volatile memory which is quickly replacing HDDs in portable computing devices. Since there are no mechanical parts in NAND flash memory, it has many advantages over HDDs in terms of form factor, weight, reliability, and energy consumption.

Recently, a new type of NAND flash memory called Multi-Level Cell (MLC) NAND flash memory has been developed to provide higher density with larger storage capacity than the previous Single-Level Cell (SLC) NAND flash memory. Whereas SLC NAND flash contains only 1-bit information per cell, a cell in MLC NAND flash can represent two or more bits of information. However, MLC NAND flash has the possibility of more bit errors because of smaller error margins, necessitating the use of stronger error correction codes (ECCs). Nevertheless, many flash memory-based storage systems are using MLC NAND flash memory in favor of low cost and high density.

There are three types of operations in NAND flash memory: read, write, and erase operations. NAND flash memory reads and writes data in the unit of *page*. A page is comprised of data area and spare area; the data area size is a multiple of sector size (512-4096 B) and the spare area size is a fixed fraction (1/32) of the data area size. The spare area is used to store bookkeeping information on the page such as bad block indicators, ECCs, and the corresponding logical page numbers. The data stored in NAND flash memory cannot be updated in-place. Instead, a larger area containing the original data should be erased first before writing a new data. The unit of this erase operation is called a *block*, which usually contains 32-128 pages. To make matters worse, there is a limitation in the number of erase operations that can be performed on a single flash memory block. Due to these characteristics, it is not easy for the host system to deal with the raw NAND flash memory directly.

C. *Flash Translation Layer (FTL)*

FTL is a software layer residing in many NAND flash-based storage devices such as DOMs, SSDs, memory cards, etc. FTL is introduced to hide the unique characteristics of NAND flash memory to the host system. Without FTL, the host system requires a special NAND flash memory driver to perform read, write, and erase operations. However, FTL allows the host system to use the conventional device drivers for hard disks because FTL makes NAND flash memory look like a hard disk drive by bridging the gap between the block device interface and NAND flash interface.

One of the most important tasks in FTL is to manage the mapping from the logical page number to the physical location in an efficient way. Commonly, FTL maintains a set of free blocks which are erased in advance, and then redirects incoming write requests to those areas to reduce the number of erase operations. This results in a one-to-one correspondence between the logical address given from the host system and the physical address on NAND flash memory.

As initial free blocks are filled with new data, FTL needs to generate new free blocks to absorb future write requests. This is done by the process known as *garbage collection* (or *merge operation*). During garbage collection, FTL selects a victim block and copies all the live pages in the block to the current free block. Then, the victim block is erased and converted into a new free block.

III. RELATED WORK

Several different FTL schemes are proposed in the previous literature. According to mapping granularities, they can be classified into page-mapped FTLs and block-mapped FTLs.

In page-mapped FTLs such as DAC [2], the logical-to-physical mapping unit is a page. This is a fine-grain mapping scheme in that each page can be freely located anywhere on NAND flash memory. Page-mapped FTLs usually perform better than block-mapped FTLs due to flexibility in storage management. However, page-mapped FTLs require an extremely large RAM space to keep track of mapping information for every individual page. For 16 GB NAND flash memory with 2 KB page size, page-mapped FTLs should have at least 24 MB of memory to operate.

To reduce the size of mapping information, block-mapped FTLs are proposed in which only the block-level mapping information is maintained in memory. All the pages belonging to a logical block are mapped into the same offset in a physical block. The Logblock FTL [3] is a representative block-mapped FTL which makes use of a set of reserved *log blocks*. When there is an update in a logical block, the Logblock FTL assigns a log block to the logical block. Thereafter, all the update operations to the same logical block are logged sequentially in the corresponding log block. If all of log blocks are exhausted, the Logblock FTL reclaims one of log blocks by performing the garbage collection process. By using log blocks as temporary buffers, the Logblock FTL can delay expensive erase operations and improve the FTL performance. Note that the Logblock FTL maintains the page-level mapping information for a small number of log blocks, while the rest of data blocks are mapped with block granularity. Therefore, the Logblock FTL consumes a little more memory than pure block-mapped FTLs.

The Superblock FTL [4] aims at improving the performance of the Logblock FTL by introducing the notion of *superblocks*. A superblock is defined as a set of adjacent logical blocks. Unlike the Logblock FTL, a log block is shared by several logical blocks in the same superblock. Superblocks are mapped with block granularity, whereas pages inside the superblock can be mapped to any location in several physical blocks similar to page-mapped FTLs. This will inevitably increase the amount of mapping information, but the Superblock FTL has a sophisticated mechanism where such fine-grain mapping information is stored in the spare area of NAND flash memory. Since the spare area is limited in size, the whole mapping information is structured in multiple levels.

In terms of memory footprint, block-mapped FTLs such as the Logblock FTL and the Superblock FTL still require too much memory to be used in resource-constrained DOMs. For example, the Logblock FTL needs at least 128 KB of memory (excluding the additional space for log blocks) for 16 GB NAND flash memory which has 65,536 blocks. The Superblock FTL demands several KBs more to cache the page-level mapping information.

IV. MOTIVATION

Previously, many researchers have introduced high-performance FTLs. They assume that there is a sufficient amount of RAM to preserve at least the block-level mapping information. However, contemporary storage devices such as DOMs and memory cards cannot accommodate the whole block-level mapping information in memory as the size of RAM is limited by the lack of on-chip space and/or manufacturing cost and energy consumption constraints. Since the size of the block-level mapping information grows in proportion to the storage capacity of NAND flash memory, the continuing trend of ever increasing storage capacity simply makes the problem worse. Thus, it is necessary to develop a new FTL scheme which shows comparable performance to the existing FTLs even under very small RAM size.

Since there is not enough space to maintain the whole block-level mapping information, FTL must store a part of them whereas the rest is stored in NAND flash memory. A simple way of achieving this is to employ a *zone-switching* scheme. In the zone-switching scheme, FTL splits the whole logical space into a fixed size of consecutive areas (or zones), and loads only the mapping information of the currently accessed zone into memory. If there is a read/write request to a different zone, the current mapping information is flushed and replaced with the mapping information of the new zone. Although it is relatively easy to extend the traditional block-mapped FTLs with the zone-switching scheme, it incurs significant overhead when several different zones are actively accessed simultaneously. Unfortunately, the trace analysis results indicate that such an access pattern is frequently seen in the real workload [5].

The basic idea of the proposed TinyFTL is to cache only the recently-accessed mapping information in memory. This is not a new idea, but difficult to implement in FTL since it requires a unit structure to manipulate easily, a tag structure to identify the wanted unit fast, and an effective eviction policy. Moreover, it remains very challenging to exhibit the similar level of performance to other block-mapped FTLs even in the presence of cache miss overhead. Wu et al. have investigated the use of a cache mechanism in the address translation of FTL [6]. However, they utilize the cache only for fast access of address translation information. The existing mapping information is still stored in RAM, thus there is no advantage at all in memory footprint.

V. DESIGN

A. Overall structure of TinyFTL

Basically, TinyFTL is a block-mapped FTL where each logical block is associated with one physical block. A small portion of physical blocks are reserved for log blocks. When there are write requests in a logical block, TinyFTL allocates a log block to the logical block to absorb incoming write requests. One of the important characteristics of TinyFTL is that pages in a logical block can reside anywhere in the associated physical block or, if any, in the log block.

Fig. 3 illustrates the overall structure of TinyFTL. The block mapping table translates the logical block number to the corresponding physical block number. Then, TinyFTL finds the right location of a page with the help of the page mapping table. As shown in Fig. 3, some of logical blocks have an additional log block, which is used to log incoming write requests directed to the corresponding logical block. When the reserved log blocks are exhausted, TinyFTL starts the garbage collection process to generate a new free block.

Compared to the Logblock FTL, the major distinction of TinyFTL is that pages need not be sorted according to their logical page numbers in the physical block. This out-of-place scheme provides a greater flexibility in locating pages, improving the overall FTL performance. If a page should be in a fixed position in the physical block as is done in the Logblock FTL, more overhead is incurred to put each page in the right position during garbage collection.

In a sense, TinyFTL can be viewed as a simplified version of the Superblock FTL. Although the Superblock FTL is one of the best FTL algorithms, it cannot be used in its present form for MLC NAND-based DOMs. First, the original Superblock FTL is proposed for SLC NAND flash memory. The Superblock FTL stores the page mapping information of a superblock into the unused spare area. However, MLC NAND flash memory devotes more space for stronger ECCs due to its high error rate, making the spare area size usable by FTL become smaller. Second, the Superblock FTL is too complex to implement in resource-constrained devices, as a log block is shared by several physical blocks in the same superblock.

TinyFTL has the same benefit as the Superblock FTL by allowing pages to be freely located in the physical block. However, in order to adapt the Superblock FTL to resource-constrained DOMs based on MLC NAND flash memory, TinyFTL has simplified the Superblock FTL by restricting the superblock size to one logical block and assigning only one log block for each block. Furthermore, unlike the Superblock FTL, TinyFTL aggressively uses most of the RAM space to cache address translation information, significantly reducing the required memory size.

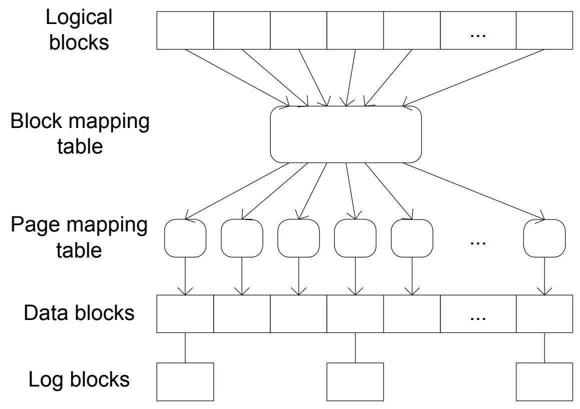


Fig. 3. The overall structure of TinyFTL.

B. Mapping structure

Although TinyFTL is a block-mapped FTL, it has the page mapping table for each logical block. Thus, the amount of mapping information is much larger than that of traditional block-mapped FTLs. As in the Superblock FTL, this mapping information needs to be squeezed into the spare area so as not to incur any additional flash operations. Since the size of the usable spare area in MLC NAND is far smaller than that in SLC NAND, the block mapping table and the page mapping table should be structured in a more compact format.

TinyFTL splits the block mapping table into block directory and block table, as it is too large to be kept in memory. This is the main difference between TinyFTL and other block-mapped FTLs such as the Logblock FTL and the Superblock FTL. A set of block mapping entries for a block table, and the location of each block table is pointed to by the block directory entry. When NAND flash memory is mounted, TinyFTL scans a reserved area called *map blocks* and restores the block directory in memory. The size of the top-level block directory is usually very small and can be kept in memory all the time. The block directory is flushed to NAND flash memory at the end of the garbage collection process, as garbage collection changes the information of the block directory.

Similarly, the page mapping table is also organized in two levels. Page table entries contain the address translation information for each logical page, and the page directory points to the location of the corresponding page table. Page directories and page tables are stored in the spare area. Fig. 4 depicts the overall mapping structure between the logical page number and the physical page number.

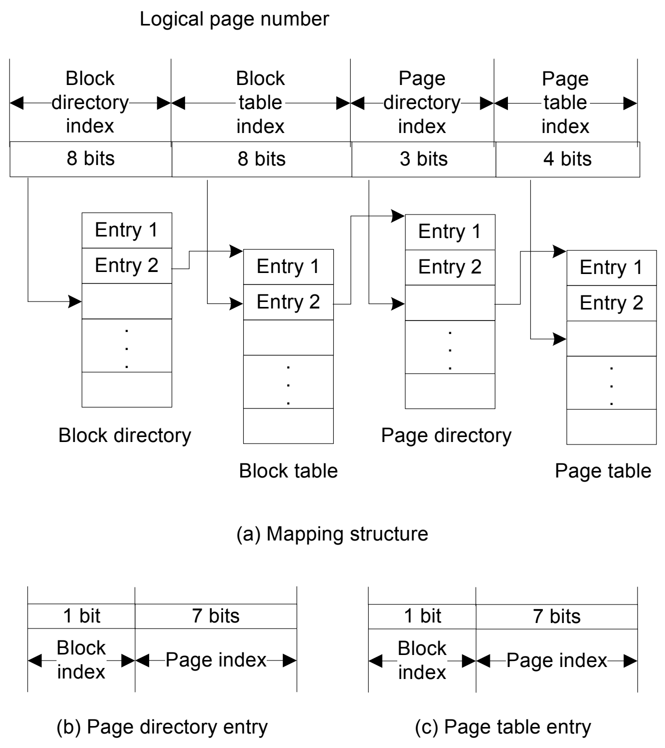


Fig. 4. The overall mapping structure and the formats of page directory entry and page table entry.

C. *Memory management*

For efficient management, TinyFTL divides the entire RAM space into four different areas: block directory cache, block table cache, page mapping cache, and metadata area.

The latest contents of the block directory are maintained in the block directory cache. The block directory resides in the RAM always because its size is very small and it is frequently accessed to find the location of the block table in the next level.

The block table cache holds a set of recently-accessed block tables. As there is no enough RAM space to store all the block tables, TinyFTL just caches a small number of block tables in memory. The block directory index is used as a cache tag to identify whether the corresponding block table resides in the block table cache. If there is no available entry in the block table cache, the victim is selected using the LRU (Least-Recently-Used) replacement policy.

The page mapping cache is used to cache the page mapping information for the given block. A pair of page directory entry and page table entry is a caching unit. TinyFTL uses the physical block number and the physical page number as a cache tag. The page mapping cache is also managed by the LRU policy.

In the metadata area, TinyFTL maintains the list of log blocks and reserved free blocks. As described in Section V.A, log blocks are used to buffer incoming write requests temporarily. The reserved free blocks are used for recovery. By using pre-assigned blocks to update the mapping information, we can simplify the recovery process when the system faces a sudden power outage. These metadata must reside in memory so as not to generate any unnecessary flash memory operation during garbage collection.

D. *Other TinyFTL features*

NAND flash memory chips have some bad blocks as shipped from the factory, and additional blocks may go bad over time if the erase count reaches a certain threshold. When a bad block is produced at run time, TinyFTL replaces the block with a reserved unused block. The list of bad blocks is stored with the block directory to save the flash memory space.

TinyFTL also provides a recovery mechanism when the mapping information is vanished by sudden power failure. Recall that the contents of block directory and block tables are stored at the end of garbage collection. The page directory and page tables are also stored simultaneously in the spare area of the page being updated. Although the latest information of the block directory and cached block tables are lost, we can recover all the information by checking reserved free blocks.

E. *TinyFTL configuration*

In this paper, we assume the MLC type of 16 GB NAND flash memory where each page consists of 2 KB data area and 64 bytes spare area, and each block has 128 pages. We assign 256 entries to the block directory and 256 entries to each block table as shown in Fig. 4(a). The page mapping information for a block is divided into 8 different page tables; a page directory has 8 entries and each page table consists of 16 entries.

In order to store the list of log blocks with the block directory in one page, we utilize the maximum 393 log blocks. The list of log blocks is linked in the LRU order. We have the total 1,536 non-data blocks by reserving 24 blocks for every 1,024 blocks. Among them, 393 blocks are used as log blocks and 256 blocks are reserved to store the block directory. The remaining 877 blocks are used for bad block management.

Table I summarizes the total memory usage of TinyFTL. As shown in Table I, the total memory required by TinyFTL is only 12,195 bytes under the aforementioned configuration.

TABLE I
THE AMOUNT OF MEMORY REQUIRED BY TINYFTL

Information	Size (bytes)	Description
Block directory cache	768	256 entries
Block table cache	6,171	8 tables with liked list pointers
Page mapping cache	3,970	128 entries with linked list pointers
Log block list Reserved free blocks	1,280 6	393 blocks with linked list pointers 3 blocks
Total	12,195	

VI. EVALUATION

A. Experimental Environment

We have implemented a trace-driven simulator for TinyFTL. To compare the performance with previous work, we have also simulated the original Logblock FTL, the Logblock FTL with zone-switching (labeled as Logblock-zoned), and the Superblock FTL. The simulator counts the number of additional read, write, and erase operations needed to process incoming update requests. It also measures the number of additional flash memory operations to maintain the mapping information for Logblock-zoned and TinyFTL. Note that the original Logblock FTL and the Superblock FTL do not incur any additional flash operations to manage the mapping information as these FTLs store the whole mapping information in memory. These counts are translated into the total processing time of FTL using the timing parameters shown in Table II [7]. For all the simulated FTLs, the number of log blocks is set to 393 blocks.

TABLE II
TIMING PARAMETERS OF MLC NAND FLASH MEMORY

Operation	Access time (μ s)
Data read (2KB data area)	112.8
Spare read (64 B spare area)	61.6
Write (2 KB + 64 B one page)	852.8
Erase (256 KB one block)	1,500.0

Table III summarizes the real workload traces used for the evaluation. These traces are extracted from a Windows-based desktop computer by using DiskMon [8]. PIC, MP3, and MOV model the typical usage scenario for DOMs or flash memory cards used for digital still cameras, MP3 players, and PMPs. These traces are obtained from 8 GB FAT32 file system. Other traces, GENERAL and WEB, represent the scenario of PDAs, smart phones, and laptops. GENERAL and WEB traces are obtained from 16 GB NTFS file system. The number of write requests and a brief description of each trace are shown in Table III.

TABLE III
WORKLOAD TRACES USED

Trace	# of r/w requests	Description
PIC	474,385	Copy and delete picture files repeatedly. Average file size is 1.9 MB.
MP3	512,762	Copy and delete MP3 files repeatedly. Average file size is 4.4 MB.
MOV	435,872	Copy and delete movie files repeatedly. Average file size is 681 MB.
GENERAL	1,019,078	General PC usage for 5 days.
WEB	124,762	Web surfing activity for 1 day.

B. Memory Footprint

We first measure the memory requirement of each FTL and the results are shown in Table IV. For 16 GB MLC NAND flash memory whose block size is 256 KB (128 pages), there are 65,536 blocks. Each block mapping entry requires 2 bytes to represent the corresponding physical block number. Therefore, the Logblock FTL requires 128 KB to maintain the block mapping table in memory. In addition to that, the Logblock FTL keeps track of page-level mapping information for log blocks, requiring another 150,912 bytes for 393 log blocks.

In the Superblock FTL, each block mapping entry consumes 3 bytes; 2 bytes for the physical block number of page directory plus 1 byte for the offset of page directory in the block. Thus, the size of the block mapping table in the Superblock FTL is 192 KB, slightly larger than that in the Logblock FTL. Moreover, the Superblock FTL requires another 1,568 bytes to cache the recently-accessed page mapping information under the default cache configuration (16 cache entries).

On the other hand, TinyFTL requires only about 12 KB RAM size (cf. Table I) since it does not have to maintain the whole block mapping information in memory. We also present the memory footprint of the Logblock FTL with zone-switching (Logblock-zoned) for comparison. Although many different configurations for the zone size are possible in Logblock-zone, we set the zone size to 6,144 blocks so that the block mapping information of each zone occupies almost the same memory size as in TinyFTL.

TABLE IV
THE MEMORY REQUIREMENT OF FTLS

FTL	Memory Requirement (bytes)
Logblock	281,984
Logblock-zoned	12,288
Superblock	198,176
TinyFTL	12,195

C. Overall Performance

Fig. 5 compares the overall performance of FTLs. The metric we use is the normalized processing time. The processing time represents the total elapsed time to replay read/write requests for a given trace. In addition to the actual time to read and write requested pages, the processing time includes the overhead for garbage collection as well as the time for managing the mapping information. In Fig. 5, the processing time is normalized with respect to the time taken by the Logblock FTL.

In general, Logblock-zoned has more overhead than the original Logblock FTL because Logblock-zoned needs to perform zone switching whenever a new zone is accessed. The zone switching involves flushing and reloading of zone mapping information. Under the configuration presented in Section V.E, the size of zone mapping information is 12 KB. Therefore, each zone switching involves at least 6 page write operations (for flushing) and 6 page read operations (for reloading). Due to this zone switching overhead, the processing time of Logblock-zoned is increased by 10.7% and 4.9% for GENERAL and WEB traces respectively, compared to the Logblock FTL. Note that Logblock-zoned has only marginal overhead for PIC, MP3, and MOV traces. This is due to the high locality in zone accesses resulted from sequential writes of large files. In fact, the MOV trace copies and deletes movie files whose average size is 681 MB (cf. Table III). In this extreme case, there is virtually no performance difference between FTLs.

Except for the MOV trace, TinyFTL improves the total processing time of the Logblock FTL by 6.2% (MP3) - 16.6% (WEB). Compared to Logblock-zoned which is configured to use the same amount of memory, TinyFTL reduces the processing time by up to 20.5%.

From Fig. 5, we can observe that the Superblock FTL performs best among all the FTLs evaluated. However, the Superblock FTL requires about 200 KB to run. In case of PIC, MP3, and MOV traces, TinyFTL shows comparable performance to the Superblock FTL using only 6.2% of memory.

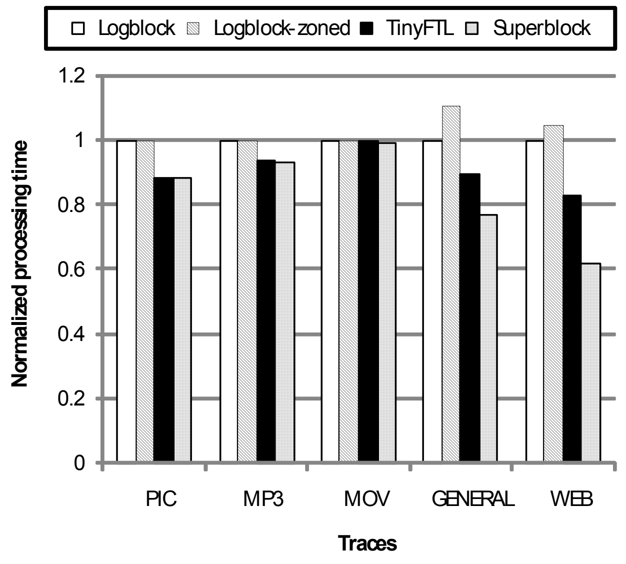


Fig. 5. A performance comparison of FTLs. Logblock indicates the original Logblock FTL, while Logblock-zoned extends the Logblock FTL with zone-switching (zone size = 6,144 blocks). Superblock represents the Superblock FTL and TinyFTL is our proposed FTL.

D. The Effect of Block Table Cache Size

In TinyFTL, the block table cache is used to accelerate accesses to block table entries. We have varied the block table cache size from 1 to 64 caching units to investigate its impact on the cache miss ratio. As described in Section V.C, each caching unit caches the entire contents of a block table and occupies 768 bytes. The simulation results are illustrated in Fig. 6.

We can see that FAT32 file system traces (PIC, MP3, and MOV) exhibit notably small cache miss ratios. As described in the previous subsection, this is because the working set sizes of those traces are small enough to fit in the block table cache. On the contrary, cache miss ratios of NTFS file system traces (GENERAL and WEB) are relatively high, reaching up to 5%, and they are continuously benefited from the increased block table cache size. The storage access patterns of GENERAL and WEB traces have many small-sized requests and they tend to touch much wider area at the same time. Hence, the working sets cannot be accommodated in the block table cache.

The default block table cache size we use in Fig. 5 is 8 units. Fig. 6 indicates that as we increase the block table cache size further, we can expect much more improvement in the processing time especially for GENERAL and WEB traces.

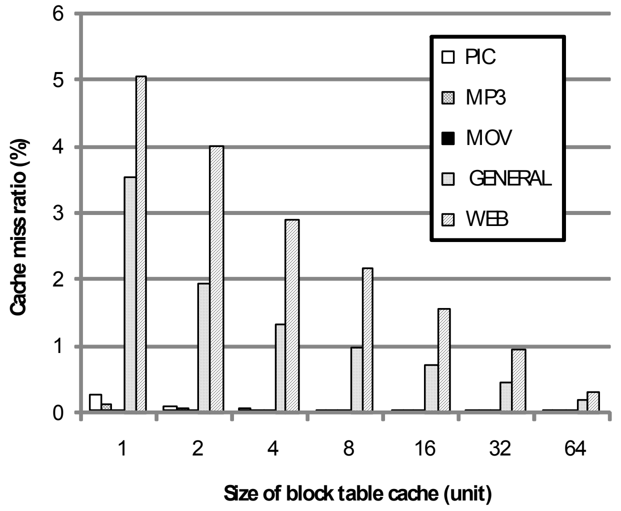


Fig. 6. Changes in the cache miss ratio when we vary the block table cache size from 1 to 64 units. Each unit holds the contents of a block table and occupies 768 bytes.

E. The Effect of Page Table Cache Size

We have also investigated the impact of the page table cache size on the overall performance. Similar to Fig. 6, we have measured the cache miss ratio when we vary the page table cache size from 8 to 512 units. Each page table cache unit consumes 24 bytes consisting of a page directory entry and a page table entry. The default page table cache size we use in Fig. 5 is 128 units.

As in the block table cache, FAT32 file system traces (PIC, MP3, and MOV) have lower cache miss ratios than NTFS file system traces (GENERAL and WEB). However, the overall cache miss ratio is higher than that of block table cache, even for PIC, MP3, and MOV traces. This is because the LRU replacement policy works poorly under large sequential access patterns. While a block table cache entry can cover the area of 64 MB, a single page table cache entry can cover only the area of 32 KB. Thus, when a large area is sequentially updated, most of cache entries will be evicted to accommodate newly referenced entries. GENERAL and WEB traces have many small-sized random requests, but the number of large-sized requests is not negligible as well, both of which contribute to poor locality in the page table cache.

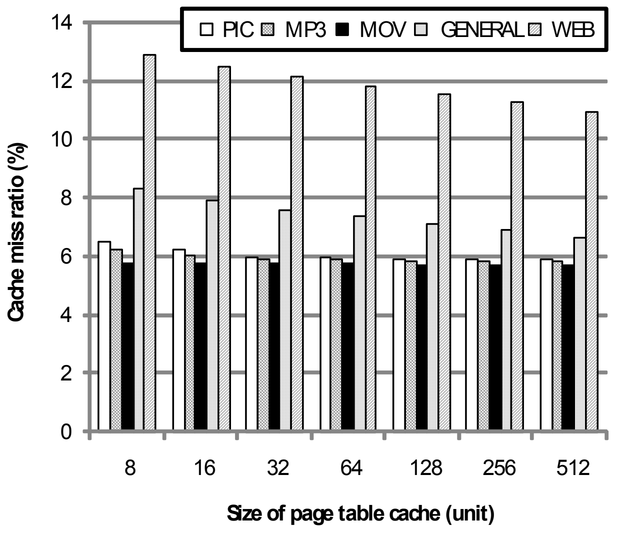


Fig. 7. Changes in the cache miss ratio when we vary the page table cache size from 8 to 512 units. Each unit represents single page mapping information consisting of a page directory entry and a page table entry, and occupies 24 bytes.

VII. CONCLUSION

This paper presents the design and implementation of TinyFTL for resource-constrained Disk-On-Modules (DOMs). Against other block-mapped FTLs, TinyFTL can work with a very small amount of memory due to an efficient memory management scheme, being a perfect solution for DOMs used in mass-market consumer electronics devices.

The basic idea behind TinyFTL is to load only the necessary mapping information on demand instead of storing the whole mapping information in memory all the times. Special cares have been taken to avoid performance degradation. First, TinyFTL allows pages in a logical block to be located anywhere in the associated physical block or, if any, in the log block for flexible storage management. Second, TinyFTL divides the mapping information into multiple levels, and stores them in the spare area of NAND flash memory so as not to incur any additional flash memory operations. Finally, most of the memory is used as a cache for recently-accessed mapping information, whose size can be adjustable depending on the workload characteristics and hardware configurations.

Our evaluation with the real workload traces demonstrates that TinyFTL shows the performance comparable to or better than the existing block-mapped FTLs using only 12 KB of memory for 16 GB NAND flash memory. This amount of memory corresponds to 4.3-6.2% of the memory footprint required by traditional block-mapped FTLs.

Our future work includes the further optimization of TinyFTL by improving the cache management policy for page table cache. We also plan to extend TinyFTL to other flash storage devices constructed with different architecture and/or different type of NAND flash memory, as flash memory technology is evolving fast.

REFERENCES

[1] *Disk on module*, http://en.wikipedia.org/Disk_on_module
[2] M.-L. Chiang, P. C. H. Lee, and R.-C. Chang, "Using data clustering to improve cleaning performance for flash memory," *Software Practice and Experience*, vol. 29, no. 3, pp.267-290, 1999.
[3] 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. Consum. Electron.*, vol. 48, no. 2, pp. 366-375, May 2002.
[4] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee, "A superblock-based flash translation layer for NAND flash memory," in *Proc. of the 6th ACM & IEEE International conference on Embedded software (EMSOFT '06)*, 2006.
[5] A. Riska and E. Riedel, "Disk drive level workload characterization," in *Proc. of USENIX Annual Technical Conference*, 2006
[6] C.-H. Wu, T.-W. Kuo, and C.-L. Yang, "A space-efficient caching mechanism for flash-memory address translation," in *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC '06)*, 2006
[7] Samsung Elec. *2G x 8 Bit NAND Flash Memory(K9GAG08UXM)*, Dec. 2006
[8] *DiskMon*, http://technet.microsoft.com/en-us/sysinternals/bb896646.aspx.

Young-Sik Lee received the B.S. degree in Computer Science from Korea Advanced Institute of Science and Technology (KAIST) in 2006. He is currently working toward his Ph.D. degree in Computer Science at KAIST. His research interests include flash memory, embedded systems, file systems, and operating systems.

Dawoon Jung (S'07) received his B.S. and M.S. degrees in Computer Science from Korea Advanced Institute of Science and Technology (KAIST) in 2002 and 2004, respectively. Currently, he is pursuing his Ph.D. degree in Computer Science at the same school. His research interests include operating systems and storage systems.

Jin-Soo Kim (M'89) received his B.S., M.S., and Ph.D. degrees in Computer Engineering from Seoul National University, Korea, in 1991, 1993, and 1999, respectively. He is currently an associate professor in Sungkyunkwan University. Before joining Sungkyunkwan University, he was an associate professor in Korea Advanced Institute of Science and Technology (KAIST) from 2002 to 2008. He was also with the Electronics and Telecommunications Research Institute (ETRI) from 1999 to 2002 as a senior member of research staff, and with the IBM T. J. Watson Research Center as an academic visitor from 1998 to 1999. His research interests include embedded systems, storage systems, and operating systems.

Seungryoul Maeng received the B.S. degree in Electronics Engineering from Seoul National University, Korea, in 1977, and the M.S. and Ph.D. degrees in Computer Science from KAIST in 1979 and 1984, respectively. Since 1984 he has been a faculty member at KAIST. From 1988 to 1989, he was with the University of Pennsylvania as a visiting scholar. His research interests include computer architecture, cluster computing, and embedded systems.