A Time-Predictable System Initialization Design for Huge-Capacity Flash-Memory Storage Systems

Chin-Hsien Wu*
Department of Electronic Engineering
National Taiwan University of Science and Technology, Taipei, Taiwan
chwu@mail.ntust.edu.tw

ABSTRACT

The capacity of flash-memory storage systems grows at a speed similar to many other storage systems. In order to properly manage the product cost, vendors face serious challenges in system designs. How to provide an expected system initialization time for huge-capacity flash-memory storage systems has become an important research topic. In this paper, a time-predictable system initialization design is proposed for huge-capacity flash-memory storage systems. The objective of the design is to provide an expected system initialization time based on a coarse-grained flash translation layer. The time-predictable analysis of the design is provided to discuss the relation between the size of main memory and the system initialization time. The system initialization time can be also estimated and predicted by the time-predictable analysis.

Categories and Subject Descriptors

C.3 [**Special-Purpose And Application-Based Systems**]: Real-time and embedded systems; D.4.2 [**Operating Systems**]: Storage Management: Secondary storage; B.3.2 [**Memory Structures**]: Mass storage

General Terms

Design, Management, Performance

Keywords

Flash Memory, Flash Translation Layer, Storage Systems, Embedded Systems

1. INTRODUCTION

Flash-memory has become a popular alternative in storage systems, due to its characteristics in non-volatility, shock-resistance, and low power consumption. Because of recent technology breakthroughs, flash-memory storage systems are much more affordable than ever. Similar to other storage media, the capacity of flash-memory chips is doubled every two years. Because of the cost management issues, vendors face tremendous challenges in the design and implementation of block-device emulation software in flash management. In particular, the system initialization time must be under control so that related products remain competitive in the markets.

In particular, the NAND Flash Translation Layer (NFTL) is proposed and popularly adopted for the huge-capacity flash-memory management [2, 3]. It provides a coarse-grained address translation to map each given logical block address (LBA) of an access to a physical address on flash-memory, where the unit in reads and writes is a page, and the mapping is done at the block level (to be explained in a late section). Although NFTL-like coarse-grained mapping designs are fine with current products, they soon become improper in the near future. It is because the rapid growing of the capacity results in the significant increasing of the system initialization time in flash-memory management. Such an observation motivates this research.

In this paper, we propose a time-predictable system initialization design for huge-capacity flash-memory storage systems. A coarse-grained flash translation layer is designed for the time-predictable requirements. The coarse-grained flash translation layer contains a coarse-grained translation table that consists of coarse-grained slots. Because a coarse-grained slot is to provide a large-chunk address mapping, linear searching in the chunk could be time expensive such that a page map table is proposed to provide fast address mapping. The page map table can be stored in flash-memory so that they can be retrieved quickly. Because the operation of the coarse-grained flash translation layer depends on the coarse-grained translation table and the page map table, the system initialization is to construct the translation table. After the construction of the translation table, the page map table can be read from flash-memory by the coarse-grained slots of the translation table. As a result, we propose an efficient method to construct the coarse-grained slots of the translation table, and especially a binary search method is proposed. A time-predictable analysis is also provided to discuss the relation between the size of main memory and the system initialization time. According to the time-predictable analysis, the system initialization time is $O(S * log_2 N)$, where S is the number of chunks in flash-memory and N is the number of pages in a chunk.

The rest of this paper is organized as follows: Section 2

*Supported in part by a research grant from the National Science Council under Grant NSC 96-2218-E-011-011

provides an overview of flash-memory characteristics. Section 3 is the related work. Section 4 is the motivation. Section 5 presents the to-be-proposed time-predictable system initialization design. Section 6 provides the system initialization time estimation and measurement. Section 7 is the conclusion.

2. FLASH-MEMORY CHARACTERISTICS

A NAND flash-memory chip consists of blocks, and each block is of a fixed number of pages. A block is the smallest unit for erase operations, while reads and writes are done in pages. A page contains a user area and a spare area, where the user area is for the storage of a logical block, and the spare area stores ECC and other house-keeping information (i.e., LBA). The typical sizes of the user area and spare area of a page are 512B and 16B, respectively. The typical block size of a NAND flash-memory chip is 16KB [4, 5]. Because flash-memory is write-once, we do not overwrite data on each update. Instead, data are written to free space, and the old versions of data are invalidated and considered as dead. Such an update strategy is called “out-place” or “write-once” update. In other words, any existing data on flash-memory could not be over-written (updated) unless its residing block is erased. Pages that store live data and dead data are called “valid pages” and “invalid pages”, respectively. After a number of page writes, free space on flash-memory would become low. Activities that consist of a series of reads, writes, and erases with the intention to reclaim free space will start. The activities are called “garbage collection” and considered as overheads in flash-memory management. The objective of garbage collection is to recycle invalid pages scattered over blocks such that they could become free pages after erasings.

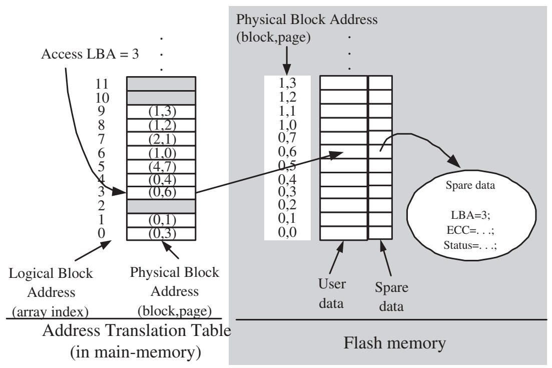

Figure 1: A RAM-resident Translation Table

3. RELATED WORK

The block-device emulation approach encourages a quick and popular deployment of flash-memory technology. Many well-known and popular (disk) file systems can be used with flash-memory block-emulated devices. Example of the block-device emulation are FTL [16, 17, 18] and NFTL [2, 3]. FTL adopts a page-level address translation mechanism (i.e., a fine-grained address translation). The translation table in Figure 1 is one kind of fine-grained address translation. The LBA ”3” is mapped to the physical block address (PBA) ”(0,6)” by the translation table. Note that LBA’s are logical addresses of pages mentioned by the operating system, and each PBA has two parts, i.e., the residing block number and the page number in the block! On the contrary, NFTL adopts a block-level address mechanism (for a coarse-grained address translation).

The main problem of FTL is on large memory space requirements for storing the address translation information because of its fine-grained address translation design. As a result, NFTL [2, 3] is proposed and popularly adopted for the huge-capacity flash-memory management. NFTL represents a typical coarse-grained design of a flash translation layer. NFTL is to map each given logical block address (LBA) of an access to a physical address on flash-memory, where the unit in reads and writes is a page, and the mapping is done at the block level. An LBA under NFTL is divided into a virtual block address and a block offset, where the virtual block address (VBA) is the quotient of the division of the LBA by the number of pages per block, and the block offset is the remainder of the division. Each VBA is associated with a data block and a log block by indexing a data block translation table and a log block translation table. When a write request is issued, the content of the write request is written to the page with the corresponding page offset in the data block. Because flash-memory is write-once, any subsequent writes of the same LBA are written to the first free page in the corresponding log block. When all of the pages of the log block are consumed, garbage collection starts by copying the valid pages of the corresponding data block and the log block into a new data block and then erasing the two blocks for recycling. Figure 2 shows the resulted data block and its log block when write requests to LBA’s $A = 100$, $B = 101$, and $F = 105$ are done for 2 times, 8 times, and 1 time. Any further writes to A, B or F would result in garbage collection, as shown in Figure 2. Some NFTL-like flash translation layers [8, 9, 10, 11] are also proposed such as BAST [9], FAST [9], N+K mapping scheme [10], and AFTL [11]. They all adopt the block-level address translation so that they have the same system initialization time with NFTL.

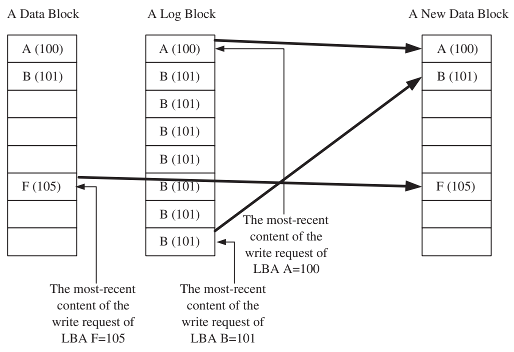
A Data Block
A Log Block
A New Data Block
A (100)
B (101)
F (105)
A (100)
B (101)
B (101)
B (101)
B (101)
B (101)
B (101)
B (101)
A (100)
B (101)
F (105)
The most-recent content of the write request of LBA A=100
The most-recent content of the write request of LBA F=105
The most-recent content of the write request of LBA B=101

Figure 2: NFTL

4. MOTIVATION

Each spare area of the first page of each block can contain enough house-keeping information for the construction of the translation table (i.e., the data block translation table and the log block translation table). As a result, when a flash-memory storage system is mounted, all (of the spare areas) of the first pages of all blocks on the flash-memory would be scanned. Consider a 32GB NAND flash-memory storage system[1]: Let the page size be 1024 bytes, and each block consists of 32 pages. For constructing the table of the block-level address translation will scan 1,048,576 (32*1024*1024/32) spare areas of the first pages of all blocks. Assume that the access time of each spare area is about $30us$ [5]. It will take about 30 secs to construct the translation table. Although some research [6, 7] provides fast system initialization methods but they can't guarantee a time-predictable system initialization so that a unpredictable system initialization time could occur. Such an observation motivates this research. We want to design a time-predictable system initialization design and the design can also provide shorter system initialization time for various large-chunk address translation layers such as NFTL, BAST, FAST, N+K mapping scheme, and AFTL.

5. A TIME-PREDICTABLE SYSTEM INITIALIZATION DESIGN

5.1 Overview

In this section, a time-predictable system initialization design is proposed for a coarse-grained flash translation layer. The coarse-grained flash translation layer must maintain a coarse-grained translation table (referred to CGTT) for the address mapping of the logical block addresses (i.e., LBA's) to the physical addresses. The coarse-grained translation table consists of coarse-grained slots. In section 5.2, two data structures: a coarse-grained slot and a page map table are defined for the efficient address translation of the logical addresses to the physical addresses. The coarse-grained slot can provide an address mapping for large chunks so that fast system initialization time can be achieved by scanning large chunks. Furthermore, because a coarse-grained slot is to provide a large-chunk address mapping, linear searching in the chunk could be time expensive so that the page map table is proposed to provide fast address mapping. The page map table can be stored in flash-memory so that they can be retrieved quickly.

Because the operation of the coarse-grained flash translation layer depends on the coarse-grained translation table and the page map table, the system initialization is to construct CGTT. After the construction of CGTT, the page map table can be obtained by the coarse-grained slots of CGTT. In Section 5.3, we propose a method to construct the coarse-grained slots of CGTT, and especially a binary search method is proposed so that the system initialization time can be $O(S * log_2 N)$, where S is the number of chunks in flash-memory and N is the number of pages in a chunk. In Section 5.4, a time-predictable analysis for the system initialization time is provided and the relation between the size of main memory and the system initialization time is also discussed in detail.

5.2 Data Structure

5.2.1 A Coarse-Grained Slot

The coarse-grained translation table consists of coarse-grained slots, where each slot can provide an address mapping for large chunks, called a segment (e.g., 1MB or above). Each segment can be defined as a series of contiguous blocks so that we can vary the size of a segment by changing the number of blocks in a segment. Each coarse-grained slot can be defined as a tuple (*cg_lba*, *cg_pba*, cg_free1, cg_free2, *cg_seq*), *cg_lba* denotes the corresponding logical address of the first page in the segment SEG whose physical address is *cg_pba*. cg_free1 denotes an address of the first free page in the segment SEG. cg_free2 denotes an address of the last free page in the segment SEG. Note that cg_free2 will be used and described in Section 5.2.2. When *cg_seq* is true, it denotes that all corresponding logical addresses of all consecutive written pages in the segment SEG are in sequential order. The purpose of *cg_seq* is to provide fast address mapping and will be described in Section 5.2.2. When a write request (LBA=α) is issued, the corresponding coarse-grained slot $slot_{cg}$ can be retrieved by quickly index-searching the address translation table.

According to the coarse-grained slots, an address mapping mechanism can be provided and faster system initialization time can be achieved because of large-chunk address mapping. Furthermore, because the garbage collection is to recycle the invalid data, these coarse-grained slots can be used to help the garbage collection distinguish valid data and invalid data.

5.2.2 Page Map Table

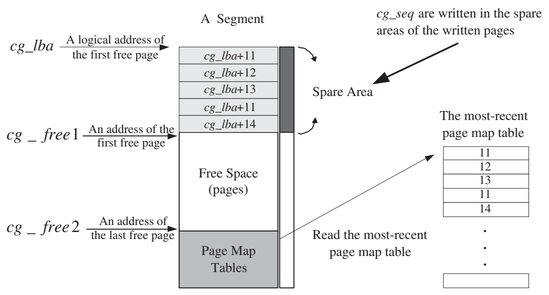
A Segment
cg_seq are written in the spare areas of the written pages
cg_lba
A logical address of the first free page
cg_lba+11
cg_lba+12
cg_lba+13
cg_lba+11
cg_lba+14
Spare Area
The most-recent page map table
cg _ free1
An address of the first free page
Free Space (pages)
11
12
13
11
14
cg _ free2
An address of the last free page
Page Map Tables
Read the most-recent page map table

Figure 3: A Page Map Table

A page map table is to help the fast address mapping from the logical block addresses to the physical addresses because a segment could be a large chunk and linear searching is too time expensive. Each segment should have a corresponding page map table that could be stored in the segment. The page map table consist of entries and each entry contains an offset from the starting logical address of the segment (i.e., *cg_lba*). For example, as shown in Figure 3, a segment has been written 5 pages and its most-recent page map table can be read from the physical address ((cg_free2)-1), where cg_free2 denotes an address of the last free page in the segment. Assume that the logical addresses of the 5 written pages are *cg_lba*+11, *cg_lba*+12, *cg_lba*+13, *cg_lba*+11, and *cg_lba*+14. The first written page (i.e., *cg_lba*+11) becomes an invalid page after the fourth page (i.e., *cg_lba*+11) is written. When a read request (LBA=*cg_lba*+11) is issued, we can locate the most-recent page data (i.e., the fourth written page) by the coarse-grained slots and the page map table. Note that if *cg_seq* is true, it means that all corresponding logical addresses of all consecutive written pages in the segment are in sequential order so that the page map table will not be needed for address mapping. As a result, for most read-only data (e.g., multimedia data or system data),

[1]A example in the market is the availability of 32Gbit flash-memory chips (SAMSUNG K9NBG08U5A).

the space for the page map tables can be saved due to the sequential logical addresses of all consecutive written pages.

The page map table should be maintained and updated efficiently in the main memory and could be written back to the flash-memory. The page map table is written in an appending fashion to the corresponding segment upwardly from *cg_*$free2$ and is read from flash-memory if needed. As a result, to maintain the page map table is an overhead that could occupy free pages. For avoiding the page map tables stored in flash-memory grow unexpectedly, the space for the page map tables in each segment should be fixed and the most-recent page map table is always stored at the end of the pre-allocated space (where a cyclic buffer scheme is adopted in the management of pre-allocated space).

5.3 System Initialization and Crash Recovery

Algorithm 1 Binary_Search(int low, int high, SEG)

```
1: if (high < low) then
2:    return no free pages
3: end if
4: int middle ← (low+high)/2;
5: if $SEG$[middle] is a free page and $SEG$[middle-1] is also
   a free page then
6:    call Binary_Search(low, middle-1, $SEG$)
7: else if $SEG$[middle] is not a free page then
8:    call Binary_Search(mid+1, high, $SEG$)
9: else if $SEG$[middle] is a free page and $SEG$[middle-1]
   is not a free page then
10:   $SEG$[middle] is the first free page
11:   the corresponding cg_$free1$ can be set as the physical
      address of $SEG$[middle]
12: end if
```

The operation of the coarse-grained flash translation layer depends on the coarse-grained translation table (i.e., CGTT) and the page map table. During the system initialization, CGTT must be reconstructed. After the construction of CGTT, the page map table can be obtained by the coarse-grained slots (i.e., *cg_*$free2$). CGTT consists of coarse-grained slots, and each coarse-grained slot is a tuple (*cg_lba*, *cg_pba*, *cg_*$free1$, *cg_*$free2$, *cg_seq*). *cg_lba* can be obtained by parsing the spare area of the first page of each segment. *cg_pba* is the physical address of the first page of each segment. *cg_seq* can be obtained by parsing the spare area of the last written page (i.e., before the first free page) after *cg_*$free1$ is obtained. It is because that when every page is written, *cg_seq* is also written to the spare area of the page and there are still undefined region in the spare area for such purpose. *cg_*$free1$ and *cg_*$free2$ can be obtained by locating the first free page and the last free page of each segment. We propose a binary search to search the first free page of each segment (i.e., *cg_*$free1$). The pseudo code of the binary search is shown in Algorithm 1. Assume that a segment SEG is searched for the first free page and SEG[i] denotes the i-th page. Note that *cg_*$free2$ can be obtained in the same way. If the space for the page map tables is 2 blocks (i.e., 64 pages), the time complexity for obtaining *cg_*$free2$ is ($\log_2 64$=6) page reads.

A crash recovery mechanism is proposed to avoid improper/failed system initialization. Crashes must be detected if any of them occurs during system initialization. Some page map tables could be lost and they could not

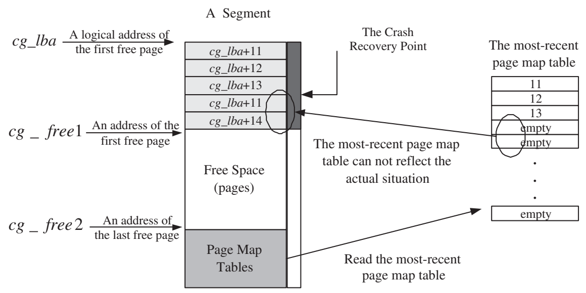
A Segment
cg_lba
A logical address of the first free page
cg_lba+11
cg_lba+12
cg_lba+13
cg_lba+11
cg_lba+14
The Crash Recovery Point
The most-recent page map table
11
12
13
empty
empty
empty
cg _ free1
An address of the first free page
The most-recent page map table can not reflect the actual situation
Free Space (pages)
cg _ free2
An address of the last free page
Page Map Tables
Read the most-recent page map table

Figure 4: Crash Recovery

be written to their corresponding segments when a crash occurs. As shown in Figure 4, the most-recent page map table that is read from flash-memory only records three written pages (i.e., *cg_lba*+11, *cg_lba*+12 and *cg_lba*+13). However, the most-recent page map table can not reflect the actual situation, where the fourth and the fifth written pages (i.e., *cg_lba*+11 and *cg_lba*+14) have been written to flash-memory. Due to a crash occurs, the most-recent page map table becomes out-of-date. As a result, when the most-recent page map table is read from flash-memory, the offset of the first empty entry in the page map table can be used to check if the corresponding page in the segment is free. If the corresponding page is not free, it means that the page map table is not the latest and a crash occurs. As a result, the crash recovery can construct fast and consistent flash-memory storage systems by skipping the scanning of the pages that have been recorded in the page table.

5.4 Time-Predictable Analysis

During the system initialization, CGTT must be reconstructed and CGTT consists of coarse-grained slots. The time-predictable analysis for the construction of CGTT would be discussed in the section.

Each coarse-grained slot (*cg_lba*, *cg_pba*, *cg_*$free1$, *cg_*$free2$, *cg_seq*) provides an address mapping for each segment so that each segment in flash-memory must be parsed to construct the corresponding coarse-grained slot. *cg_lba* can be obtained by parsing the spare area of the first page of each segment so that the time complexity is one page read. *cg_pba* is the physical address of the first page of each segment so that it can be obtained immediately. *cg_*$free1$ and *cg_*$free2$ can be obtained by the binary search so that the time complexity is $O(log_2 N)$ page reads, where N denotes the number of pages in a segment. After *cg_*$free1$ is obtained, *cg_seq* can be obtained by parsing the spare area of the last written page (i.e., before the first free page) and the time complexity is one page read. Assume that there are S segments in flash-memory, the total time complexity for the construction of CGTT is as follows:

$$O(S*(1+1+log_2 N)) = O(S*log_2 N) \qquad (1)$$

Let S be F/α, where F denotes the size of flash-memory and α denotes the size of each segment. Let N be α/p, where p is the size of each page (e.g., 1KB). As a result, Equation 1 can be as follows:

$$O(S*log_2 N) = O(\frac{F}{\alpha}*log_2(\frac{\alpha}{p})) \qquad (2)$$

Let $S = F/\alpha$ be M/γ , where M denotes the size of main memory (i.e., DRAM) that can be used and γ denotes the size of each coarse-grained slot. That is, the number of segments is equal to the number of coarse-grained slots so that each coarse-grained slot can handle each segment. The number of coarse-grained slots is limited to the size of main memory and the size of each coarse-grained slot. As a result, the size of each segment can be defined as follows:

$$\alpha = (F * \gamma)/M \quad (3)$$

According to Equation 2 and Equation 3, we have the following observations about the size of main memory and the system initialization time for CGTT. Assume that F and p are constant parameters because the flash-memory chips are hardware devices so that we can't control them in software design. γ is also a constant parameter because the size of each coarse-grained slot is determined in advance.

- If M is decreased for CGTT, then α would be increased so that the size of each segment would be increased. That is, each coarse-grained slot should provide sophisticated address mapping strategy and garbage collection management to handle the larger segment, and especially system performance for the address mapping of the logical block addresses to the physical addresses could be deteriorated. However, when α is increased, the system initialization time for GCTT can be decreased quickly according to Equation 2.
- If M is increased for CGTT, then α would be decreased so that the size of each segment would be decreased. According to Equation 2, the system initialization time for CGTT would be increased. However, each coarse-grained slot can handle the smaller segment so that address mapping performance and garbage collection overhead can be improved.

As a result, according to the time-predictable analysis, we can design a reasonable M so that the system initialization time for CGTT can be also predictable and under control.

6. SYSTEM INITIALIZATION TIME ESTIMATION AND MEASUREMENT

The size of flash-memory (F)	20GB
The size of each page (p)	1KB
The size of each coarse-grained slot (γ)	12B
The size of each block (β)	32 pages

Table 1: Constant Parameters

In the experiments, the size of flash-memory (F) was 20GB, the size of each page (p) was 1KB, the size of each coarse-grained slot (γ) was 12 bytes, and the size of each block was 32 pages (i.e., 32KB), as shown in Table 1. According to the parameters, the system initialization time estimation (i.e., $O(S * log_2 N)$) , S, α and N can be obtained under different size of main memory (i.e., M), as shown in Table 2. Compared to NFTL, NFTL would scan 655,360 (20*1024*1024/32) spare areas during the system initialization. As shown in Table 2, when the size of main memory was decreased to 0.75MB, our proposed method can have faster system initialization time (i.e., 545,386 spare areas) than NFTL. We also measured the system initialization time under the real trace characteristics. The characteristics of the experimental trace over 20GB disk is summarized in Table 3. In the trace, there were 13,198,805 and 2,797,996 sectors that were written and read, respectively, where each sector was of 512B. We must point out that there were 1,669,228 different LBA's that were accessed. The trace shows that many written data had spatial locality, where each LBA was written for 7.9 times averagely. During the collection of the trace, real applications were executed to have realistic workloads in daily life.

M (MB)	S (segments)	α (MB)	N (pages)	$S * log_2 N$ (spare areas)
1.5	131,072	0.15625	160	959,700
0.75	65,536	0.3125	320	545,386
0.375	32,768	0.625	640	305,461
0.1875	16,384	1.25	1,280	169,115
0.09375	8,192	2.5	2,560	92,750
0.046875	4,096	5	5,120	50,471

Table 2: System Initialization Time Estimation under Different Size of Main Memory

CPU	Intel Celeron 750 MHz
RAM	320 MB
Operating System	Windows XP
File System	NTFS
Storage Capacity	20GB
Applications	Web Applications, E-mail Clients, MP3 Player, MSN Messenger, Media Player, Programming, and Virtual Memory Activities
Duration	1 week
Total Write/ Read Requests	13,198,805 / 2,797,996 sectors
Different LBA's	1,669,228

Table 3: Trace Characteristics

Assume that the access time of each spare area of each page was about $30us$ [5]. The system initialization time estimation and measurement of the proposed method were shown in Figure 5, where the system initialization time of NFTL was also measured. The system initialization time estimation of the proposed method was calculated according to Table 2. The system initialization time of the proposed method and NFTL were measured under the experimental trace in Table 3. The difference between the system initialization time estimation and the system initialization time measurement is that the measurement can skip any free segment that consists of all free pages. For example, when the size of main memory was large such as 1.5MB, the measurement could skip more free segments than the estimation so that the measurement could be faster than the estimation. As a result, we can observe that the system initialization time estimation was close to the system initialization time measurement when the size of main memory was decreased. It shows that the time-predictable analysis of the proposed method is corresponding to the real system initialization time. Furthermore, the system initialization time measurement was less than the system initialization time of NFTL in all cases so that the proposed method can provide a fast and predictable system initialization time.

As shown in Figure 5, when the size of main memory was decreased, the number of segments was decreased and the number of pages in a segment was increased. Because the system initialization time was bounded in $O(S * log_2 N)$, the number of segments was a dominant parameter. The system initialization time would be decreased when the size of main memory was decreased. Nevertheless, when the size of main memory was decreased from 0.375MB to 0.046875MB, the system initialization time was linearly reduced and no significant improvement was observed. As a result, the size of main memory should be set as a reasonable value so that system initialization time and management overhead of each segment can be under control.

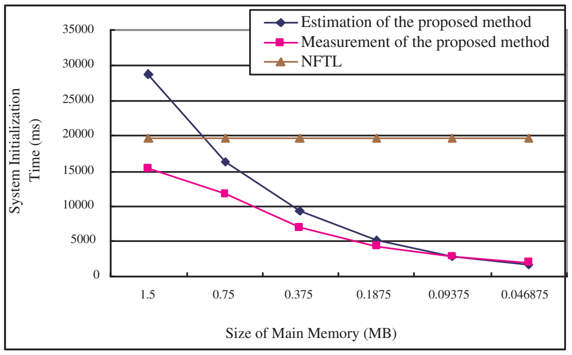

Figure 5: System Initialization Time Estimation and Measurement

7. CONCLUSION

A time-predictable system initialization design is proposed for huge-capacity flash-memory storage systems. The time-predictable system initialization design is based on a coarse-grained flash translation layer. The operation of the coarse-grained flash translation layer is based on a coarse-grained translation table and page map tables. The coarse-grained translation table consists of coarse-grained slots that can provide large-chunk address mapping so that fast system initialization time can be achieved by scanning large chunks. The page map table is to provide fast address mapping because linear searching in the large chunk could be time expensive. The system initialization is to construct the coarse-grained slots of the translation table and page map tables. The major contributions of this paper are summarized as follows:

- We propose an efficient method to construct the coarse-grained slots by binary searching. After the construction of the coarse-grained slots of the translation table, the page map table can be read from flash-memory by the coarse-grained slots.
- We propose a time-predictable analysis to predict the system initialization time and discuss the relation between the size of main memory and the system initialization time. According to the time-predictable analysis, the system initialization time is $O(S * log_2 N)$, where S is the number of chunks in the flash memory and N is the number of pages in a chunk.
- According to the system initialization time estimation, we can really predict the system initialization time under different size of main memory. As a result, the system initialization time of the design is predictable and under control.

For future research, we should further explore different application characteristics and the designs of super-huge-capacity flash-memory storage systems. Especially, how to efficiently manage a large segment (chunk), and at the same time, to provide a reliable and scalable flash translation layer with time-predictable system initialization will become an important research topic. Furthermore, for different embedded application systems and different requirements for embedded storages, a sophisticated customization of flash-memory storage systems and tool development become important issues.

8. REFERENCES

[1] http://www.samsung.com/Products/Semiconductor/NANDFlash/index.htm
[2] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho, "A Space-Efficient Flash Translation Layer for Compact-Flash Systems," IEEE Transactions on Consumer Electronics, Vol. 48, No. 2, MAY 2002.
[3] U.S. Pat. No. 5,937,425 "FLASH FILE SYSTEM OPTIMIZED FOR PAGE-MODE FLASH TECHNOLOGIES"
[4] R. Bez, E. Camerlenghi, A. Modelli, and A. Visconti, "Introduction to Flash Memory," Proceedings of The IEEE, Vol. 91, No. 4, April 2003.
[5] Samsung Electronics. NAND flash-memory datasheet and SmartMedia data book, 2006.
[6] C. H. Wu, T. W. Kuo, and L. P. Chang, "The Design of Efficient Initialization and Crash Recovery for Log-based File Systems over Flash Memory,'ąě accepted and to appear in ACM Transactions on Storage (ACM TOS).
[7] Keun Soo Yim, Jihong Kim, and Kern Koh, "A Fast Start-Up Technique for Flash Memory Based Computing Systems," To appear in Proceedings of the ACM Symposium on Applied Computing (SAC'05), Santa Fe, USA, March 2005.
[8] Soo-Young Kim and Sung-In Jung, "A Log-based Flash Translation Layer for Large NAND Flash Memory," ICACT, Feb 2006.
[9] Sang-Won Lee, Won-Kyoung Choi, and Dong-Joo Park, "FAST: An Efficient Flash Trnaslation Layer for Flash Memory," EUC Workshop 2006.
[10] Chanik Park, Wonmoon Cheon, Yangsup Lee, Myoung-Soo Jung, Wonhee Cho and Hanbin Yoon, "A Re-configurable FTL (Flash Translation Layer) Architecture for NAND Flash based Applications," IEEE International Workshop on Rapid System Prototyping, 2007.
[11] C. H. Wu and T. W. Kuo, "An Adaptive Two-Level Management for the Flash Translation Layer in Embedded Systems," IEEE/ACM 2006 International Conference on Computer-Aided Design (ICCAD), San Jose, USA, November 5-9, 2006.
[12] D. Woodhouse, Red Hat, Inc. "JFFS: The Journalling Flash File System".
[13] Intel Corporation, "LFS File Manager Software: LFM".
[14] Aleph One Company, "Yet Another Flash Filing System".
[15] M. Rosenblum, and J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System," ACM Transactions on Computer Systems 10(1) (1992) pp.26-52.
[16] Intel Corporation, "Understanding the Flash Translation Layer(FTL) Specification".
[17] Intel Corporation, "Software Concerns of Implementing a Resident Flash Disk".
[18] Intel Corporation, "FTL Logger Exchanging Data with FTL Systems".