μ-FTL: A Memory-Efficient Flash Translation Layer Supporting Multiple Mapping Granularities*

Yong-Goo Lee, Dawoon Jung, Dongwon Kang, and Jin-Soo Kim
Computer Science Division
Korea Advanced Institute of Science and Technology (KAIST)
Daejeon 305701, Korea
{yglee,dwjung,dwkang}@camars.kaist.ac.kr, jinsoo@cs.kaist.ac.kr

ABSTRACT

NAND flash memory is being widely adopted as a storage medium for embedded devices. FTL (Flash Translation Layer) is one of the most essential software components in NAND flash-based embedded devices as it allows to use legacy files systems by emulating the traditional block device interface on top of NAND flash memory.

In this paper, we propose a novel FTL, called μ-FTL. The main design goal of μ-FTL is to reduce the memory footprint as small as possible, while providing the best performance by supporting multiple mapping granularities based on variable-sized extents. The mapping information is managed by μ-Tree, which offers an efficient index structure for NAND flash memory. Our evaluation results show that μ-FTL significantly outperforms other block-mapped FTLs with the same memory size by up to 89.7%.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management—*Garbage Collection*

General Terms

Management, Measurement, Performance, Design

Keywords

NAND flash memory, flash translation layer (FTL), address translation

1. INTRODUCTION

As mobile embedded devices such as cellular phones, digital cameras, and MP3 players become increasingly popular, demand for NAND flash memory is growing rapidly. The trend is expected to accelerate in the foreseeable future due to the recent proliferation of NAND flash-based Solid State Disks (SSDs), which are quickly penetrating desktop and server markets.

NAND flash memory is usually used as a storage medium in place of Hard Disk Drive (HDD) because of its non-volatility and large I/O unit. Since NAND flash memory has no mechanical parts, it has a lot of advantages compared to HDDs such as short read and write latency, low power consumption, small and lightweight form factor, and solid state reliability.

In spite of these strengths, it is not straightforward to replace HDDs with NAND flash memory due to its *erase-before-write* nature; if data are once written in NAND flash memory, it cannot be overwritten until the area containing the original data is erased. To make matters worse, the erase unit (called *block*) is larger than the read and write unit (called *page*) by 32 – 128 times.

In order to make NAND flash memory look like a traditional block device, a software layer called *Flash Translation Layer* (FTL) is used to conceal those unfavorable characteristics from host systems. Normally, FTL redirects incoming write requests from a host into an empty area on NAND flash memory, and keeps track of the mapping information from the logical address used by the host into the physical address in NAND flash memory. The performance of FTL considerably varies depending on how to manage such logical-to-physical address mapping. In addition, the amount of RAM required by a particular address mapping scheme is another concern in designing FTLs as it is directly related to cost and energy consumption of embedded devices.

Most FTLs can be categorized into two classes according to their mapping granularities: *page-mapped* FTLs and *block-mapped* FTLs. A page-mapped FTL literally maps a logical address into a physical address in a page unit. It is highly flexible as a logical page can be written to any physical page in NAND flash memory, and thus there is much room for improving the performance of FTL. However, the amount of mapping information, which has significant impact on the RAM footprint of FTL, becomes enormously large as each page needs its own mapping entry. On the other hand, the mapping unit of a block-mapped FTL is a block. This type of FTL requires much smaller amount of mapping information than a page-mapped FTL does. However, a logical page can no longer reside anywhere in flash memory, but can only be written to the designated page offset within a physical block. Due to this restriction, block-mapped FTLs tend to incur more overhead.

Most write requests of realistic workloads exhibit either

*This work was supported by the IT R&D program of MKE/IITA. [2006-S-040-03, Development of Flash Memory-based Embedded Multimedia Software]

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, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EMSOFT'08, October 19–24, 2008, Atlanta, Georgia, USA.
Copyright 2008 ACM 978-1-60558-468-3/08/10 ...$5.00.

small and random or *large and sequential* characteristics [5]. For example, write requests originating from file system metadata operations or temporary files downloaded while browsing the Internet consist of small and random access patterns. On the other hand, copying or downloading multimedia files generate large and sequential write requests covering a wide range of the logical address space. A block-mapped FTL can deal with these large and sequential write requests effectively, but pretty poor in handling small and random write requests. Thus, it is desirable to support multiple mapping granularities not only for reducing the size of mapping information but also for enhancing the performance of FTL.

In this paper, we propose a novel FTL called μ-FTL (*mu*-FTL or *m*inimally *u*pdated-FTL). The characteristics of μ-FTL can be summarized as follows. First, μ-FTL dynamically adjusts mapping granularities according to the size of write requests by maintaining mapping information in an *extent*-based μ-Tree structure [8]. μ-Tree is a variant of B$^+$-Tree, offering an efficient indexing scheme on NAND flash memory. μ-FTL makes use of coarse-grain mapping granularities for large and sequential write requests, while fine-grain mapping granularities for small and random requests. Second, only a small and fixed-size RAM is used as a cache for mapping entries and per-page validity information, hence the run-time RAM footprint of μ-FTL is bounded to this cache size. Finally, μ-FTL divides the whole logical address space into a set of non-overlapping partitions and forwards incoming write requests to different physical blocks depending on the partitions they belong to. This is effective because each partition has a different degree of "hotness". Our experimental results show that μ-FTL outperforms other block-mapped FTLs with the same memory size by up to 89.7%.

The rest of the paper is organized as follows. Section 2 overviews the characteristics of NAND flash memory, flash translation layer (FTL), and μ-Tree. Section 3 summarizes related work. In Section 4, the detailed design of μ-FTL is discussed. Section 5 presents performance evaluation results, and Section 6 concludes the paper.

2. BACKGROUND

2.1 NAND Flash Memory Characteristics

A NAND flash memory device is composed of a number of blocks, and a block in turn consists of a number of pages. The page is a unit of read and write operations, while the block is a unit of erase operations. Each page has an additional *spare area*, which typically contains housekeeping information such as error correction code (ECC) and the associated logical page number.

Currently, two types of NAND flash memory are being widely used: SLC (Single-Level Cell) NAND [3] and MLC (Multi-Level Cell) NAND [4]. In SLC NAND, the page size is 2KB and a block consists of 64 pages. The recently introduced MLC NAND expands its capacity by storing two bits per each memory cell. Accordingly, the page size of MLC NAND is doubled to 4 KB, and the number of pages in a block is also increased to 128. Although the operational latency of MLC NAND is much longer than that of SLC NAND, a large portion of NAND flash-based storage available today are based on MLC NAND due to its low cost-per-bit. For this reason, we only focus on MLC NAND flash memory in this paper. Table 1 compares the characteristics of SLC and MLC NAND flash memory.

Table 1: The characteristics of SLC and MLC NAND flash memory

	read latency	write latency	erase latency
SLC NAND [3]	77.8μs (2KB)	252.8μs (2KB)	1500μs (128KB)
MLC NAND [4]	165.6μs (4KB)	905.8μs (4KB)	1500μs (512KB)

2.2 Flash Translation Layer (FTL)

FTL bridges the semantic gap between block device interface and flash memory operations. FTL allows legacy disk-based file systems to be used without any modification, by emulating the traditional block device interface on top of NAND flash memory.

Two major factors that affect the performance of FTL are *address mapping mechanism* and *garbage collection policy*. The address mapping mechanism hides the erase-before-write nature of NAND flash memory, by redirecting each write request to other free space and setting up a mapping entry between the logical page address and the physical page address. When there is no enough free space, a process called garbage collection is invoked to reclaim invalid (out-dated or dead) pages. The garbage collection policy first selects a victim block, copies its valid (up-to-date or live) pages to other block, and eventually generates a new free block by erasing the victim block.

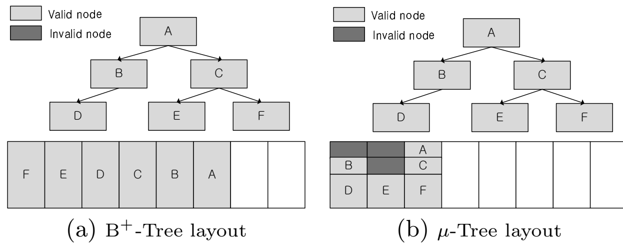
According to mapping granularities, FTLs are classified as either page-mapped or block-mapped. A page-mapped FTL maintains a logical-to-physical mapping on a page basis. Although page-mapped FTLs have potential to improve the overall performance due to its flexibility, the amount of mapping information becomes extremely large by keeping a mapping entry for every single logical page. When a mapping entry occupies 4 bytes, the required size reaches up to 32MB for 32GB MLC NAND flash memory. Even worse, page-mapped FTLs usually keep track of per-page validity information for all physical pages to isolate invalid pages from valid ones during garbage collection. This information further increases memory pressure of page-mapped FTLs.

For block-mapped FTLs, the total amount of mapping information is relatively small because it is sufficient to keep mapping entries on a block basis. For 32GB MLC NAND, the amount of mapping information is reduced to 256KB. However, block-mapped FTLs incur more garbage collection overhead as a logical page cannot be located outside the boundary of the associated physical block. Especially, block-mapped FTLs are known to suffer from a considerable performance degradation for the workload consisting of small and random write requests.

Several researchers have proposed hybrid mapping schemes which integrate the advantages of page-mapped FTLs and block-mapped FTLs. The hybrid mapping scheme complements block-level mapping with the use of page-level mapping in a limited way. Several examples of hybrid mapping schemes will be discussed in Section 3.

2.3 μ-Tree

μ-Tree [8] is a variant of B$^+$-Tree, tailored to the characteristics of NAND flash memory. B$^+$-Tree is a balanced search tree, where keys and pointers to children in the next



(a) B^+-Tree layout (b) μ-Tree layout

Figure 1: B^+-Tree vs. μ-Tree

level are stored in interior nodes and all records are stored in leaf nodes. A simple way of implementing B^+-Tree on NAND flash memory is to use each page as a node for storing keys and records (Fig. 1(a)). In this case, however, if there is an update on a record in the leaf node, all the ancestor nodes should be also updated since in-place update is not allowed in NAND flash memory. This results in flash write operations as many as the height of B^+-Tree.

μ-Tree avoids this *wandering tree* problem by allowing multiple nodes along the path from the root node to the leaf node to be in a single page (Fig. 1(b)). Consequently, any insertion, deletion, or update in the leaf node normally requires only one flash write operation in μ-Tree.

A small amount of in-memory cache can accelerate the operation of μ-Tree, further reducing the number of flash read and write operations. In the current version of μ-Tree, a read cache and a write cache are separately maintained on a page basis and those caches are consulted first before any read or write request reaches NAND flash memory. If the read cache is full, the least recently used (LRU) page is discarded from the cache. On the other hand, all the pages in the write cache are written in bulk into NAND flash memory in case the write cache becomes full. It is recommended to refer to [8] for further details on μ-Tree.

3. RELATED WORK

Before we discuss related work, we clarify several terminologies used throughout the paper. Basically, FTLs manage three kinds of physical blocks: *data block*, *update block*, and *free block*. A data block is usually full of valid pages, but may have invalid pages if those pages have been updated. An update block denotes the physical block being used as a buffer for incoming write requests. The update block may have free pages as well as valid and invalid pages. Finally, a free block consists of free pages only.

DAC [6] is a representative page-mapped FTL. DAC groups all physical blocks into several regions, and clusters data with a similar update frequency into the same region. This is based on the observation that separating hot data from cold data has a beneficial effect on the performance of systems having out-of-place update characteristics [12, 7]. Although DAC shows performance that is hard to beat in many workloads, a significantly large amount of RAM is required to maintain mapping entries for all logical pages.

The log block scheme [10] refines the pure block-mapped FTL by letting an update block (called log block) act like a buffer for a particular logical block. For a given write request, the log block scheme allocates an update block to the corresponding logical block. Thereafter, subsequent write requests to the same logical block are directed to the update block. A problem in the log block scheme is that update blocks can be merged too early before they are fully used, especially when there are many small and random writes.

FAST [11] is proposed to increase the utilization of update blocks. Unlike the log block scheme where a single update block is dedicated to a specific logical block, a global update block is shared among all logical blocks. This strategy maximizes the utilization of an update block, but makes the garbage collection process more complex as each page in an update block may come from a different logical block.

The superblock scheme [9] introduces the notion of *superblock* to exploit the block-level temporal and spatial locality. A superblock consists of a set of adjacent logical blocks. In the superblock scheme, the block mapping is still used at the superblock level, while logical pages within a superblock can be freely located in one of the physical blocks allocated to the superblock. The superblock scheme makes use of spare areas in NAND flash memory to store page-mapping information so as not to incur any additional overhead in terms of space and flash memory operations. However, the limited spare area size actually prevents the superblock size from exceeding a certain threshold. Although the superblock scheme outperforms many existing block-mapped FTLs, the mapping scheme is not as flexible as page-mapped FTLs and its performance still lags behind DAC.

Chang and Kuo [5] have proposed a flexible management scheme based on two different kinds of binary trees: *LC tree* for address translation and *PC tree* for garbage collection. The LC tree provides multiple mapping granularities to reduce the amount of mapping information. However, the RAM usage still reaches up to 17MB for 16GB flash memory without any caching mechanism. In order to bound the RAM usage, they divide both logical and physical address space into several groups, and define a *virtual group* as a pair of a logical group and a physical group. PC trees and LC trees are independently maintained for each virtual group, and some of them are cached in RAM. This caching mechanism, however, incurs a severe cache miss penalty because all spare areas in the cache-missed group should be read from flash memory at run time. In μ-FTL, only a few page reads are sufficient to locate the cache-missed tree node since the whole mapping information is organized in a more structured way on NAND flash memory using μ-Tree. Moreover, the caching mechanism in μ-FTL is more effective as only the frequently accessed nodes are cached in memory.

4. μ-FTL

4.1 Design Goals

The design goal of μ-FTL is to reduce the RAM usage as small as block-mapped FTLs, while providing the performance comparable to page-mapped FTLs such as DAC. This seems to be impossible at first as there is always tradeoff between the RAM usage and the performance of FTL. μ-FTL achieves this goal by combining the following innovative solutions.

First, the basic idea behind μ-FTL is to use coarse-grain mapping granularities for large and sequential write requests, allowing to use fine-grain mapping granularities, if necessary, for small and random write requests. The decision on the right mapping granularity is not statically determined by the logical address. Instead, μ-FTL adjusts mapping granularities according to the size of incoming write requests

adaptively. In most cases, a substantial portion of the logical address space is expected to be covered with large and sequential write requests, hence the amount of mapping information is kept low similar to other block-mapped FTLs. Only those requests, that are small in size and scattered randomly all over the entire address space, will be managed by fine-grain mapping.

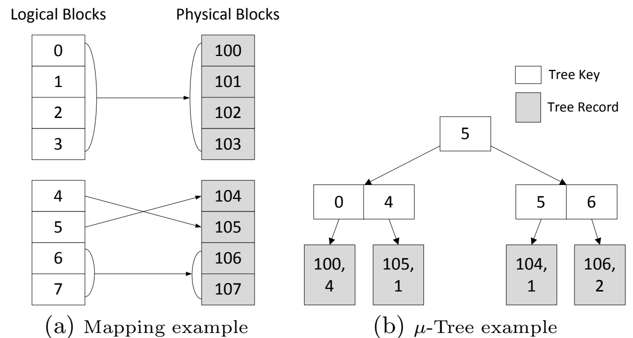
Most traditional FTLs keep track of logical-to-physical mapping information using a table. Although the table is a simple and easy-to-build data structure which allows for $O(1)$ lookup performance, it has a problem that the mapping granularity should be fixed either as a page or as a block. In order to support multiple mapping granularities, a more flexible data structure, such as tree or hash table, is necessary. μ-FTL takes advantage of μ-Tree, which is best suited as an efficient index structure for NAND flash memory. Similar to B$^+$-Tree, μ-Tree organizes the whole mapping information in a structured way on NAND flash memory so that any lookup/insertion/deletion/update operation can be performed in $O(log\ n)$.

Another important feature of μ-FTL is that the required RAM size is independent of the amount of mapping information. In μ-FTL, all the mapping information is stored in NAND flash memory as a form of μ-Tree, and only the frequently-accessed mapping entries are cached in the RAM. Therefore, one can control the total RAM usage easily by limiting the cache size.

Second, μ-FTL handles *per-page validity information* efficiently by storing them also in μ-Tree. Per-page validity information (or *bitmap* information) is needed during garbage collection for distinguishing valid pages from invalid pages in a victim block. The easiest way of checking the validity of each physical page, without any additional data structure, is to put a logical page number in the spare area of the physical page and to lookup μ-Tree to see if the logical page is still mapped to the particular physical page. However, this policy incurs a significant overhead because all pages in the victim block have to be checked individually through μ-Tree. Another possible way is to keep this information in RAM, but it occupies the precious RAM space and its size is in proportion to the total NAND flash memory size. For 32GB MLC NAND flash memory, the total amount of bitmap information reaches up to 1MB (with using 1 bit per physical page).

Although maintaining bitmap information in μ-Tree simplifies overall design, we notice the μ-Tree cache is polluted by this information, with the cache hit ratio of mapping entries severely impaired. To remedy this problem, we devise a separate *bitmap cache*, which buffers updates of bitmap information before applying them to μ-Tree. The bitmap cache is implemented using a hash table, as bitmap updates are sparse over the physical address space.

Finally, μ-FTL statically divides the logical address space into several *partitions*. An interesting observation is that a part of the logical address space written by large and sequential writes is *cold* (i.e., rarely updated), while a part of the logical address space touched by small and random writes tends to be *hot* (i.e., frequently updated). This is because file systems allocate different portions of the logical address space for different purposes. For example, the logical space allocated to large multimedia files will be rarely updated, while the logical space allocated to file system metadata or temporary Internet files will be frequently overwritten.



(a) Mapping example (b) μ-Tree example

Figure 2: An example of multiple mapping granularities

In μ-FTL, each partition has its own update block and incoming write requests are forwarded to different update blocks depending on the partitions they belong to. This is effective in preventing various data which show different degree of "hotness" from being mixed in the same physical block.

4.2 Address Mapping

4.2.1 Extent-based mapping

Each mapping entry in μ-FTL maps various numbers of pages from 1 to 128 (the block size). Fig. 2 illustrates an example of using multiple mapping granularities in address mapping. In Fig. 2(a), we assume a block is composed of four pages. Each small box represents a page, and each number in a white and a gray box indicates the logical page number (LPN) and the physical page number (PPN), respectively. Each arrow depicts a logical-to-physical mapping. We can see that LPN 0–3 are mapped at block granularity, while LPN 4 and LPN 5 are mapped at page granularity. Two pages starting from LPN 6 are mapped to two consecutive physical pages starting from PPN 106.

Fig. 2(b) shows the corresponding μ-Tree which represents the address mapping information of Fig. 2(a). LPNs are used as keys (white boxes in Fig. 2(b)) and each record (gray box in Fig. 2(b)) at the lowest level indicates the starting PPN and the length of the mapped pages. For example, the μ-Tree record <100, 4> pointed to by a key 0 represents that four adjacent logical pages LPN 0–3 are mapped to physically contiguous pages in PPN 100–103.

We call this variable-sized mapping entry an *extent*. An extent e can be represented as a tuple $e = < l, p, n >$, where l denotes the first LPN, p indicates the first PPN corresponding to l, and n is the number of pages mapped by this extent. As shown in Fig. 2(b), l is used as a key for μ-Tree and the record pointed to by l stores the mapping information on p and n.

Fig. 3 depicts the detailed key and record structures of an extent used in μ-Tree. The first bit of the key distinguishes mapping entries from bitmap entries (cf. Section 4.3). The rest of the key is used to describe the LPN (l). The first 25 bits of a record represent the PPN (p), which can specify 2^{25} different pages (corresponding to 128GB). The remaining 7 bits in the record represent the length (n) of an extent. The maximum extent length is 128 pages, which is identical to the number of pages in a block in MLC NAND flash memory.

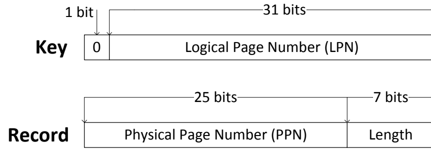


Figure 3: The key and record structures of an extent

Note that an extent cannot cross physical block boundaries. Although the structures shown in Fig. 3 are designed to fit into 32 bits, they can be extended to 40 bits or further for larger size of NAND flash memory.

4.2.2 Handling read requests

μ-FTL receives a read request in the form of $r =< L, N >$, where L represents the starting page number and N is the number of pages to read[1]. In order to service the read request, μ-FTL first searches for an extent $e =< l, n, p >$ such that $(l = L) \vee (l < L \wedge L < (l+n))$. From the extent e, μ-FTL can locate n physical pages containing the requested logical pages. In case $(l + n) < (L + N)$, the successive extents next to e are visited until μ-FTL retrieves the remaining $(L + N) - (l + n)$ pages.

4.2.3 Handling write requests

Handling write requests is somewhat complicated as they involve updates to existing entries in μ-Tree. For a given write request $w =< L, N, D >$, where the meanings of L and N are the same as in read requests and D denotes data to write, μ-FTL carries out the following steps.

First, μ-FTL writes requested data D to the update block allocated to the partition corresponding to the logical page number L. Let N_f be the number of free pages in the update block. If the number of free pages is not sufficient to accommodate all the requested data, i.e., $N_f < N$, another update block is allocated to the partition and μ-FTL writes the remaining data D' to the new update block by issuing $w' =< L + N_f, N - N_f, D' >$. This is repeated until all the requested data are written to update blocks. If μ-FTL is running out of free blocks, garbage collection is invoked (cf. Section 4.3).

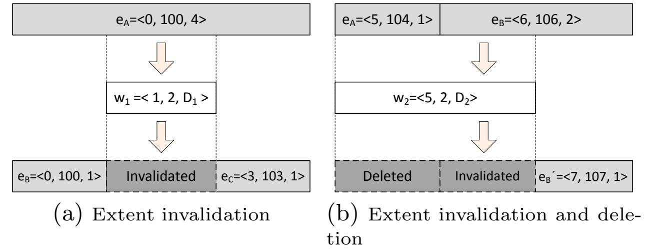
The second step is to modify the old mapping information stored in μ-Tree. Assume that, as a result of the previous step, μ-FTL wants to insert a new extent $e =< l, p, n >$ into μ-Tree. Let $e_i =< l_i, p_i, n_i >$ be another extent stored in μ-Tree. There are several different cases we need to consider:

Case 0 $(l \leq l_i \wedge (l_i + n_i) \leq (l + n))$: This is the simplest case. The old extent e_i is just deleted from μ-Tree.

Case 1 $(l_i < l \wedge (l + n) < (l_i + n_i))$: This is the case where e is included in e_i. Since a part of e_i is being updated, the updated region should be invalidated by splitting e_i into $e_l =< l_i, p_i, (l - l_i) >$ and $e_r =< (l+n), (p_i + l + n - l_i), (l_i + n_i - l - n) >$.

Case 2 $(l_i < l \wedge (l_i + n_i) \leq (l + n))$: In this case, the right-hand side of e_i is being updated as a result of e. Hence, the original e_i is modified to $e_i' =< l_i, p_i, (l - l_i) >$, reducing the extent length by $(l_i + n_i - l)$.

[1]In fact, the actual read or write operation in the block device interface specify the logical *sector* number and the number of *sectors* as parameters. In this paper, we assume for ease of exposition that they are translated into the logical page number and the number of pages, respectively.



(a) Extent invalidation (b) Extent invalidation and deletion

Figure 4: Examples of extent invalidation and deletion

Case 3 $(l \leq l_i \wedge (l+n) < (l_i + n_i))$: This is similar to Case 2 except that the left-hand side of e_i is affected by e. The old extent e_i is updated to $e_i' =< (l + n), (p_i + l + n - l_i), (l_i + n_i - l - n) >$.

Fig. 4 shows examples of extent invalidation and deletion. In Fig. 4(a), the previous extent $e_A =< 0, 100, 4 >$ is split into $e_B =< 0, 100, 1 >$ and $e_C =< 3, 103, 1 >$ due to the new write request $w_1 =< 1, 2, D_1 >$ (Case 1). In Fig. 4(b), the extent $e_A =< 5, 104, 1 >$ is deleted (Case 0), and the extent $e_B =< 6, 106, 2 >$ is updated to $e_B' =< 7, 107, 1 >$ (Case 3) by the write request $w_2 =< 5, 2, D_2 >$. Note that changing e_B to e_B' is actually implemented by first deleting e_B from μ-Tree, and then inserting a new extent e_B' into μ-Tree.

As the third step, new extents containing the latest mapping information are inserted into μ-Tree. The number of newly inserted extents is the same as the number of update blocks used in the first step.

During the insertion of new extents, an extent merge could occur if a new extent $e =< l, p, n >$ and one of the existing extents $e_i =< l_i, p_i, n_i >$ in the same physical block are not only logically but also physically contiguous, i.e., if $(l_i + n_i = l) \wedge (p_i + n_i = p)$. The new extent e is not merged with the extent e_i such that $l < l_i$ in any case, since they cannot be physically consecutive due to the restriction that pages are always written sequentially from the first page to the last page within an update block.

Finally, as the new data are written, μ-FTL updates the bitmap information to mark the pages containing the previous data as invalid. The details on managing the bitmap information are described in the following subsection.

4.3 Garbage Collection

4.3.1 Garbage collection policy

μ-FTL initiates the garbage collection process when the number of free blocks becomes two. Due to the possibility of write cache flush in μ-Tree during garbage collection, μ-FTL needs to reserve at least two free blocks all the time.

μ-FTL selects a victim block among data blocks (including μ-Tree blocks) which has the largest number of invalid pages. Valid pages in the victim block are copied into free pages in the associated update block. If there is not enough free space in the update block, μ-FTL allocates a reserved block as a new update block for the partition and continues to copy valid pages. Eventually, the victim block is erased and reclaimed by μ-FTL. μ-FTL repeats this process until the number of free blocks becomes larger than or equal to the predefined value (currently, it is set to 12).

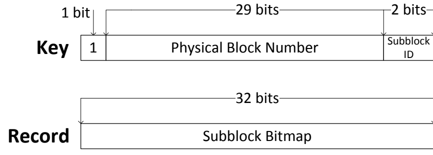


Figure 5: The key and record structures of a bitmap

4.3.2 Garbage collection information

To facilitate the garbage collection process, μ-FTL maintains two kinds of information: per-block invalid page counter and bitmap information (per-page validity information).

The per-block invalid page counter maintains the number of invalid pages in each physical block. This is used by μ-FTL to select a victim block quickly. Without this information, μ-FTL would require to count the number of invalid pages from the bitmap information for every physical block, which is an apparently time-consuming task. μ-FTL assigns 1-byte counter for each physical block, which takes only 64KB when 32GB MLC NAND flash memory is assumed.

The bitmap information, which is necessary to identify valid pages during garbage collection, is stored in μ-Tree as mentioned in Section 4.1. Fig. 5 represents the key and record structures of a bitmap entry. The first bit of the key is set to '1' to distinguish it from extents. We need 128 bits to build the full bitmap for a data block, but the record size of μ-Tree is configured to be 32 bits. Thus, we divide a data block into 4 subblocks of the same size (32 pages each), and allocate a μ-Tree entry for each of them. Accordingly, the lowest 2 bits in the key specify the subblock, and the next 29 bits point to the physical block.

In order to reduce the number of bitmap entries, μ-FTL stores bitmap entries only for the subblocks which have at least one invalid page. In other words, the bitmap for a fully valid subblock is not stored in μ-Tree. If the search for a subblock fails, all the pages in the subblock are considered valid. The measurement result shows that only less than 16% of data blocks have one or more invalid pages for all traces, and bitmap entries for those blocks occupy less than 20% of μ-Tree entries.

A write request inevitably accompanies the invalidation of physical pages that contain the previous data. When a physical page is invalidated, μ-FTL first searches for the corresponding bitmap entry in μ-Tree. If successful, μ-FTL updates the bitmap entry. Otherwise, a new bitmap entry is created and then inserted into μ-Tree. When a block is erased, at most four bitmap entries are deleted from μ-Tree, as the block does not have invalid pages any more.

4.3.3 Bitmap cache

After inserting bitmap entries in μ-Tree, we have suffered from a significant performance degradation. The overall management overhead is roughly doubled compared to when μ-Tree consists of extents only. We find out that the cache overhead caused by μ-Tree's page reads (on read cache miss) and page writes (on write cache flush) has been significantly increased. We have analyzed the cache overhead further according to the record type and it turns out that mapping entries and bitmap entries almost equally contribute to the overhead. Specifically, almost half of the μ-Tree cache is occupied by extents, while the rest is taken by bitmap entries, even if only less than 20% of μ-Tree entries are bitmap entries.

This is because μ-FTL always updates both one or more extents and a similar number of bitmap entries in μ-Tree for every write request. As a result, the μ-Tree cache, a precious resource, is always contended by extents and bitmap entries. To solve this problem, we have focused on the following characteristics of bitmap updates.

- We do not have to write the latest bitmap information in μ-Tree whenever a page is invalidated, because the information is only needed when the garbage collection process is initiated.

- Within a certain time interval, a very limited number of physical blocks receive bitmap updates. Namely, bitmap updates are *sparse* over the whole physical address space.

Based on these observations, we have devised *bitmap cache*, a hash table structure for buffering bitmap updates before applying them to μ-Tree. The hash table is a suitable data structure for indexing a widely distributed, but sparsely accessed bitmap entries with the limited number of hash table entries.

Because a write request usually invalidates several consecutive physical pages at once, we use an *invalid extent* as a hash table entry which encodes the invalidation of at most 128 contiguous physical pages using only 32 bits. An invalid extent consists of a physical page number (25 bits) indicating the first physical page of a particular invalidation and its length (7 bits). Note that the organization of an invalid extent is exactly the same as the record structure of an extent. The difference is that an invalid extent represents physically adjacent invalid pages, but an extent denotes logically and physically contiguous valid pages.

The remainder of the target physical block number divided by the number of hash buckets is used as a hash key. This hash function maps adjacent physical blocks to different hash buckets, thus can reduce the number of hash collisions when several physical blocks are sequentially invalidated.

When a few consecutive physical pages are invalidated due to a write request, the corresponding invalid extent is inserted into the bitmap cache. During the insertion, the merge operation between invalid extents may occur as in the case of an extent insertion (cf. Section 4.2.3).

Invalid extents are deleted from the bitmap cache in the following three cases. (1) When the size of the bitmap cache reaches the predefined limit, all invalid extents in the bitmap cache are flushed to μ-Tree and deleted. (2) When a victim block is selected by the garbage collection process, all invalid extents in the chain of the victim block's hash bucket are flushed to μ-Tree and deleted. (3) Lastly, as soon as all the pages in a block become invalid, the block is erased and the invalid extent of the block is deleted from the bitmap cache, without being flushed to μ-Tree. Such a block can be detected instantly by the per-block invalid page counter.

4.4 Logical Address Space Partitioning

As briefly mentioned in Section 4.1, different portions of the logical address space exhibit different access patterns. For example, an area for file system metadata (FAT area in the FAT file system or Master File Table zone in the NTFS file system) is located in the designated region of the logical

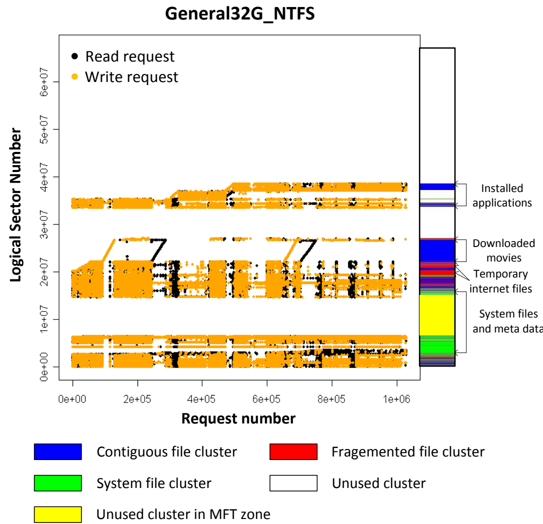


Figure 6: A disk access pattern of a general PC usage trace

address space. A part of the logical address space could be used for downloading large multimedia files such as mp3's and movies. In some cases, a user would be able to explicitly divide the logical address space into several disk partitions for different usages.

Fig. 6 depicts a disk access pattern of general PC usage gathered from a 32GB HDD with the NTFS file system during five days. Fig. 6 shows which logical sectors are read or written as time goes by; the x-axis of the graph represents a request number and the y-axis denotes the corresponding logical sector number. The rectangle in the right side represents the final organization of disk clusters. This information is obtained using DiskView [1], which shows not only the characteristics of each disk cluster, but also the file name associated with the disk cluster.

From Fig. 6, we can see that contiguous file clusters are mostly for movies and installed applications. This region tends to be sequentially written. On the other hand, the majority of fragmented file clusters are from temporary Internet files. The bottom region of the address space includes system files and file system metadata.

The region containing contiguous file clusters tend to be cold as they are rarely overwritten. On the contrary, the regions for fragmented file clusters and system file clusters are usually hot since temporary Internet files and file system metadata are updated frequently.

Traditional page-mapped FTLs do not exploit this hot-coldness existing in each region of the logical address space, as all write requests are redirected to the same global update block. Instead, μ-FTL divides the whole logical address space into several partitions. An update block is allocated for each partition to prevent data in different partitions from being mixed in the same physical block. An update block for each partition is lazily assigned only when a write request actually arrives at the partition.

4.5 μ-FTL Design Summary

Fig. 7 presents the overall architecture of μ-FTL. For handling a write request, μ-FTL writes data to the corresponding partition's update block (step (1)), updates extents in

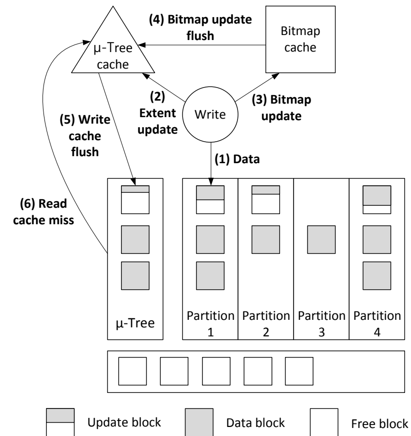


Figure 7: The overall architecture of μ-FTL

μ-Tree (step (2)), and inserts invalid extents into the bitmap cache (step (3)). The bitmap cache flushes bitmap updates to the μ-Tree cache when it becomes full or a victim block is selected by the garbage collection process (step (4)). μ-Tree flushes pages if the write cache becomes full (step (5)), and reads pages when a read cache miss occurs (step (6)).

5. EXPERIMENTAL EVALUATION

5.1 Evaluation Methodology

We have implemented trace-driven simulators for μ-FTL and other FTLs such as the log block scheme, FAST, the superblock scheme, and DAC. We assume MLC NAND flash memory is used whose page size is 4KB and the number of pages in a block is 128. The simulators count the number of page reads, page writes, and block erases while replaying a given trace. The actual time needed for performing such operations is calculated based on the timing parameters shown in Table 1.

We have used real workload traces for performance evaluation. The total six traces are obtained from a Microsoft Windows-based laptop computer by using DiskMon [2]. Half of them (PIC, MP3, and GENERAL) model the workload of embedded multimedia devices and the others (WEB, GENERAL, and SYSMARK) represent the workload of typical PC usage scenarios. For multimedia traces, we create a separate 8GB of FAT32 disk partition, and only the requests coming to the partition are collected. For PC traces, a single NTFS disk partition is used which also contains operating systems and pre-installed applications. Table 2 summarizes the characteristics of each trace used in this paper.

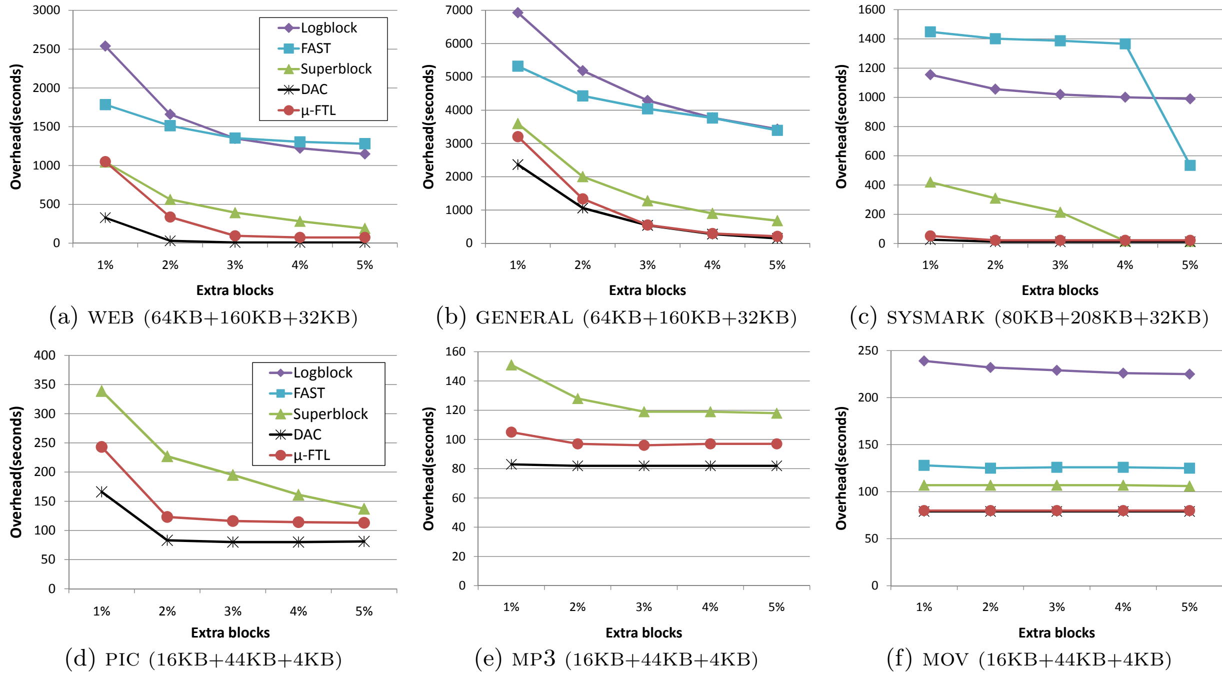
Unless otherwise stated, all experiments are performed in the following conditions. The partition size is 256 MB (512 logical blocks). The bitmap cache size is set to 32KB for PC traces and 4KB for multimedia traces. About 3% of the total physical blocks are reserved for *extra blocks*. Extra blocks are internally used by FTL as update blocks and they are not visible to the host system. When initializing a simulation, we set all extra blocks as free blocks, and other physical blocks as fully valid data blocks.

5.2 The Comparison of FTL Performance

Fig. 8 compares the performance of five different FTLs (the log block scheme, FAST, the superblock scheme, DAC,

Table 2: The characteristics of multimedia and PC traces

Trace	Storage Size	Description	# of Req's
PIC	8GB	We copy 6GB of pictures into the disk, and then repeat the process of deleting and copying 2GB of pictures 10 times. (avg. file size = 1.9MB)	465,268
MP3	8GB	The scenario is exactly the same as in PIC. In this trace, mp3 files are used instead of pictures. (avg. file size = 4.4MB)	457,836
MOV	8GB	The scenario is exactly the same as in PIC. In this case, movie files are used instead of pictures. (avg. file size = 681MB)	430,025
WEB	32GB	This trace is gathered from one-day long web surfing.	200,606
GENERAL	32GB	This trace is obtained from a 5-day long general PC usage. The trace includes office works, downloads, web surfing, installations, etc.	1,029,053
SYSMARK	40GB	This trace is collected from SYSmark 2007 Preview. The benchmark includes e-learning, office works, video creation, and 3D modeling.	158,591



(a) WEB (64KB+160KB+32KB) (b) GENERAL (64KB+160KB+32KB) (c) SYSMARK (80KB+208KB+32KB)

(d) PIC (16KB+44KB+4KB) (e) MP3 (16KB+44KB+4KB) (f) MOV (16KB+44KB+4KB)

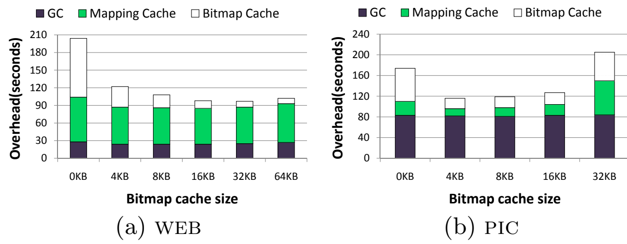
Figure 8: A performance comparison of FTLs. The values (A+B+C) in the parenthesis represent the size for per-block invalid page counter (A), the μ-Tree cache size (B), and the bitmap cache size (C) used in μ-FTL.

and μ-FTL) by changing the portion of extra blocks from 1% to 5%. The y-axis in Fig. 8 represents the management overhead calculated from additional read, write, and erase operations requested by an FTL (not by the host). The management overhead actually means the additional time required by FTL for garbage collection and cache management. All the evaluations in this paper use the management overhead as a performance metric, as many previous work did [6, 9, 10]. The results of the log block scheme and FAST are omitted in PIC and MP3 since they show very poor performance in these traces.

We have configured μ-FTL so that its RAM usage is comparable to other block-mapped FTLs. We assume that each mapping entry requires 4 bytes in block-mapped FTLs. Since block-mapped FTLs require one mapping entry for each block, the total amount of RAM needed is 64KB for 8GB, 256KB for 32GB, and 320KB for 40GB MLC NAND flash memory.

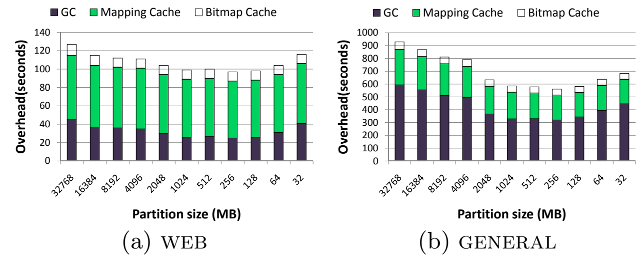
In Fig. 8, we labeled each trace with values of the form (A+B+C) to represent the specific RAM usage of μ-FTL. Each value, A, B, and C, represents the amount of RAM allocated to the per-block invalid page counter, the μ-Tree cache, and the bitmap cache, respectively. For example, μ-FTL uses 16KB for the per-block invalid page counter, 44KB for the μ-Tree cache, and 4KB for the bitmap cache with the PIC trace. The total size, 64KB in this case, is equal to the amount of RAM used by block-mapped FTLs for 8GB storage size.

We can see that μ-FTL demonstrates almost the same or similar performance as DAC with 3% or more extra blocks, even if the memory footprint of DAC is extremely large. In PIC and MP3 traces, μ-FTL is slightly defeated by DAC mainly due to the increased cache misses in the μ-Tree cache and the bitmap cache. There is not much locality in these traces compared to the number of extents they generate, as they delete and copy a large amount of files repeatedly. The



(a) WEB (b) PIC

Figure 9: The effect of bitmap cache size



(a) WEB (b) GENERAL

Figure 10: The effect of partitioning

same scenario is also applied to MOV, but the cache is used more effectively in MOV as most extents have the full size (128 pages) and many bitmap entries are removed from the bitmap cache before being written to μ-Tree. This allows μ-FTL to exhibit almost the same performance as DAC with only 1/128 of the RAM usage.

The SYSMARK trace randomly accesses much larger range of the logical address space than other traces. DAC and μ-FTL incur almost no overhead even with 1% extra blocks owing to their fine-grain mapping granularities (cf. Fig. 8(c)). For the superblock scheme and FAST, the amount of extra blocks should be larger than 4%–5% to accommodate the entire working set properly.

Overall, μ-FTL outperforms the superblock scheme, which shows the best performance among block-mapped FTLs, by 76.1%, 56.8%, and 89.7% for WEB, GENERAL, and SYSMARK traces respectively, and by 40.5%, 19.3%, and 25.2% for PIC, MP3, and MOV traces respectively, when 3% of the total physical blocks are reserved as extra blocks.

5.3 The Effect of Bitmap Cache Size

Fig. 9 shows the impact of bitmap cache on the overall performance in WEB and PIC traces, when we vary the bitmap cache size up to 64KB. In Fig. 9, the overhead is further categorized to the mapping cache overhead, the bitmap cache overhead, and the garbage collection overhead. The mapping cache overhead and the bitmap cache overhead denote the time for handling read cache misses and write cache flushes in μ-Tree caused by extents and bitmap entries, respectively. The garbage collection overhead represents page reads, page writes, and block erases occurred during the garbage collection process.

Since the total amount of RAM is fixed in Fig. 9, the larger the bitmap cache becomes, the smaller the mapping cache becomes. Therefore, the overhead is improved as we increase the bitmap cache size, but it starts to grow beyond 32KB for WEB and 8KB for PIC due to the increase in the mapping cache overhead. All PC traces show the best performance at the bitmap size of 32KB, while all multimedia traces except for MOV at 4KB. At those configurations, using the bitmap cache improves the overall performance by 52.5% in WEB and by 33.3% in PIC. The MOV trace is an exception which incurs no bitmap cache overhead even without the bitmap cache. This is because it is highly likely in MOV that all pages in a block are invalidated simultaneously, making it unnecessary to insert any bitmap entry to μ-Tree.

The bitmap cache overhead is closely related to the number of bitmap entries, which in turn depends on the total storage capacity and the characteristics of the target workload. Our measurement results indicate that it is necessary to allocate roughly 1KB to the bitmap cache per 1GB storage capacity. In case the workload's access pattern is predominantly sequential as in PIC, MP3, and MOV, it may be considered to reduce the bitmap cache size to 512 bytes or less per 1GB storage capacity.

5.4 The Effect of Partitioning

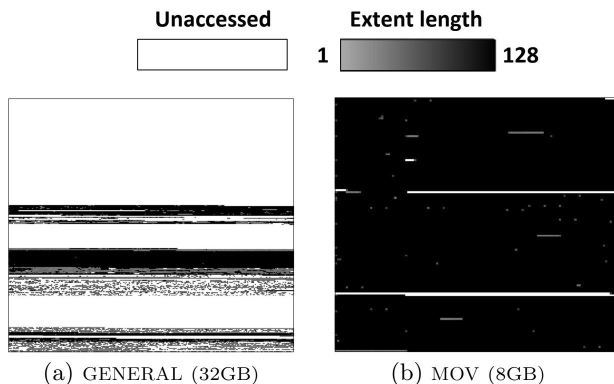
Fig. 10 illustrates the effect of logical address space partitioning in WEB and GENERAL traces. These traces are obtained from 32GB storage, hence the partition size of 32,768MB at the leftmost position corresponds to the case where the partitioning is not used at all.

When the partition size is 256MB, both traces show the best performance. At this partition size, the overall performance is improved by 23.6% in WEB and 39.6% in GENERAL. We can observe that most of the improvement comes from the significant decrease in the garbage collection overhead.

Several different factors affect the garbage collection overhead either positively or negatively depending on the partition size. As the partition size decreases, hot data can be more precisely separated from cold data. Thus, the number of hot blocks, whose pages tend to become invalid shortly, is increased. When a block becomes fully invalid, the block is immediately erased and reclaimed by μ-FTL. This is often called *switch merge* [10] and is the most inexpensive way to generate a new free block. As more free blocks are generated by the switch merge, the actual garbage collection can be postponed much longer and the number of valid pages in a victim block becomes smaller.

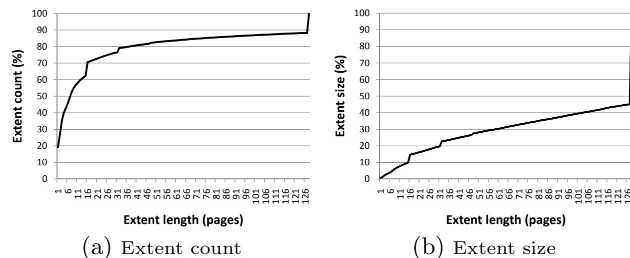
For instance, let us consider a situation when we change the partition size from 32GB to 256MB in GENERAL. In this case, the number of switch merge has increased by 17.4% (from 18,363 to 21,550), which reduces the number of garbage collection by 29.3% (from 1,109 to 784). Since the garbage collection process is invoked less frequently, the number of blocks erased during garbage collection is reduced by 34.6% and the number of valid pages, which each victim block owns, is decreased from 30.0 pages to 23.0 pages. Accordingly, the garbage collection overhead is improved by 46.0% from 594 seconds to 321 seconds.

On the other hand, if the partition size becomes too small, the garbage collection overhead increases. The smaller the partition size is, the more update blocks are needed, and finally the utilization of update blocks decreases. The low utilization of update blocks makes μ-FTL invoke the garbage collection process more frequently. When we reduce the partition size further from 256MB to 32MB in GENERAL, the number of garbage collection is increased by 5.8% and the average number of valid pages in a victim block grows again to 29.6 pages. Due to these negative effects, the garbage collection overhead rises by 38.6%.



(a) GENERAL (32GB) (b) MOV (8GB)

Figure 11: The distribution of the average extent length over the logical address space



(a) Extent count (b) Extent size

Figure 12: The cumulative distribution of the extent count and its size (GENERAL)

For almost all traces we have examined, the performance has been maximized at the partition size of 256MB, with both factors being balanced.

5.5 Extent Length Distribution

Finally, we investigate whether μ-FTL effectively adjusts the extent length according to the different access pattern. Fig. 11 depicts the distributions of the extent length over the logical address space after replaying GENERAL and MOV traces. We plot the average extent length of a logical block in a two-dimensional graph. The darker a color is, the longer an extent length is. The left-bottom pixel corresponds to the first logical block and the right-top corresponds to the last logical block. A logical block number first increases horizontally, and then vertically.

In GENERAL, the logical blocks corresponding to contiguous file clusters (cf. Section 4.4) have long extents (denoted as black in Fig. 11(a)). On the other hand, the logical blocks associated with fragmented file clusters and system file clusters have much shorter extents (denoted as gray in Fig. 11(a)). In case of MOV, however, almost all logical blocks have the maximum extent length (128 pages).

Fig. 12 displays the cumulative distributions of the number of extents and the amount of space covered for a given extent length in the GENERAL trace. From Fig. 12(a), we can see that more than 50% of the total extents have the lengths less than or equal to 8 pages. Only about 10% of the total extents have the maximum extent length. However, in terms of the extent size (extent length $\times$ extent count) shown in Fig. 12(b), more than 50% of the logical address space is covered by the full-sized extents, and only less than 10% of the address space is mapped by the extents whose lengths are less than 16 pages.

From the above results, we can confirm that a considerable range of the logical address space is covered by coarse-grain extents, while a portion of the address space requiring fine-grain mapping granularities is mapped by a number of small extents. By supporting multiple granularities, μ-FTL not only enhances the performance of FTL, but also reduces the amount of mapping information.

6. CONCLUSION

Due to the widespread use of NAND flash-based devices such as cellular phones, digital cameras, MP3 players, flash memory cards, and SSDs, the importance of developing an efficient FTL cannot be overstated.

In this paper, we propose μ-FTL, a novel FTL which incurs low management overhead with a small and fixed memory. By supporting multiple mapping granularities based on extents, μ-FTL not only enhances the performance but also reduces the amount of mapping information. The mapping information is managed by μ-Tree, which provides an efficient index structure for NAND flash memory. We also devise the bitmap cache and manage the logical address space with several different partitions to accelerate the performance of μ-FTL further.

Our experimental results show that μ-FTL significantly outperforms other block-mapped FTLs with the same RAM usage. In many cases, μ-FTL shows almost the same performance as DAC, one of page-mapped FTLs whose RAM usage is extremely large.

7. REFERENCES

[1] *DiskView*, http://technet.microsoft.com/en-us/sysinternals/bb896650.aspx.

[2] *DiskMon*, http://technet.microsoft.com/en-us/sysinternals/bb896646.aspx.

[3] Samsung Elec. *2Gx8 Bit NAND Flash Memory (K9GAG08U1A)*, 2006.

[4] Samsung Elec. *2Gx8 Bit NAND Flash Memory (K9GAG08U0M-P)*, 2006.

[5] L.-P. Chang and T.-W. Kuo. Efficient management for large-scale flash-memory storage systems with resource conservation. *Trans. Storage*, 1(4):381–418, 2005.

[6] M.-L. Chiang, P. C. H. Lee, and R.-C. Chang. Using data clustering to improve cleaning performance for flash memory. *Softw. Pract. Exper.*, 29(3):267–290, 1999.

[7] J.-W. Hsieh, L.-P. Chang, and T.-W. Kuo. Efficient on-line identification of hot data for flash-memory management. In *Proceedings of the 2005 ACM symposium on Applied computing (SAC '05)*. ACM, 2005.

[8] D. Kang, D. Jung, J.-U. Kang, and J.-S. Kim. μ-tree: an ordered index structure for nand flash memory. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software (EMSOFT '07)*. ACM, 2007.

[9] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee. A superblock-based flash translation layer for nand flash memory. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software (EMSOFT '06)*. ACM, 2006.

[10] J. Kim, J. M. Kim, S. Noh, S. L. Min, and Y. Cho. A space-efficient flash translation layer for compactflash systems. *Consumer Electronics, IEEE Transactions on*, 48(2):366–375, May 2002.

[11] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song. A log buffer-based flash translation layer using fully-associative sector translation. *Trans. on Embedded Computing Sys.*, 6(3):18, 2007.

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