The Fifth International Workshop on Software Support for Portable Storage

Scottsdale, Arizona, U.S.A.

October 28, 2010
The International Workshop on Software Support for Portable Storage (IWSSPS 2010) will focus on software issues related to portable storage. New emerging storage media such as Flash memory require extensive software support for higher performance and reliability, lower power consumption, and value-added functionalities. The goal of this workshop is to bring together people from industry and academia who are interested in all aspects of software support for portable storage. The workshop will provide a forum to present and discuss new ideas, new research directions and to review current trends in this area. The scope of the workshop includes, but is not limited to, the following topics:

- File systems for portable storage
- Interaction between file systems and portable storage
- Flash memory storage designs
- Solid state disks (SSDs)
- Power management for HDDs including microdrives
- DRM (Digital Right Management) for portable storage
- Security support for portable storage
- Distributed mobile storage
- Software support for new non-volatile memories (PRAM, FRAM, etc.)
- Software reliability for portable storage
- Software fault tolerance techniques for portable storage
- Novel applications of portable storage
The Fifth International Workshop on
Software Support for Portable Storage (IWSSPS 2010)
Scottsdale, Arizona, U.S.A.
October 28, 2010

Program Committee

Nitin Agrawal, NEC Laboratories, USA
Yuan-Hao Chang, National Taipei University of Technology, Taiwan
Jongmoo Choi, Dankook University, Korea
Jen-Wei Hsieh, National Taiwan University of Science and Technology, Taiwan
Insik Jin, Hitachi Global Storage Technologies, USA
Bjorn Por Jonsson, Reykjavik University, Iceland
Jaesoo Lee, Samsung Electronics, Korea
Chang-Gun Lee, Seoul National University, Korea
Sang-Won Lee, Sungkyunkwan University, Korea
Qiong Luo, Hong Kong University of Science and Technology, Hong Kong
Suman Nath, Microsoft Research, USA
Praveen Rao, University of Missouri-Kansas City, USA
Mustafa Uysal, VMware, USA
Stratis Viglas, University of Edinburgh, UK

Organizers

Jen-Wei Hsieh, National Taiwan University of Science and Technology, Taiwan, General Chair
Bongki Moon, University of Arizona, USA, Program Committee Chair
Table of Contents

Summary of Design Flows for Using NAND Flash Memory for Soft/Hard Real-Time Programs ....... 1
Kyoung-Soo We, Duhee Lee, Chang-Gun Lee

FTL Design for TRIM Command ................................................................. 7
Joohyun Kim, Haesung Kim, Seongjin Lee, Youjip Won

A Preemptive Log Buffer-Based Flash Translation Layer for Real-Time Applications ............ 13
Hyunjin Cho, Dongkun Shin, Sang-Won Lee, Young Ik Eom

Compression Support for Flash Translation Layer ...................................... 19
Youngjo Park, Jin-Soo Kim
ABSTRACT
NAND flash memory has been widely used as a nonvolatile storage for data. However, it does not support byte-level random access and hence directly executing program codes from NAND flash memory is not possible. Thus, we have to use RAM as working storage for preloading the program codes and executing them from RAM. This approach, however, requires a large amount of expensive RAM as the program size dramatically increasing.

With this regard, our research group has been developing a framework for executing real-time program codes on NAND flash memory with the minimal usage of RAM. This paper overviews the entire framework targeting not only for soft real-time programs but also for hard real-time programs. Specifically, we first explain our baseline design flow called RT-PLRU (Real-Time constrained Pinning and LRU combination) targeting a single tasking soft real-time application. Then, we generalize it to mRT-PLRU (Multi-tasking RT-PLRU) so that the design flow can be applied to multi-tasking soft real-time application. This mRT-PLRU design flow is further extended to HRT-PLRU (Hard RT-PLRU) targeting multi-tasking hard real-time applications.

Categories and Subject Descriptors
B.3.3 [Memory Structure]: Performance Analysis and Design Aids—Worst-case analysis

General Terms
Design, Experimentation, Performance

Keywords
Real-Time Systems, NAND Flash Memory, Pinning/LRU Combinations, Convex Optimization

1. INTRODUCTION
For embedded systems to provide more and more advanced features, their software program sizes are dramatically increasing. This trend is also true for embedded real-time systems such as portable media players, personal navigators, and automobiles systems. Expecting such dramatic increase of program codes, the current embedded architecture with small RAM and flash memory should be entirely revised by employing a cost-effective storage architecture for storing and executing large volume real-time program codes.

With this respect, a solid-state flash memory such as NOR flash and NAND flash is a good candidate for storing large volume program codes due to its high degree of integration, non-volatility, shock resistance, and low power consumption. Figure 1(a) shows the NOR+NAND storage architecture where program codes are stored and directly executed from NOR flash memory—called XIP (eXecution In Place), while data are stored and used from NAND flash memory via RAM. This storage architecture, however, is not the best in terms of the cost as the program size becomes huge, because it has to employ a huge NOR flash memory whose per-byte cost is much more expensive than NAND flash memory.

Thus, we propose to use All-NAND storage architecture as in Figure 1(b) where the program codes are also stored in NAND flash memory. However, NAND flash memory only supports 2KB page-based reads, not byte-level random reads, which prevents XIP from NAND flash memory. Thus, the necessary code pages must be loaded into RAM so that the codes are executed from RAM. This is the reason why the most commonly used approach in the industry is shadowing [15] that copies the entire program codes from NAND flash memory to RAM and executes them from RAM. However, this approach requires a large amount of RAM, which sacrifices the price benefit of NAND flash memory. In order to run programs in NAND flash memory with a smaller RAM, [12] proposes NAND-based demand-paging mechanisms that read pages into RAM when they are actually accessed while replacing pages as needed. However, due to unpredictable page-faults, this approach does not provide

---

1This work was supported in part by the IT R&D program of MKE [KEIT-10035243] and in part by the ITRC support program of MKE [NIPA-2010-C1090-1011-0008]. The ICT at Seoul National University provides research facilities for this study. The corresponding author is Chang-Gun Lee.

---

1For small-block NAND flash memory, one page size is 512 bytes. However, for explanation purposes, we assume large-block NAND flash memory with the 2KB page size.
any real-time guarantee of program executions. Addressing these limitations, our research group has been developing design flows [7, 9, 4] for using NAND flash memory for storing and executing real-time program codes with minimal usage of RAM.

The purpose of this paper is to summarize and overview the design flows [7, 9, 4] to form a generalized framework for using NAND flash memory for executing program codes of both soft and hard real-time applications. The first design flow called RT-PLRU [7] is explained targeting for the simplest setting of a single-tasking soft real-time application. Then, we explain its extended design flow called mRT-PLRU [9] targeting for general multi-tasking soft real-time applications. Finally, we explain how this design flow is further generalized to HRT-PLRU [4] targeting for general multi-tasking hard real-time applications.

The rest of this paper is organized as follows: The next section presents a brief background on NAND flash memory and motivations for our research. In Section 3, we summarize and overview design flows which our research group has been developing for real-time program executions. Section 4 presents the experimental results of the design flows. Section 5 surveys the related work. Finally, Section 6 concludes this paper.

2. BACKGROUND AND MOTIVATIONS
A NAND flash memory chip consists of a number of blocks. Typically, each block has 64 pages where each page consists of 2 KBytes main area and 64 Bytes spare area [14]. The main area is used to store regular information such as program codes/data and movie files while the spare area is used for storing auxiliary information such as error correction codes (ECC) and page-mapping information. Each page is read or written through an one-page size internal register. Related to this internal structure, NAND flash memory exhibits challenging characteristics: (1) 2KB page based read/write, i.e., no byte-level random access, (2) long delay write, and (3) erase before write with even a longer delay and a erase count limit.

These characteristics make it hard to execute a program directly from NAND flash memory. Therefore, we have to use RAM as a working storage for executing a program as shown in Figure 1(b). Noting that the per-byte price of RAM is about 5 times more expensive than that of NAND flash memory, we need to minimize the required RAM size to reduce the unit cost of production. If we pre-read and keep (i.e., “pin”) all the NAND pages into RAM, which is called the shadowing approach, the actual program execution does not incur any NAND read time, called time cost. However, it requires one RAM page for every NAND page, called space cost, resulting in a huge total space cost. As the other extreme, if we on-demand read (i.e., “demand-paging”) all the NAND pages into a single RAM page, it causes frequent page faults whenever the program tries to access codes and data in a different NAND page. Thus, the overall time cost is huge resulting in the deadline miss. So, our problem is how to optimally trade-off the time and space costs by combining pinning and demand-paging such that the space cost is minimized while keeping the time cost under the timing constraint.

3. DESIGN FLOWS FOR USING NAND FLASH MEMORY FOR REAL-TIME PROGRAMS
For now, we overview design flows to form a generalized framework for using NAND flash memory for executing program codes of both soft and hard real-time applications. For the simplicity, not data sections but only code sections of programs are considered in the design flows.

3.1 Design flow for a single-tasking soft real-time application
The first design flow called RT-PLRU [7] is explained targeting for the simplest setting of a single-tasking soft real-time application. RT-PLRU is applied to a media player program targeting a PMP (portable media player). Once started, the media player program executes periodically for playing a sequence of movie frames. Each execution for playing the i-th movie frame is called the i-th instance. Each instance has a deadline, and ideally, the instance needs to be completed within its deadline. But practically, if the percent of instances meeting the deadlines is greater than a threshold ProbA, user does not perceive any distortion of quality. Thus, we can regard the media player program as a soft real-time program, and our temporal constraint is that the deadline meet probability should be greater than ProbA.

We extract timing-related information and page reference traces by pre-running the given executable file of the media player to get optimal combination of pinning and demand-paging for the target program. For the demand-paging, of course, we assume the LRU (Least Recently Used) replacement policy. Our goal is to maintain the additional page-fault count below the maximum tolerable page-fault count with the minimum possible RAM. The maximum tolerable page-fault count can be decided when all the program code is pre-loaded into RAM, that is, shadowing. If we use only LRU, all the RAM pages are shared by NAND code pages referenced during the execution. Once the RAM size L and the page reference sequence are given, we can compute the number of page-faults. On the other hand, if we use only pinning, we can pin up to L − 1 pages reserving one page for non-pinned pages. In this case, non-pinned NAND code pages shares one RAM page. However, neither LRU nor pinning may give the optimal solution. Thus, we propose to optimally combine these two strategies.
Figure 2: Optimal combination of pinning and LRU

Figure 2 conceptually shows this—as the number of pinned pages increases (solid line), the number of required LRU pages decreases (dashed curve). The sum of the number of pinned pages and the number of required LRU pages is the total number of required RAM pages, which is represented by the dotted curve. The minimum point of the dotted curve is the optimal combination of pinning and LRU.

Exhaustive search can find the true optimal combination of pinning and LRU, but it has an exponential complexity. So an approximated algorithm that considers only one combination for a given “pinning size” based on “most commonly occurring first” heuristic is proposed in [7]. After getting “required RAM pages for LRU” of each instance for a given pinning size, the number of required RAM pages for LRU is represented as a random variable \( X \). Then each RAM pages for LRU can be considered as \( N \) sample data for the random variable \( X \). And we assume the probability distribution of \( X \) as a normal distribution for the simplicity. Finally we can compute required RAM pages for LRU for a given pinning size, \( X_{th} \), such that the cumulative probability of \( X \) being less than \( X_{th} \) is higher than the given threshold \( Prob_{th} \), i.e.,

\[
Prob(X \leq X_{th}) > Prob_{th},
\]

using this normal distribution.

3.2 Design flow for multi-tasking soft real-time applications

RT-PLRU proposed in [7] is limited to a single real-time task. But in general, several tasks are being executed concurrently in practical real-time systems. So we extend RT-PLRU to mRT-PLRU (Multi-tasking RT-PLRU) [9] targeting for general multi-tasking soft real-time applications. mRT-PLRU makes multiple real-time tasks optimally use RAM so that they can guarantee their deadlines with a probability higher than a threshold denoted by \( Prob_{th} \) of each task denoted by \( \tau_i \). The technique consists of two steps. In the first step, called a per-task analysis, we investigate each task’s NAND page access pattern to find the function of “RAM size vs. optimal execution time (and the corresponding optimal PLRU).” Using those functions found for all the tasks, in the second step, we perform a special convex optimization called a stochastic-analysis-in-loop optimization that iteratively increases RAM allocations to tasks in a way of minimizing the total RAM size until the stochastic-analysis says that all the tasks can satisfy the deadline meet probability requirements. At this point, to avoid inter-task conflicts in RAM, we partition RAM and allocate partitions to tasks for their dedicated use.

In per-task analysis step, we analyze each task one by one by using the technique proposed in [7]. After the first step, we can collect the following data:

- the optimal pinning/LRU combination of \( \tau_i \) for RAM partition size denoted by \( S_{i}^{\text{pinning}} \),
- its corresponding average execution time \( avgE_i(S_i) \) of number of jobs of \( \tau_i \) for \( S_i \), and
- its corresponding execution time distribution of number of jobs of \( \tau_i \) for \( S_i \).

The sum of RAM pages to be used for pinning denoted by \( S_{i}^{\text{pinning}} \) and RAM pages to be used for LRU denoted by \( S_{i}^{\text{LRU}} \) is \( S_i \).

In the second step, stochastic-analysis-in-loop optimization, the stochastic-analysis proposed in [5] takes the execution time distributions \( E_1, E_2, \ldots, E_n \) as inputs and computes the response time distributions \( R_1, R_2, \ldots, R_n \) as outputs. From the computed response time distribution \( R_i \) of each task \( \tau_i \), we can simply calculate the deadline meet probability of \( \tau_i \) by summing up the probabilities that the response time is smaller than or equal to the deadline \( D_i \). The research on the stochastic-analysis says that the average system utilization is the most important factor that determines the deadline meet probabilities. That is, there is a strong tendency that if the average system utilization is small, the deadline meet probabilities are high for all the tasks. Thus, our optimization tries to maximally reduce the average utilization for the same increase of RAM in order to meet all \( Prob_{th} \) requirements with the smallest possible increase of \( \sum S_i \).

For this, we first transform the \( avgE_i(S_i) \) function of each \( \tau_i \) to the average utilization function denoted by \( avgU_i(S_i) \) by dividing \( avgE_i(S_i) \) by the period \( p_i \), that is,

\[
avgU_i(S_i) = \frac{avgE_i(S_i)}{p_i}.
\]

From \( avgU_i(S_i) \), we construct a convex approximated function \( \overline{avgU}(S_i) \) consisting of only convex-hull frontiers as
shown in Figure 3. Such computed convex approximation function \( \overline{\text{avg}U_i(S_i)} \) is represented by a list in increasing \( S_i \)-order as follows:

\[
\overline{\text{avg}U_i(S_i)} = \left\{ \left( \frac{S_1}{\text{avg}U_1^i} \right), \left( \frac{S_2}{\text{avg}U_2^i} \right), \ldots, \left( \frac{S_n}{\text{avg}U_n^i} \right) \right\}.
\]

With this convex approximated function \( \overline{\text{avg}U_i(S_i)} \) for every task \( \tau_i \in \{ \tau_1, \tau_2, \ldots, \tau_n \} \) and stochastic-analysis, we allocate a RAM page to a specific task in a way of maximizing the ratio of the average utilization reduction until response time of all tasks meet their deadlines with a probability higher than \( \text{Prob}_{1\text{h}} \). For example, if we are given \( \overline{\text{avg}U_1(S_1)} \) and \( \overline{\text{avg}U_2(S_2)} \) as shown in Figure 4 for two tasks \( \tau_1 \) and \( \tau_2 \), we pick \( \tau_1 \) to increase \( \text{cur}_1 \) and hence \( S_1 \) since its slope to the next convex hull frontier is sharper than that of \( \tau_2 \).

In this way, stochastic-analysis-in-loop optimization can make the system meet \( \text{Prob}_{k\text{h}} \) for all the tasks with the smallest increase of total RAM size \( \sum_{i=1}^{n} S_i \).

3.3 Design flow for multi-tasking hard real-time applications

RT-PLRU [7] and mRT-PLRU [9] are only applied to soft real-time systems. So we propose HRT-PLRU [4] targeting for general multi-tasking hard real-time applications. HRT-PLRU is verified with the actual real-time programs for unmanned autonomous driving, which are six real-time programs with variable periods and code sizes. The technique for finding optimal pinning/LRU combination of RAM consists of two steps similar to mRT-PLRU. The first step is "per-task analysis", and the second step is "convex optimization".

In the first step, we analyze each task one by one to find the relation between "allocated RAM partition size vs. WCET". Unlike [9], to analyze each task, we consider control flow graphs of tasks as in Figure 5(a) where \( x_S \) and \( d_S \) denote the numbers of visits to the corresponding basic blocks and edges. And Figure 5(b) represents the NAND-page flow graph which is transformed from the Figure 5(a). Once this control flow graph is given, we can find the WCET of each task using an ILP-based WCET analysis [10]. We, of course, extend the ILP-based WCET analysis to consider the NAND page faults that happen when the RAM size is not large enough to hold all the program code pages in NAND flash memory. The extension is pretty similar to the cache hit/miss analysis in [10].

Using ILP-based WCET analysis [10], we fill up the table in Figure 6(a) as increasing \( S_i^{LRU} \) and \( S_i^{Pinning} \). For a fixed \( S_i^{LRU} \) value, when we increase \( S_i^{Pinning} \) from \( v \) to \( v + 1 \), we encounter an issue of selecting a page to be pinned. Fortunately, when we find \( WCET_i(S_i^{LRU}, v) \), the ILP solver also gives the page miss counts for all the pages that make the worst case scenario. Thus, we choose the page that has the largest miss count. If such a page is pinned when we calculate \( WCET_i(S_i^{LRU}, v + 1) \), it is expected to help reduce the WCET as much as possible.

In the \( WCET_i(S_i^{LRU}, S_i^{Pinning}) \) table of Figure 6(a), WCET values in a dotted diagonal line correspond to the same size RAM, i.e., \( S_i = S_i^{LRU} + S_i^{Pinning} \) but different combinations of \( (S_i^{LRU}, S_i^{Pinning}) \). Those WCET values are grouped, sorted, and depicted as dots on the same \( S_i \) in Figure 6(b). By taking only the minimum points for varying \( S_i \), we can finally make the "RAM size \( S_i \) vs. WCET" relation represented by the solid line in the figure. The minimum point for a \( S_i \) value is the near optimal combination of pinning and LRU in terms of minimizing the WCET with the allocated RAM size of \( S_i \). Only for those minimum points (see the circled point as an example), we keep (1) \( S_i^{LRU} \),...
(2) \( S_{\text{pinning}}^i \), and (3) the set of \( S_{\text{pinning}}^i \) code pages to be pinned, which are carried to the next step, i.e., convex optimization step, for optimally allocating the RAM size \( S_i \) to each task \( \tau_i \).

The second step, convex optimization, is similar to the second step of mRT-PLRU [9]. But WCRT analysis is used for schedulability check instead of stochastic-analysis, since all tasks are hard real-time tasks in [4]. In this step, we can find optimal pinning/LRU combination with \( WCET(S_{\text{LRU}}^i, S_{\text{pinning}}^i) \) of each task \( \tau_i \).

4. EXPERIMENTS

This section justifies the proposed design flows. In [7], as a media player program, we use MPlayer V 1.0rc2, and we use EDB9315A equipped with ARM9 processor and SNU-Hydra NAND board as a target embedded board. As a typical multi-tasking soft real-time application, we consider a mobile video phone scenario where the video decoder and encoder tasks should run concurrently in [9]. As a prototype system, we use Samsung Q1UMPC equipped with Intel A110 800Mhz processor, 64Gbyte NAND flash memory and 1GByte RAM in [9]. In [4], we consider six programs for unmanned autonomous driving and apply Rate Monotonic scheduling policy for those tasks. As a target processor, we consider Intel Core2Duo E7500 operating at 2.93 GHz.

In order to see the effect of the probabilistic real-time requirement \( \text{Prob}_{th} \) in [7], Figure 7 compares the required RAM sizes as varying \( \text{Prob}_{th} \). From 70% to 90% while fixing page fault delay as the default value 400 us. When \( \text{Prob}_{th} = 70\% \), the saving by our RT-PLRU compared to the shadowing is most significant, since we can take more advantage of relaxed requirement. As \( \text{Prob}_{th} \) gets larger, the required RAM pages by RT-PLRU slightly increases to meet the deadlines for more execution instances. Figure 8 compares the required total RAM sizes by the shadowing and mRT-PLRU [9]. Page fault delay is 400 us, too. As the number of tasks increases, the required total RAM sizes also increase for both methods. However, the increasing ratio for mRT-PLRU is much smaller than the shadowing, which enlarges the gap between them. This is because mRT-PLRU adds only the necessary size RAM partition for the probabilistic deadline guarantee of a newly added task, while the shadowing adds a large amount of RAM to hold the entire program code of the newly added task. Figure 9 compares the three approaches with six hard real-time programs as increasing the task periods, that is, scaling the periods by factors of 1, 2, and 4. Page fault delay is 100 us. \( \text{NOR XIP} \) is using NOR flash memory to store and execute program codes, and \( \text{All-NAND Shadowing} \) is using the All-NAND architecture but shadows the entire program codes into RAM for their execution, and \( \text{All-NAND Ours} \) is using All-NAND architecture optimized by our HRT-PLRU [4] for hard real-time systems. As we can easily expect, the NOR XIP approach requires NOR flash memory whose size is equal to the total size of all the program codes, independently to task periods. The All-NAND Shadowing approach requires the NAND flash memory with the same size of the entire program codes. It also requires the same size RAM for shadowing the entire program codes regardless of task periods. On the other hand, the All-NAND Ours approach requires the same size NAND flash memory but much smaller RAM thanks to our optimal partitioning and pinning/LRU combination. Such save of RAM becomes more significant as we increase the task periods. This is because the increased periods make a bigger room for allowing more page faults, which in turn allows more reduction of RAM. The gap between shadowing and HRT-PLRU is

Figure 7: Required RAM sizes as varying \( \text{Prob}_{th} \)

Figure 8: Required total RAM size for varying task sets

Figure 9: Total required NOR, NAND, RAM sizes as scaling up the periods
smaller than the results of RT-PLRU and mRT-PLRU, since hard real-time programs are target programs in [4] unlike [7, 9].

5. RELATED WORK

Recently, there has been much work on software technologies considering physical characteristics of NAND flash memory. For example, JFFS [17] and YAFFS [11] are file systems specially designed for NAND flash memory. [3, 6] are studies on how to use stripping and pipelining mechanisms in order to maximize the NAND throughput by hiding its physical characteristics. However, none of these studies address the issues of satisfying real-time requirements.

The problem of using RAM as working memory for executing codes in NAND flash memory is theoretically similar to the classical idea of using the cache for speeding-up main memory accesses. In order to minimize the impact of cache-miss penalties, there are thousands of studies using different techniques, e.g., prefetching [1] and branch prediction [13]. However, most of work focuses on maximizing the cache-hit ratio rather than guaranteeing real-time performance. In the real-time community, there have been great efforts to make the cache performance predictable, e.g., cache-locking [16], cache-partitioning [2], and cache-behavior analysis [8]. However, none of the above cache-related ideas can provide an integrated solution of pinning/LRU combinations and optimal RAM allocations for executing hard real-time program codes in NAND flash memory with the minimum RAM size.

6. CONCLUSIONS

This paper summarizes and overviews the design flows to form a generalized framework for using NAND flash memory for executing program codes of both soft and hard real-time applications. The optimal RAM configurations called RT-PLRU, mRT-PLRU, HRT-PLRU determine the optimal allocation of RAM partitions to multiple tasks and also the optimal pinning/LRU combinations for each RAM partition such that all the concurrent real-time tasks can meet their deadline with the minimum size of total RAM.

In our future work, we plan to extend the design flows to apply not only code sections but also data sections of the task, and reduce another amount of RAM by sharing RAM pages among tasks.

7. REFERENCES


FTL Design for TRIM Command

Joohyun Kim Haesung Kim Seongjin Lee Youjip Won
Department of Electronics and Computer Engineering
Hanyang University, Seoul, Korea
{createmain|matial|james|yjwon}@ece.hanyang.ac.kr

ABSTRACT
Introduction of SSD (Solid State Drive) to the market place has improved performance of PC dramatically because SSD has fast I/O processing capability, is shock resistant, and has no mechanical movements. TRIM command that notifies information of unused sector is suggested to ATA standard Technical Committee T13 to avoid overwrites and to handle invalid data effectively. We propose that in order to enhance the performance of the SSD up to enterprise standards, the Flash Translation Layer (FTL) has to support parallelism and TRIM command properly. We show that depending on the implementation of parallelism in FTL, write amplification factor can be an issue, especially on Hybrid Mapping FTL. In this paper, we introduce essential data structures to support TRIM command in FTL under multi-channel SSD architecture, and also address how and when the controller exploits this information. While exploiting the real-time evaluation and measurement system, Full System Emulator, we study the effect of the TRIM command implemented in two modified FTL schemes: Hybrid Mapping and Page Mapping FTL. We also provide a model to measure the effectiveness of the FTL. Experiment of installing OS shows that TRIM command generally increases the performance by 2% in Hybrid Mapping FTL, and 13% in Page Mapping FTL as compared to FTL without support for TRIM command. IOZone benchmark result shows that exploiting TRIM in Page Mapping FTL provides 10.69% increase in throughput for random write.

Categories and Subject Descriptors
D.4.2 [Operating Systems]: Storage Management; D.4.8 [Operating Systems]: Performance

General Terms
Design, Experimentation, Performance

Keywords
TRIM Interface, Performance Evaluation, SSD Emulator, FTL (Flash Translation Layer)

1. INTRODUCTION
NAND Flash Memory is a nonvolatile storage semiconductor that constitutes a SSD as a basic element. Its advantages include fast I/O throughput, low power consumption, and shock resistance. However, NAND Flash Memory has its own demerits. Operation in SSD has different units. For example, erase operation affects a block, which is a group of pages, and a read/write operation affects a page. Another fact that a file system does not take account of is that cells in a NAND Flash Memory have limited number of erase counts. These characteristics hinder optimal performance of conventional file system. Therefore, hiding the inherent characteristics of NAND Flash Memory and emulating HDD interfaces in SSD is essential to exploit existing file system. To provide the same interface as the HDD and hide all the complication from OS, SSD uses a layer called Flash Translation Layer (FTL). The purpose of FTL is to map a sector onto a Page of NAND Flash Memory. Even sophisticated FTLs cannot completely provide solutions to the overwrite issue, asymmetric I/O, and the restricted number of write counts.

In most cases, overwrite operation is a bottleneck to performance of SSD because it issues two time consuming operations repetitively, namely erase and write operations. Thus, many researches focus on minimizing or optimizing overwrites in SSD based on NAND Flash Memory. Agrawal et al. [1] suggests that minimizing effect of overwrite operation can significantly improve the performance of SSD. A simple approach in avoiding overwrite of records is to direct the corresponding sector to a free area and modify corresponding mapping information. Remapping of the data not only reduces number of overwrite and erase operations but also distributes used sectors in NAND Flash Memory and wear levels; however, it causes an increase in the number of invalid sectors in NAND Flash Memory.

In this work, we aim to devise a novel data structure and algorithm to effectively and efficiently handle trim command [14]. The algorithm and the data structure proposed in this work are to be embedded at the SSD controller. TRIM enabled file system and Operating System inform SSD controller about “useless” blocks. Controller can exploit this information in erase and merge operation and avoids unnecessary block and page copy operation of “useless” blocks. SSD controller is responsible for storing incoming trim information efficiently and for taking proper action at the right time exploiting the information provided by trim command. This work aims at addressing two technical issues in depth. The first issue is the design of data structure for trim com-
mand. File system can send trim information in bursty manner. SSD controller should be able to store and search block deallocation information instantly. The second issue is how and when the controller exploits this information.

1.1 Motivation
Modern state of art SSD controller has much to gain from exploiting block deallocation information issued by TRIM command. Block deallocation information can be exploited in many operations, e.g., file deletion, application installation, OS boot, etc., where file system collects information on unused blocks and to-be deleted blocks. When passed to SSD controllers, SSD controller can exploit this information in efficiently managing the NAND Flash pages and NAND Flash blocks. To fully take advantage of block deallocation information, two constraints need to be addressed. First, application file system or Operating System should be designed to exploit TRIM command. Secondly, underlying SSD needs to be designed to incorporate block deallocation information in its FTL algorithm. Unfortunately, however, few applications actually exploit TRIM command. EXT4 claims that it adopts TRIM command, but according to our experiment, we found that EXT4 does not generate TRIM command at all. Windows 7 is one of the few softwares which actually uses TRIM command and passes block deallocation information to SSD. However, we carefully believe that this feature is not mature enough either to fully exploit the nature of SSD or to properly incorporate the limitation of currently used SSD.

Let us provide an example. According to our experiment, Windows 7 Operating System uses TRIM command and sends unused sector list to SSD in boot phase. For fresh file system partition of tens of GByte capacity, the number of sectors, whether they are consecutive or not, can easily go over tens of millions. Therefore, to exploit full potential of block deallocation information, file system should have capability for search of used sectors and SSD should be capable of harbor block deallocation command in search-efficient and insert-efficient data structure.

Exploiting parallelism is important in SSD because it brings accreted effect by amalgamating the performance of low bandwidth Flash Memories. Although parallelism of SSD is not an issue directly related to TRIM command, we implement parallelism in this work because all modern SSD adopts multi-channel and multi-way architecture to increase read and write bandwidth. Parallelism can be realized in a FTL by designing striping, interleaving, and pipelining functions [6]. Striping divides a request into sub-requests and delegates each sub-request to separate channels; interleaving technique makes channel managers responsible for assigning requests to each channel to earn parallelism; pipelining technique provides parallelism by sharing a single channel by processing command-processing phase in second request right after when first request is through with data transfer phase. In order to maximize the parallelism of SSD, all aspects of the system has to be considered such as characteristics of plane, Page size in I/O, Block size in Erase operation, and configuration of Bus; maintaining the parallelism is a key to acquiring high performance SSD.

There are three major contributions in this paper.

1. Issues in implementing TRIM Interface Although TRIM command is introduced to the research community, there is little studies on the effect of TRIM to

<table>
<thead>
<tr>
<th>Features</th>
<th>Spec.</th>
<th>Features</th>
<th>Spec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capacity</td>
<td>16,384 Mbit</td>
<td>Read Page</td>
<td>60 µs</td>
</tr>
<tr>
<td>Program Page</td>
<td>800 µs</td>
<td>Erase Block</td>
<td>1.5 ms</td>
</tr>
<tr>
<td>Flash size</td>
<td>8192 blocks</td>
<td>Block Size</td>
<td>128 pages</td>
</tr>
<tr>
<td>Page Size</td>
<td>2048 Byte</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

storage system and to SSD. This paper presents insight in exploiting the interface. We integrate the interface with two of renowned FTL schemes, FAST [9] and Page Mapping FTL [2]. In the course of maximizing the performance of modified FTLs, we also propose data structures to effectively utilize the parallelism in the SSD architecture.

2. Effect Factor of SSD In this paper, we present a simple, yet intuitive measure to assess the performance of the FTL. It takes account of number of blocks used in programming requested data, used in programming requested sectors, and average erases that a program operation incurs.

3. Full System Emulator Unlike test driven experiments that does not take account of running system, our full system emulator can acquire real time information of Operating System, file system, and most importantly the Storage System. In this paper, we present the detailed view of the emulator and how two FTL architectures are integrated in the emulator.

2. SYNOPSIS

2.1 NAND Flash Memory

This subsection introduces NAND Flash Memory model used in the paper. There are two types of a NAND Flash Memory, Single-Level Cell (SLC) and Multi-Level Cell (MLC). SLC type NAND Flash Memory can store one bit in a cell, and it has better read and write speed and endurance level compare to MLC type NAND Flash Memory; however, SLC is expensive and has less storage capacity than that of MLC. Since MLC type NAND Flash Memory can store two bits in a cell, MLC can store more data in same dimension. Experiments in this paper use MLC type NAND Flash Memory. Features and performance of NAND Flash Memory is based on K9LAG08U0M Model and key specifications such as read, program, erase operations, and other features are shown in Table 1.

2.2 SSD Architecture

In this subsection, we discuss the SSD architecture implemented in full system emulator. SSD has three main parts, host-side processing module, controller module, and set of NAND Flash Memory. Host-side processing module communicates ATA commands with controller in the storage system. Controller module, known as FTL in SSD, consists of functions to process read, program, and other requests to a NAND Flash Memory. All data are stored in the third component of SSD, the NAND Flash Memory. Location of DRAM controller is dependent on the architecture of the storage system; some include it inside the controller. We
develop full system emulator where we can implement multi-way/multi-channel SSD with various FTL algorithms, write buffer management schemes, error correction module, etc.

SSD used in this paper is comprised of four K9LAV08U0M based Flash memory with capacity of 8 GB, and its parallelism is maximized by placing independent bus and NAND controllers to each NAND Flash Memory. Host-side module translates ATA standard command requested from file system into command that a FTL can process, such as Read, Program, and TRIM. Programming data in a NAND Flash Memory comprised of three steps, command processing phase, data transferring phase, and data programming phase [11]. In this study, we install commodity OS (Windows 7) on our simulator and examine the detailed behavior of SSD with and without TRIM command. At the time of this study, we were not aware of the other OS that uses TRIM command. In initial phases, OS determines whether the underlying storage is SSD or HDD. If it is determined to be SSD, then it checks whether the storage device is TRIM enabled SSD or not. File system used in Windows 7 exploits TRIM feature as soon as the OS finds out that attached storage system can process TRIM command. In our simulated SSD, each NAND Flash Memory in the SSD is connected to independent bus, and the SSD provides programming speed of 18.8 MB/s and read speed of 207.2 MB/s. NAND Flash Memory used in full system emulator has four planes. Size of DRAM is not limited in this work, and to be fair in measuring the effect of TRIM in FTL, additional features like caching is not implemented. Since TRIM command affects write and erase operations mostly, we do not consider its effect in read intensive workloads.

3. TRIM AND HYBRID MAPPING

In Hybrid Mapping scheme [7, 5, 9, 8], there are two types of blocks: Data and Log. For each type of blocks, that a Hybrid Mapping scheme uses two different Mapping schemes. In managing Data block, Hybrid Mapping scheme generally exploits Block Mapping, and Log Block is managed by Page Mapping scheme. This Hybrid scheme appreciates the small mapping table size compared to Page Mapping scheme and better I/O performance compared to Block Mapping scheme, but suffers from recurring Merge operations to move data on Log Blocks to Data Blocks. Copying contents of Log Blocks to Data Blocks are not always straightforward in SSD because it does not support in-place update; Merge operation checks both Data Block and Log Block, and copies only validate pages in each block to a clean Data Block, then erases old blocks.

Although many have proposed Hybrid-Mapping schemes, I/O propagation issue in Merge operation requires much attention from the research groups to minimize the effect. FAST scheme proposed by Lee et al [9]. FAST minimizes Merge operation frequent in Log Blocks of BAST [7] assigning exclusive properties to Log Blocks. One set of Log Block is dedicated to sequential requests and the other set is dedicated to random requests. By having dedicated sequential Log Blocks, the performance of sequential requests increases because cost of merge operation reduces. By having two different Log Blocks with dedicated purposes, implementing FAST becomes a better choice than BAST in terms of speed and endurance. However, the fact that FAST does not consider parallelism in processing write operation to Log Blocks it requires modifications to accelerate the I/O performance.

3.1 Hybrid Mapping and Multi-channel Architecture

FAST FTL in full system emulator could not handle the real time OS installation workload because it is not designed to operate on multi-channel/multi-way environment. FAST spends much of its time in merging and switching its log blocks, instead of processing the installation. One way to overcome the overhead of real time workload is to exploit the parallelism in the SSD architecture in FTL. How to write a data into a data block and log block has to be reconfigured to support striping, interleaving, and pipelining in FAST. Data is striped based on the plane and channels in NAND Flash Memory when a write request is issued. Since each NAND Flash Memory is connected to independent channels, we distribute consecutive write requests to each channel without explicitly exploiting the interleaving technique. There are two approaches in striping sectors. First approach is to strip it in units of page. The controller strips the request into consecutive sub-requests with length of sector adequate for single page, which is four since we use four channels in our implementation, and processing them upon receiving sub-requests in different NAND Flash Memory. The other approach can divide the same request into larger sub-requests, in units of block where each NAND Flash Memory receives 256 sectors. Programming in units of pages and blocks distributes the writing requests to many cells in the memory. The dispersion causes frequent occurrence of Merge operation, which calls for close attention.

There are two types of log block in FAST FTL: sequential log block and random log block. Adopting parallelism in random log block is not as difficult as sequential log block because random log block is only dependent on merge operation. On the other hand, sequential log block is dependent not only on merge operation but also on switch operation. A merge operation can cause increase in write amplification factor [4]. The write amplification factor depends on how parallelism in data block is implemented. Since a request is striped and distributed to separate sequential log blocks, it decreases the probability of a sequential log block being switched. We consider two data structures in implementing parallelism in the FTL, page-level and block-level parallelism.

In page-level parallelization shown in Fig. 1, controller partitions a write request into pages and evenly distributes them each channel. Using Page-level parallelism in a NAND Flash Memory greatly enhances speed of reading and programming because it divides a request and programs them into number of blocks in the Flash memory, and reads pages from number of blocks in the NAND Flash Memory all together. However, this method adversely induces associativity problem in performing Merge operation. For example, if a file system sends a write request of 1024 sectors, page-level parallelism in FTL distributes the request to all available random log blocks. When FTL decides to merge the 1024 sectors written in all random log blocks, FTL has to refer to corresponding random log blocks, and process the merge operation. As number of associated log blocks increase, I/O propagation also increases, which leads to longer processing time for Merge operation and at the same time degrading the performance.

In block level parallelization shown in Fig. 2, write request is partitioned into a unit of Flash Memory block and each
of the blocks is allocated to different channel. If the size of write request is smaller than a block, the data is written to single channel and thus write operation cannot exploit the parallelism. In this work, we use block-level parallelism in address interleaving. We found that page-level parallelism makes the associativity of log block prohibitively large, especially under multi-channel architecture and therefore that hybrid mapping cannot be used with page level parallelism. Block-level parallelism delivers reasonable performance with much less overhead in log block merge compared to the overhead of log-block merge operation in hybrid mapping with page-level parallelism.

3.2 Adopting TRIM: Hybrid Mapping
TRIM command carries set of sector numbers and respective lengths that notifies the number of requested sectors. Once file system sends TRIM command, it does not send further I/O requests until storage system returns response. Upon receiving TRIM command, FTL has two choices. First is to process them as soon as receiving the request and send response to the file system. Second is to send response to the file system, but processes the request at a convenient time. In some cases, processing the TRIM command in real-time can be ineffective. For example, according to our experiment, Windows 7 issues more than millions of unused sector lists to storage system via TRIM command. It is not practically possible to perform erase operation for millions of sectors in real-time. Erase operation can hinder performance because discrepancies in operation speed of erase and read/write holds the performance behind, and I/O requests are not further processed until all TRIM command is properly handled.

In this work, we propose to maintain block deallocation information passed by TRIM in memory and to postpone its processing until the actual merge operations occur. The prime objective of this approach is to minimize the latency and the interference with the ongoing operations. For efficient search, insert, and delete operation, data structure for maintaining block deallocation command plays a key role. We propose hash based linked list (HBL), Fig. 3. In HBL, all sectors are partitioned into N sets, each of which contains consecutive sectors. Block deallocation information takes the form of \([Start\ No, Length]\). Incoming block deallocation information is inserted into appropriate group. If incoming block deallocation information lies across the group boundary, we partition it so that the resulting information lies within a group. The sectors within a group are sorted with sector number and are organized using linked list. Our implementation of TRIM sector lists in Hybrid Mapping FTL allows to exploit the list in the case of Merge operation, and also be utilized in enhancing the performance of Wear-leveling, and Garbage Collection (GC).

4. TRIM AND PAGE MAPPING

Page Mapping FTL maps the entire sector into pages of any Blocks. Exploiting parallelism through Bus and Flash Memories in Page Mapping FTL is easier because Page Mapping FTL does not require frequent Merge operation or overwrite operations, etc.

When Page Mapping FTL receives data for programming, it partitions the request to smaller sub-requests with four sectors that can be issued to a page of a block in plane. Striped sectors are programmed exploiting pipelining. Note that an even numbered block comprises even plane, and odds comprises odd plane. This configuration can produce six times faster OS installation time than Hybrid Mapping Scheme. Invalidated pages caused by overwrites are continuously monitored in units of blocks, and when all pages in a block is invalidated, corresponding blocks are erased. Behavior of GC is similar in terms of moving the data from one block to another and erasing the blocks, difference of GC from merging is that it begins only when remaining storage area is insufficient.

4.1 Adopting TRIM: Page Mapping
Since Page Mapping FTL does not entail Merge operations instead activates GC to procure sufficient free space in the storage. Page Mapping FTL does not have to manage the TRIM sector List used in Hybrid Mapping FTL. When the FTL receives TRIM command, it simply erases sectors from the storage.

We designed two types of Page mapping table to manage the unnecessary sectors in Page Mapping FTL as shown in Fig. 4. Forward Mapping table allows to search a page using the location of a sector, conversely Backward Mapping table locates sectors in a Page. Upon receiving a data to be programmed, both of the Mapping table is modified to reflect the changes. Backward Mapping table provides information on remaining number of valid sectors in each block. When all sectors in a Block is invalidated, the corresponding block is erased from Backward Mapping table. Sectors requested by TRIM command are only erased in the Forward Mapping table, which causes discrepancies in contents of two Mapping table. The difference is reflected on Backward Mapping table regularly. Page Mapping FTL activates GC process when remaining free space in SSD under 30 percent; GC process chooses blocks with least number of valid sectors as a victim. As the remaining storage area decreases, the frequency of GC process increases.

5. EXPERIMENTS
This section provides experiments and evaluation of performance and effect of TRIM command. Note that we disabled buffer cache management in the SSD to visualize the effect of TRIM command. Our implementation of SSD architecture supports four parallel NAND flash channels.

5.1 Full System Emulator with QEMU
This subsection describes full system emulator, which measures the performance of the OS and Storage system. The emulator allows processing the I/O requests from OS and Application in real time. Unlike other trace driven evaluation approaches, it provides holistic performance measurements to changes in storage system. Our system emulator is based on QEMU/KVM [3] as shown in Fig. 5.

Full system emulator adopts two aforementioned FTL schemes. An independently operating real time monitoring tool verifies the I/O, Merge, GC, etc in SSD, which have significant benefits over trace driven evaluation. Full system emulator generates a log file after each iteration of experiment to make record of wear-level of each NAND Flash Memory, and mapping table.

Current implementation of Hybrid FTL uses single log block per chip for sequential requests and four log blocks per chip for random requests. There are four chips configured in the SSD architecture. Block Status Monitor module gathers status of each block. TRIM manager manages all TRIM commands, and exploits the information when FTL performs Merge operation. Every request that TRIM manager receives are reflected in mapping table right away. Block Manager manages the status of each Block.

5.2 Modeling FTL Effect Factor
This subsection describes the performance result of Hybrid Mapping FTL and Page Mapping FTL on two experiments. First is installing Windows7 that implements TRIM command in file system and second is IOZone benchmark, which is standard in measuring the performance of a storage [10]. Then, we describe a model for assessing the performance of SSD.

Eq. 1 measures the effect of TRIM command, where number of valid blocks, $N_{vb}$, denotes number of blocks with valid data after all program requests in the experiment is processed. Number of programmed blocks, $N_{pb}$, denotes number of blocks programmed while processing the program requests in a experiment. Number of erased blocks, $N_{eb}$, denotes average erase counts of a block in a experiment.

$$E_{fit} = \frac{N_{vb}}{N_{pb} \times N_{eb}}$$ (1)

5.3 OS Installation
Table 2 shows performance result of Hybrid Mapping FTL in installing Windows7, with and without triggering TRIM. The command did not cause difference in time spent in installing the OS; duration of the installation lasted 41 minutes for both. Examining the result triggering TRIM right after installing the OS, $E_{fit}$ measures 53.0 and 51.9 for TRIM and without TRIM, respectively, where enabling TRIM gives slightly better effectiveness. Wear leveling of each block in Hybrid Mapping FTL with TRIM improves about 2% compared to Hybrid Mapping FTL without the support for TRIM command. Wear leveling of Page Mapping FTL with TRIM enhances about 11.6% compared to Page Mapping FTL without support for TRIM command. Wear leveling measurement of Page Mapping FTL compared to Hybrid Mapping FTL produces about 60 times better performance. The result indicates that not only in I/O performance but also in wear leveling in Page Mapping is by far better than the hybrid method.

Table 3 compares performance of two FTL scheme in installing Windows7. Page Mapping FTL outperforms Hybrid Mapping FTL both in number of programmed blocks, $N_{pb}$, and time spent in installing the OS. Hybrid Mapping FTL uses 2.3 times more blocks and 5.7 times longer period in installation. There are two main reasons behind the outstanding performance of Page Mapping FTL with support for TRIM command. First, there is no merge operation, which is recurrent in Hybrid Mapping FTL. Second, there is no GC process, which is activated when the free space in the storage is less than 30 percent. The result suggests that enterprise level SSD exploiting Page Mapping scheme can produce incomparably better performance.

5.4 IOZone Benchmark
This subsection describes the IOZone Benchmark result on Table 2. The effect of TRIM measured by $E_{fit}$ of Hybrid Mapping FTL with IOZone benchmark reads 79.3 and 78.5 for TRIM and without TRIM, respectively. Result shows that there are more random writes and overwrites in IOZone than in installing the OS. The result also indicates that TRIM enhances the efficiency of FTL and wear leveling in IOZone benchmark.

Table 4 shows effect of TRIM command in Page Mapping FTL. Enabling TRIM enhances 12.4% of write speed compared to FTL without TRIM capability. It also shows higher throughput on other measures; it enhances 13.5% of re-write and 10.7% of random write test. TRIM command in Page Mapping FTL helps to enhance the performance in short
<table>
<thead>
<tr>
<th>Mapping</th>
<th>$N_{\text{ph}}$</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hybrid</td>
<td>53119</td>
<td>41 Min.</td>
</tr>
<tr>
<td>Page</td>
<td>23512</td>
<td>7 Min.</td>
</tr>
</tbody>
</table>

### Table 4: IOZone: Page Mapping FTL (MByte/Sec)

<table>
<thead>
<tr>
<th></th>
<th>w/ TRIM</th>
<th>w/o TRIM</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write</td>
<td>137.6</td>
<td>120.5</td>
<td>12.4%</td>
</tr>
<tr>
<td>Re-write</td>
<td>222.6</td>
<td>192.7</td>
<td>13.5%</td>
</tr>
<tr>
<td>Random Write</td>
<td>222.8</td>
<td>199.0</td>
<td>10.7%</td>
</tr>
</tbody>
</table>

term; however, as the storage suffers from deficiency of storage area, performance is severely affected by GC. The GC scheme runs when available storage area goes under certain threshold level. In our implementation of FTL, GC scheme is executed when available storage area is less than 30%. When GC is running under TRIM enabled Page Mapping FTL, the result of IOZone benchmark compared to TRIM disabled FTL gives 1.3% lower write throughput, 4.7% and 2.3% lower throughput for re-write and random write, respectively. It is because there were more GC operations in TRIM enabled FTL than TRIM disabled FTL. Exploiting TRIM command in FTL is essential in enjoying higher throughput; however, it calls for better design and implementation of GC scheme to be exploited along side of TRIM command.

#### 5.5 Analysis

The major findings in this paper are two-fold. First is that FTL can exploit deallocation information from file system, which gives advantage of reducing the number of merge and GC. Second observation, on the other hand, is not so favorable, indeed it needs much attention. From our experiments, we find that TRIM in file system causes ill behavior in SSD. The side-effect comes from file system trying to exploit TRIM, while not knowing that the FTL is actively endeavoring to reduce the number of erase behind-the-scenes. When Windows 7 file system finds that the underlying SSD is TRIM enabled, Windows 7 file system allocates new file system block to perform update instead of overwriting the new content to the same logical block. Similar behavior can be found in log-structured file system[12]. Since the file system always performs "write" to new block which is free block in SSD, this approach entails less merge operation for Page FTL. However, since this file system always writes to free block, it quickly uses up the free blocks in the SSD and garbage collection is triggered in relatively frequent manner. The overhead of garbage collection is orders of magnitude larger than that of merge operation. In our experiment, OS installation actually takes 5.7 times longer in Windows 7 when we use TRIM enabled SSD with hybrid FTL. This shows that although TRIM command brings good news to SSD, it still needs proper and elegant support from the file system to reduce the conflict and to achieve the better performance.

### 6. CONCLUSIONS

Recent introduction of TRIM command to T13 Technical Committee that allows a file system to inform locations of erased sectors to SSD has opened possibilities to reduce the cost of merge operation. However, FTL designers are responsible for implementing the proposed interface. In this paper, we propose data structure to manage the deallocation information and search the information. We also propose how and when to exploit this information in the FTL under multi-channel architecture. We measure effectiveness of TRIM enabled FTLs, Page Mapping FTL and Hybrid Mapping FTL, in real time system emulator with system-monitoring tool. Performance of TRIM enabled FTL compared to FTL without TRIM capability shows 2% and 13% increase in Hybrid and Page Mapping FTL, respectively. Although exploiting TRIM in Page Mapping FTL provides 10.7% better throughput in random write IOZone benchmark, Page Mapping FTL can suffer from GC as the storage utilization becomes high and available storage area becomes low, which requires careful design and implementation of GC algorithm.

### 7. REFERENCES


A Preemptive Log Buffer-Based Flash Translation Layer for Real-Time Applications

Hyunjin Cho, Dongkun Shin, Sang-Won Lee, Young Ik Eom
School of ICE
Sungkyunkwan University
Suwon, Korea
hjcho@ece.skku.ac.kr, dongkun@skku.edu, swlee@skku.edu, yieom@ece.skku.ac.kr

ABSTRACT

Flash memory is a good candidate for the storage device of real-time systems due to its non-fluctuating performance, low power consumption and high shock resistance. However, the garbage collection in flash memory can invoke a long blocking time. Moreover, an urgent write request should wait the completion of the garbage collection invoked by a normal write request. In this paper, we propose a preemptive flash translation layer (FTL), called pFAST, where garbage collection can be preempted by an urgent request. pFAST enables to estimate the worst-case waiting time of an urgent request which is determined by the size of the urgent request. Experiments using simulation showed that the response times of urgent requests under pFAST are significantly improved compared to the current FTL schemes.

1. INTRODUCTION

Flash memory has been widely used as a storage device for mobile embedded systems (such as MP3 players, PDAs and digital cameras) because of its low-power consumption, non-volatility, high random access performance and high mobility. Over the past several years, there has been a significant growth in the NAND flash market due to the increase of MP3 player and digital camera since these devices need a large amount of data storage. Recently, solid-state disks (SSDs), which are made of NAND flash memory chips, are substituted for hard disk drives in general purpose computing such as desktop PC [1, 2].

Since flash memory has a high shock resistance, it is useful for real-time systems such as industrial control systems, automobile and space shuttle which require a high reliability. In addition, the seek time of flash memory does not fluctuate unlike hard disk drive, thus it is favorable to providing a predictable I/O performance to real-time systems. Mobile system is also a good example requiring a real-time storage system. When a mobile system downloads data from Web, it stores data at a volatile buffer temporary and transfers them to the flash memory storage. Unless the storage provides a required write bandwidth, the data in the volatile buffer can be corrupted by the following downloaded data that should use the buffer space. Therefore, it is important to guarantee the maximum delay time for each I/O request.

However, flash memory has also a fluctuation on its bandwidth compared to DRAM due to its special features. This is a critical obstacle to using flash memory in real-time systems. The first is its “erase-before-write” architecture. To write a data in a block, the block should be erased before. The second feature is its asymmetric operations. While the flash memory is erased by the unit of block, the read and write operations are performed by the unit of page. A block is a bundle of several pages. For example, in Samsung’s large block NAND flash memory, one block is composed of 64 pages and the size of a page is 2 KB. In addition, the read, write and erase operations have different latencies, i.e., 25 µs, 200 µs and 2 ms, respectively. The third feature is that the page writing (programming) within a block should be done sequentially. That means that the page of lower address in a block should be programmed before the page of higher address in the block. Random page address programming is prohibited. The feature is called “sequentiality of programming” (SOP).

In order to handle these features, NAND flash memory file systems such as JFFS2 [3] and YAFFS [4] should be used. However, compatibility is a more important issue at most systems, and thus they prefer the block device file systems such as FAT, NTFS and EXT2/3 to the flash file systems. Since the block device file systems are designed for hard disk drives, a special software called flash translation layer (FTL) is required between the file system and the flash memory device. FTL provides a logical (virtual) address space to the file systems and maps a logical address to a physical address in flash memory devices. There are two kinds of mapping schemes depending on the mapping granularity, block-level mapping and page-level mapping. Block-level mapping permits only the block-level update. Therefore, even when only one page of a logical block is updated, all other pages of the block should be rewritten at the newly-allocated physical block requiring a high management cost.

For high performance, page-level mapping technique is suitable. For example, assume that an application sent a write request for the logical page address 100 and the page-level mapping wrote the data to the physical page address 200. The address mapping information was recorded at the page-level mapping table. If the application sends an update request on the data at the logical page address 100, FTL writes the new data at any other physical page, for example, the physical page address 201, because the flash memory page cannot be overwritten. The old data at the physical page address 200 are invalidated and the mapping information for the logical page address 100 is updated.

After a large number of data updates on flash memory, there will be many invalid pages. Then, a garbage collection
(GC) is required to reclaim the space occupied by invalid pages. The garbage collection should find the flash memory blocks which have many invalid pages and copy the valid pages of the blocks into other clean blocks in order to erase the old blocks. Therefore, intermittently, the response time of a flash I/O request might become quite long if it comes across the garbage collection operation. Chang et al. [5] proposed a real-time garbage collector for flash memory which uses the page-level mapping. However, page-level mapping requires too large mapping table to be used for a large-sized flash memory such as SSD.

To solve the problems of block-level mapping and page-level mapping, hybrid-level mapping techniques are proposed. In this scheme, several flash memory blocks are reserved for log buffer. The FTL using hybrid-level mapping scheme is called as a log buffer-based FTL. While the log blocks in the log buffer use the page-level mapping scheme, normal data blocks are handled by the block-level mapping. Therefore, the log buffer-based FTL requires a smaller-sized mapping table than the page-level mapping does. The maximum number of log blocks is given in order to limit the size of page-level mapping table. When a write request is sent to FTL, the data are first written into a log block and the corresponding old data in data block are invalidated. When all the log blocks are full and there is no empty space, one log block is selected as a victim and all valid pages in the log block are moved into data blocks to make a free space for on-going write requests. This step is called log block merge.

The log block merge operation of current log buffer-based FTL schemes could impose a long blocking time on real-time tasks depending on the log block association policy to be explained at Section 2. For a time-critical system, it is highly important to reduce the worst-case I/O latency so as to higher responsiveness. Although researchers have proposed various hybrid mapping FTLs focusing on minimizing the total I/O cost [6, 7, 8, 9, 10], there is little work done in providing a deterministic performance guarantee for flash memory storage systems.

This paper is motivated by the need of the FTL scheme providing a non-fluctuating performance to real-time systems. We propose a novel preemptive log buffer-based FTL, called pFAST, which is a modified version of the FAST scheme. To minimize the waiting time of an urgent write request, pFAST can preempt the garbage collection invoked by a normal non-urgent request.

The rest of the paper is organized as follows. In Section 2, the related works are introduced and are compared with the proposed scheme. Section 3 describes the details of pFAST scheme. Experimental results are presented in Section 4. Section 5 concludes with a summary.

2. RELATED WORKS

There have been many researches on log buffer-based FTLs, which can be divided into three kinds depending on the block association policy, i.e., block associative mapping (BAST) [6], fully associative mapping (FAST) [7] and set associative mapping (SAST) [9]. The block association policy means how many data blocks are associated with a log block. When a log block is allocated for only one data block in the BAST scheme, it can be used for multiple data blocks in the FAST scheme. In the SAST scheme, a set of log blocks can have the updated data of an associated set of data blocks. The distinction among these three schemes is similar to that of three cache architectures, i.e., the direct-mapped, fully-associative, and set-associative caches.

Figure 1 shows examples of hybrid mapping FTL schemes. There are six data blocks (B0∼B5) and four log blocks (L0∼L3). We assume that each block has four pages. When host sends an update request on data block, the log buffer-based FTL writes the new data into the log blocks invalidating the old pages in the data block. (The invalidated pages have a gray color.) Figure 1 shows the final status of flash memory for the write request sequence of “p0, p4, p8, p12, p16, p20, p1, p5”, every write request starting from p16 replaces one of four log blocks generating log block thrashing problem [7]. For the write request sequence of “p0, p4, p8, p12, p16, p20, p1, p5”, every write request starting from the write on p16 replaces one of four log blocks generating expensive block merge operation in the BAST scheme. Moreover, every victim log block in this example holds only one page when it is replaced; the other three pages remain empty. Therefore, the log blocks in BAST would show low

Figure 1: Log buffer-based FTL schemes.
space utilizations when they are replaced from the log buffer. To solve the problem of BAST scheme, the FAST scheme was proposed, where a log block can be used for several data blocks as shown in Figure 1(b), thus reducing the frequency of block merges. The data are written into the log blocks in the order of requests regardless of the corresponding data block number. Therefore, the utilization of log block is improved and the log block thrashing problem can be prevented. For the write request sequence of \(p0, p4, p8, p12, p16, p20, p1, p5^7\), there is no block merge in FAST while each write request incurs a log block merge in BAST. As a special case, FAST uses one sequential log block (SLB), which is associated to only one data block. Since all the pages in the SLB are written by the in-place scheme, SLB can be merged by either switch merge or partial merge.

However, the drawback of FAST scheme is that each log block has a high block associativity. For example, when the log block L0 in Figure 1(b) is merged with its associated data blocks, sixteen pages (16 pages = 4 blocks \(\times 4\) pages) should be copied since the log block L0 is associated with four data blocks, B0, B1, B2 and B3. This means that FAST incurs a large merge cost per one block merge operation. We denote the set of data blocks which are associated to the log block L as \(A(L)\). The number of elements in \(A(L)\) is called as the block associativity of L and is denoted by \(k(L)\). The maximum merge cost for a log block L is as follows [10]:

\[
N_{\text{block}} \cdot k(L) \cdot C_{\text{copy}} + (k(L) + 1) \cdot C_{\text{erase}}
\]  

(1)

where \(C_{\text{copy}}\) and \(C_{\text{erase}}\) are the costs of a page copy and a block erase, respectively. \(N_{\text{block}}\) is the number of pages in a block.

Since the maximum value of \(k(L)\) is the number of pages in a block, the worst case cost under the FAST scheme is \(N_{\text{block}}^2 \cdot C_{\text{copy}} + (N_{\text{block}} + 1) \cdot C_{\text{erase}}\). For the multi-level-cell (MLC) NAND flash memory, where a block consists of 128 pages, and the times for page read, page program and block erase are 50 \(\mu\)s, 650 \(\mu\)s and 2 ms, respectively, the worst-case time of a log block merge is about 6 seconds while the best-case time is 2 ms for a switch merge. Therefore, the worst-case cost is 3,000 times of the best-case cost. Figure 2 shows an example of log block merge cost variation under the FAST scheme. It takes about 1.4 sec at maximum for a log block merge operation. The fluctuation of block merge cost is significant.

Consequently, BAST shows a poor performance due to its log block thrashing problem and FAST shows a significant difference between the worst-case and best-case block merge times due to its variable block associativity. Therefore, neither of them is suitable to the real-time systems.

In the SAST scheme [9], a log block set with \(K\) number of log blocks can be used only for one data block set with \(N\) number of data blocks (\(N:K\) log block mapping). Therefore, it is guaranteed that the maximum block associativity of a log block is \(N\). Figure 1(c) shows an example of the SAST scheme using 3.2 log block mapping. Since the SAST scheme is a compromised version of BAST and FAST, it cannot solve both the log block thrashing problem and the high block associativity problem completely. For example, it invokes frequent log block merges when the corresponding data block set of write requests is frequently changed.

There are several real-time garbage collection mechanisms for flash memory storage such as RTGC [5] and ASR [11]. Instead of invoking a long latency garbage collection task which reclaims all invalid pages when there is no free space, they periodically launch a short latency garbage collection task to reclaim a partial blocks in order to reduce the block merging incurred from garbage collection. However, these techniques are devised for page-level mapping FTLs.

The opportunistic garbage collection (OGC) [12] handled the BAST hybrid mapping FTL. It permits to read flash pages when it arrives during a log block merge operation. However, a write request must wait until the log block merge operation is completed even when it is an urgent request. This is because the write request needs a free space that can be obtained by the log block merge operation. However, the proposed pFAST can preempt the log block merge operation to service a more urgent write request.

KAST [13] is a configurable log buffer-based FTL for real-time systems. User can configure the worst-case block merge time by limiting the maximum associativity of log block. However, it did not handle the waiting time of urgent request.

3. PREEMPTIVE GARBAGE COLLECTION

3.1 Block Merge Preemption

For a write request, \(r_i\), the response time, \(d(r_i)\), can be represented as follows:

\[
d(r_i) = w(r_i) + gc(r_i) + c(r_i)
\]  

(2)

where \(w(r_i)\), \(gc(r_i)\), and \(c(r_i)\) are the waiting time until the request can be serviced, the garbage collection time to make free space for the request, and the pure write cost, respectively. Generally, \(c(r_i)\) is determined by the data size of write request. However, \(w(r_i)\) and \(gc(r_i)\) are changed depending on the status of flash memory system. Therefore, we should be able to predict and minimize the delay times of \(w(r_i)\) and \(gc(r_i)\) in order to use the flash storage for real-time systems. In this paper, we focus on minimizing the waiting time \(w(r_i)\).

The long block merge operation of FAST may increase the waiting time of an urgent request. In order to minimize the waiting time, the proposed pFAST scheme provides a preemptable log block merge, where a log block merge operation is divided into several sub-merge operations as shown in Figure 3. For example, when a log block \(L_0\) is associated with three data blocks \(B_0, B_1\) and \(B_2\), the full merge operation for \(L_0\) can be divided into three sub-full merge (SFM) operations and one erase operation. Each sub-full merge operation copies the valid pages of one data block and erases the old data block. When the associativity of a log block is \(k\), the block merge operation for the log block can be divided into \(k\) number of sub-merge operations. If
the log block is a sequential log block, the partial merge operation can be divided into the page copy operation and the log block erase operation as shown in Figure 3(b). The preemptable log block merge operation checks whether there is a waiting urgent request at the check points. If an urgent request is waiting, the log block merge operation is preempted and the urgent request is serviced.

Figure 4 shows the log block merge preemption for an urgent request service. A normal request $T_L$ which writes the three pages $p_1$, $p_2$, and $p_3$ arrived at $t_0$. As shown in Figure 4(b), a full merge operation is invoked after one page is written at the log block $L_0$ with the physical block number (PBN) 523. If the log block $L_0$ is selected for a victim, the data blocks $D_1$, $D_3$, and $D_4$ should be merged with the log block $L_0$. The first sub-full merge operation copies the valid pages $p_4$-$p_7$ in $D_1$ and $L_0$ into the free block $F$ whose physical block number is 360. The block $D_1$ with the PBN 101 is erased and is changed to a free block. Since the urgent request $T_H$ arrived during the sub-full merge operation, pFAST suspends the merge operation and writes the page $p_{49}$ of the urgent request at the free block $F$ as shown in Figure 4(c).

After handling the urgent request, pFAST does not resume the suspended merge operation. Instead, pFAST cleans up the preempted log block merge operation since there is no empty free block to perform the remaining sub-merge operations. The remaining valid pages of $L_0$ are moved into the free block $F$. Then, $F$ with the PBN 101 is changed to a log block $L_0$ and the physical block with the PBN 400 is reserved for a free block after it is erased as shown in Figure 4(d). Therefore, one free log block is obtained after the clean-up operation. The normal request $T_L$ writes the remaining page $p_2$ at the log block $L_0$ and invokes a new full merge operation to make free space for the page $p_3$.

### 3.2 Preemptability Check

In pFAST, both the urgent data and the valid data in the suspended log block are written at the free log block. Therefore, pFAST should check the preemptability of the merge operation before the preemption. There should be a sufficient free space for both the urgent request and the suspended log block merge. We can summarize the process of log block merge preemption as follows:

1. check preemptability.
2. suspend the merge operation.
3. write urgent data.
4. clean up the suspended merge operation.
5. resume the normal write request.

To preempt a log block merge operation, the required free space should not be larger than the available free space. For a simple explanation, assume that the maximum number of pages of the urgent write request $(\pi(r_u))$ is the number of flash memory pages of one block $(N_{block})$, i.e., $\pi(r_u) \leq N_{block}$. When the merge operation for the log block $L_v$ is preempted by an urgent request, the required free pages for log block merge preemption are the sum of the number of valid pages of $L_v$ $(N_{valid}(L_v))$ and the number of pages of the urgent write request $(\pi(r_u))$. If the log block $L_v$ is an SLB, $N_{valid}(L_v)$ is 0 since we need no free space to perform a partial merge for the SLB. In addition, we can get free
pages from random log blocks since the partial merge can be invoked even when there are free pages in the random log blocks (RLBs). Therefore, the preemptability condition is as follows:

\[ N_{\text{need}}(L_v) + \pi(r_i) \leq N_{\text{block}} + N_{\text{allowed}}(L_v) \quad (3) \]

where \( N_{\text{need}}(L_v) = \begin{cases} 0, & \text{if } L_v \text{ is an SLB;} \\ N_{\text{valid}}(L_v), & \text{if } L_v \text{ is a RLB;} \end{cases} \)

and \( N_{\text{allowed}}(L_v) = \begin{cases} 0, & \text{if } L_v \text{ is a RLB;} \\ N_{\text{free}}, & \text{if } L_v \text{ is an SLB;} \end{cases} \)

\( N_{\text{free}} \) is the number of free pages in the random log blocks.

If the preemptability condition is not satisfied, the urgent request cannot preempt the log block merge operation and more sub-merge operations should be performed until the condition is satisfied. As each sub-merge operation is performed, the value of \( N_{\text{valid}}(L_v) \) is decreased. In order to minimize the waiting time, the order of sub-merge operations is important. It is better to merge the data block which has a larger number of valid pages within the victim log block before other data blocks.

3.3 Worst Case Waiting Time

In order to use pFAST for real-time applications, the worst-case waiting time should be modeled. If the preemptability condition is satisfied when an urgent request arrives, the worst-case waiting time is the time for one sub-full merge operation. During the sub-full merge operation, \( N_{\text{block}} \) number of pages are copied at the worst-case and one data block is erased. Therefore, the worst-case time for one sub-full merge operation \( W \) is \( N_{\text{block}} \cdot C_{\text{copy}} + C_{\text{erase}} \). When the preemptability condition is not satisfied, the worst-case waiting time is determined by the number of sub-full merge operations until the preemptability condition is satisfied. At each sub-full merge, \( N_{\text{valid}}(L_v) \) in Equation (4) is decremented. The worst case is when the value is decremented by one at each sub-full merge. Therefore, the maximum number of sub-full merge operations is as follows:

\[ N_{\text{SM}}(r_i) = N_{\text{valid}}(L_v) + \pi(r_i) - N_{\text{block}} \quad (6) \]

\( N_{\text{SM}}(r_i) \) has its maximum value of \( \pi(r_i) \) when \( N_{\text{valid}}(L_v) \) is equal to \( N_{\text{block}} \). Therefore, the worst-case waiting time of the urgent request \( r_i \) is \( \pi(r_i) \cdot W \), whose value depends on the urgent request size.

4. EXPERIMENTS

We evaluated the performance of pFAST scheme using simulation. We used a single-level-cell (SLC) NAND flash memory model, where a block consists of 64 pages, and the times for page read, page program and block erase are 25 \( \mu \)s, 200 \( \mu \)s and 2 ms, respectively.

Four kinds of normal I/O workloads are used for evaluation. IE trace is the storage I/O trace extracted running web browser which generates many random accesses on small files. PCapp is collected running several desktop applications, such as document editor, music player, web browser and game in Microsoft Windows XP-based desktop PC. Phone-1 and Phone-2 are collected under the FAT32 file system executing real mobile phone applications such as SMS, PIMS, media player, game, and web browser at the

![Figure 5: The average waiting time of urgent requests at pFAST.](image)

![Figure 6: The maximum waiting time of urgent requests at pFAST.](image)
Figure 7: The CDFs of urgent request waiting times at pFAST.

Figure 8: The average waiting time of urgent requests varying the log buffer size at pFAST.

PCapp have random request patterns, they are more sensitive to the log buffer size.

Figure 9: The response time change ratios of non-urgent request under the preemptive scheme.

KAST [13] FTL which limits the maximum associativity of log block for real-time systems.

6. REFERENCES

5. CONCLUSIONS
Flash memory is a suitable storage device for real-time systems due to its non-fluctuating seek time. However, the garbage collection invokes a fluctuating response time under hybrid mapping FTLs. The proposed preemptive log buffer-based FTL for real-time systems, called pFAST, enables an urgent write request to preempt the log block merge operation by a normal write request. The urgent write requests have better response times under pFAST compared to the current FAST FTL scheme. In addition, it also improves the response time of normal request by suspending the log block merge operation.

As a future work, we will study a preemptive version of
Compression Support for Flash Translation Layer

Youngjo Park and Jin-Soo Kim

Department of Embedded Software
Sungkyunkwan University (SKKU), Suwon 440-746, South Korea
{thepark, jinsookim}@skku.edu

ABSTRACT
NAND flash memory has many advantageous features as a storage medium, such as superior performance, shock resistance, and low-power consumption. However, the erase-before-write nature and the limited number of write/erase cycles are obstacles to the promising future of NAND flash memory. An intermediate software layer called Flash Translation Layer (FTL) is used to overcome these obstacles.

In this paper, we present a flash translation layer called zFTL which has data compression support to reduce the amount of data written to flash memory. zFTL is based on page-level mapping, but it is extended to support on-line, transparent data compression and decompression. We implement zFTL on the MTD layer of the Linux kernel. Our experimental results with five workloads from Linux and Windows show that zFTL improves the write amplification factor by 27% ~ 92%.

Categories and Subject Descriptors
General Terms
Design, Reliability, Performance, Experimentation

Keywords
Flash Memory, Flash Translation Layer (FTL), Storage Systems, Data Compression, Embedded System

1. INTRODUCTION
Recently, NAND flash memory has become a necessity not only in mobile devices but also in high-end laptops and desktop systems, thanks to its superior performance, shock resistance, and low-power consumption. With technology advancing, NAND flash memory’s capacity is getting larger and its price is getting lower.

However, NAND flash memory has several limitations: 1) The previous data should be erased before a new data can be written in the same place. This is usually called erase-before-write characteristic. 2) Normal read/write operations are performed on a per-page basis, while erase operations on a per-block basis. The erase block size is larger than the page size by 64-128 times. In MLC (Multi-Level Cell) NAND flash memory, the typical page size is 4KB and each block consists of 128 pages. 3) Flash memory has limited lifetime; MLC NAND flash memory wears out after 5K to 10K write/erase cycles.

The aforementioned limitations are effectively hidden through the use of an intermediate software layer called Flash Translation Layer (FTL). Most FTLs employ address remapping, which writes an incoming data into one or more pre-erased pages and maintains the mapping information between the logical sector number and the physical page number. As the new data is written, the previous version is invalidated, and those obsolete pages are collected and then eventually converted to free pages via the procedure known as garbage collection. To cope with the limited write/erase cycles, FTLs also perform wear-leveling which distributes erase operations evenly across the entire flash memory blocks [1, 2].

Although garbage collection and wear-leveling improve the overall performance and lifetime, they cause additional writes. One way to quantify the added cost of an FTL is to measure the write amplification factor (WAF). The WAF is defined as the ratio of actual data written into NAND flash memory as compared to the actual data written by the host system. A lower WAF is a measure of efficient storage and housekeeping algorithms inside FTL, improving the overall life expectancy of NAND flash memory by lowering the total write/erase cycles required to manage the data stored in flash memory [3]. Basically, the WAF of hard disks is 1.0. The WAF can be as high as 10 on low-end SSDs, while Intel claims that its X25-M SSD keeps the WAF down to 1.1 [4].

In this paper, we present the design and implementation of a flash translation layer called zFTL, which internally compresses or decompresses data. Data compression is an effective way to lower the WAF further down to below 1.0, thus improving FTL performance and lengthening flash lifetime. Specifically, this paper discusses and evaluates several design issues arise when we support on-line, transparent compression/decompression inside FTL. zFTL is based on page-level address remapping [5] and the compression unit size is set to 4KB. We focus on the management of the compressed data, assuming the actual compression/decompression is done by dedicated hardware. We consider two compression algorithms, namely Zlib [6] and LZ77 [7].

We implement zFTL on the MTD layer of the Linux kernel 2.6.32.4 and evaluate it on NANDSim [8], which emulates the behavior and timing of NAND flash memory with RAM. zFTL is evaluated with five realistic workloads from Linux and Windows.
Our results show that the use of data compression improves the WAF up to 92%.

The rest of the paper is organized as follows. The next section discusses the related work. Section 3 introduces the overall architecture and design issues of zFTL. Section 4 presents the experimental results and Section 5 concludes the paper.

2. RELATED WORK
Data compression techniques have been studied in various layers in computer systems. JFFS2 [9] is a representative flash-aware file system inspired by the log-structured file system [10]. JFFS2 provides an option to use zlib-based data compression. CramFS [11] and SquashFS [12] are compressed read-only file systems, mainly targeting the root file system in small embedded systems. Hyun et al. [13] proposed LeCramFS which modifies CramFS for NAND flash memory. These flash-aware file systems do not require FTL, as they work directly on NAND flash memory.

Yim et al. [14] studied a flash compression layer for SmartMedia card system, proposing an internal packing scheme (IPS) to manage internal fragmentation. The IPS best-fit scheme can reduce the internal fragmentation effectively, but it may incur some read overhead as unrelated logical sectors are packed together to minimize internal fragmentation [14]. Chen et al. [15] proposed another internal packing scheme called IPS real-time. The IPS real-time scheme splits the compressed data and stores them into two consecutive flash pages, but it has no consideration for random reads; it needs to access two flash pages to read a sector which spans two pages. Both approaches focused only on reducing the internal fragmentation, without considering other issues such as mapping information management and garbage collection under the presence of compressed data. In addition, they are devised for old 512-byte flash page size, which has been outdated by new generations of flash memory chips. An SSD controller from SandForce Inc. is known to use real-time data compression and data deduplication to lower the WAF as low as 0.5 [4]. However, its internal architecture has not been published in detail.

Special hardware compressor/decompressor engines have been proposed in several literatures. IBM’s Memory Expansion Technology (MXT) [16] performs compression and decompression between the shared cache and the main memory, to expand the effective main memory size using hardware implementation of the LZ77 algorithm [17]. Benini et al. [18] investigated a hardware-assisted data compression for memory energy minimization. They describe the implementation of hardware compression algorithms including LZ-like one in detail and show no penalty in performance. Kjelso et al. [19] proposed the X-Match compression algorithm for main memory, which is easy to implement in hardware. X-Match is another variant of LZ77, differing in that phrases matching works in four bytes unit [20]. For brevity, we assume data compression/decompression is assisted by special hardware that is fast enough to hide its overhead.

3. zFTL
3.1 System Architecture
Figure 1 shows the overall architecture of zFTL. File systems issue read/write requests to zFTL. The size of each request is a multiple of the disk sector size (512 bytes). For write requests, zFTL aggregates the requested data in the write temporary buffer, whose size is equal to the compression unit size. If the write temporary buffer is full, the data in the buffer is compressed and then appended into the flash write buffer. The size of the flash write buffer is a multiple of the flash page size (4KB in MLC NAND). A single flash page size may hold a number of compression units depending on the compression ratio.

When the flash write buffer has not enough space for the incoming compressed data, the write buffer is flushed into flash memory. Before flushing data, the corresponding logical sectors are remapped to new physical pages by zFTL. In case the previous data is available in any of buffers, it is removed from the buffer, ensuring data consistency and preventing the invalidated data from being flushed into flash memory. If the number of free blocks is below a certain threshold, zFTL initiates garbage collection to reclaim erase blocks. We will discuss the garbage collection process of zFTL in detail in Section 3.5.

For reads, zFTL first searches the requested data in the write temporary buffer as it has, if any, the most recent version of the data. When the search fails, zFTL looks up the data in the flash write buffer. If the data is found in the flash write buffer, the compressed data is decompressed and then loaded into the read temporary buffer. When the data is still not found in the flash write buffer, zFTL examines the read temporary buffer and the flash read buffer. Note that the two write buffers should be looked up before the two read buffers, as they may keep the up-to-date data. While the requested data is stored in any of these buffers, the read request can be satisfied without issuing any flash read operations. Otherwise, zFTL needs to decompress the requested data after reading the corresponding page from flash memory. The detailed read/write flows in zFTL are depicted in Figure 2.

3.2 Compression Unit
The unit of data compression is an important factor affecting the compression ratio and speed. In particular, dictionary-based algorithms such as LZ77 have the characteristic that the bigger compression unit tends to yield the better compression ratio. This is because these algorithms replace a repeated pattern of strings within the compression unit by a much shorter but uniquely identifiable string.

We have considered two options related to the unit of compression. One is to compress the variable-sized data as a whole as it is delivered by a single write request from the file system. In Linux, the number of sectors written by a write request is usually a multiple of the file system block size and can be as large as 256 sectors (i.e., 128KB) for sequential writes. Thus, this scheme can improve the overall compression ratio and reduce the number of mapping entries. However, the use of the variable-sized compression unit presents a number of issues that need careful handling. For example, when a portion of the compressed data is read by a read request, the entire compressed data should be fetched from flash memory for decompression. An even worse scenario occurs when the compressed data is partly updated by a later write operation. In this case, the original data should be merged with the new data after decompression. Then, it can be either recompressed and stored into flash memory as a single compression unit, or split into two or three pieces each of which is separately compressed and stored.
Another option is to compress a fixed size of data at a time. In fact, any power of two multiple of the sector size, such as 512B, 1KB, 2KB, 4KB, 8KB, etc., can be used as the compression unit size. As discussed before, the use of larger compression unit size is favored for better compression ratio. However, if the compression unit size becomes too large, the system suffers from unnecessary overhead when the compressed data is partly read or updated. Moreover, enlarging the compression unit size has a diminishing return in the compression ratio. Burrows et al. [21] and Yim et al. [14] have shown that there is no significant difference in the compression ratio for 2KB to 8KB compression unit sizes.

For the above reasons, zFTL uses a fixed compression unit size of 4KB (simply called a block, for short). Since most file systems in Linux use 4KB as the file system block size, they rarely issue I/O operations smaller than this size and the read/write request sizes are usually a multiple of 4KB. In addition, the compression unit size of 4KB is large enough to achieve good compression ratio.

3.3 Compression Algorithms

The choice of compression algorithms is also one of the important design issues, because it determines the speed of compression/decompression, the compression ratio, and the complexity of hardware implementation. However, the efficiency of various compression algorithms is beyond the scope of this paper. We currently implement zFTL with Zlib [6] and LZ77 [7] algorithms. Both algorithms are very well known for their performance and reliability. Especially, efficient hardware implementations of LZ77 or variants have been proposed in several previous studies.

3.4 Address Mapping

zFTL is based on page-level mapping where a per-page mapping entry from the logical page number to the physical flash page number is maintained in the Page Mapping Table (PMT). Similar to other FTLs with page-level mapping, PMT is accessed by the logical page number. To support data compression, zFTL extends the structure of PMT slightly. Each 32-bit mapping entry includes the incompressible block flag (FLAG) and the page index (IDX), as well as the physical page number (PPN) where the page is stored. FLAG indicates whether the corresponding logical page is compressed or not. Since a single flash page may accommodate compressed blocks from several logical pages in zFTL, IDX is used to represent the relative position of each logical page within the physical page. Figure 2 illustrates an example of PMT in zFTL. Note that PMT entries for the logical page number 100, 101, and 102 have the same value for the PPN field, representing that the data for those logical pages are compressed and stored in the same physical page number 100 in the order indicated by the IDX value.

Some data are inherently incompressible as they come from multimedia files or compressed files. It is pointless to compress these data again inside zFTL as it will not save any space. zFTL identifies these incompressible blocks based on the resulting size after compression. If the size of a block after compression is not small enough, the block is stored in flash memory as is, setting the corresponding FLAG to 1 (cf. the PMT entry of the logical page number 103 in Figure 2).

Depending on FLAG, the physical flash page has two different structures. For incompressible blocks (FLAG = 1), the entire page is devoted to the (uncompressed) original data block. When the page size is larger than the compression unit size, each data block is identified by IDX. On the other hand, when the value of FLAG is 0, the related physical page includes such information as the total number of compressed blocks in the page, a set of offsets for each compressed block, and a set of compressed blocks, as depicted in Figure 2. The offset indicates the last byte position of the corresponding compressed block in the page.

3.5 Garbage Collection

As in other FTLs, zFTL reserves a set of erase blocks (5% of the total erase blocks, by default) to absorb the incoming write requests. When zFTL runs out of available erase blocks, garbage collection is invoked to reclaim the space allocated to obsolete pages. zFTL uses the greedy policy to choose a victim erase block, i.e., the erase block which has the smallest number of valid pages is selected as a victim. During garbage collection, the remaining valid pages in the victim erase block are copied into another erase block and the victim erase block is cleared to be used later.

Since each physical page normally contains the data from more than one logical page in zFTL, it can be partially invalidated by subsequent write operations. Therefore, zFTL should be able to identify the current status of each compressed data stored in the
same physical page, in order to copy only the valid data during garbage collection. For this reason, zFTL maintains the Page Status Table (PST) in memory. Unlike PMT, PST is indexed by the physical page number, and each PST entry keeps track of the number of valid logical pages and the bitmap for each logical page stored in the given physical page number. The bitmap indicates whether the corresponding logical page is valid or not.

<table>
<thead>
<tr>
<th>PPN 100</th>
<th># of valid pages</th>
<th>Bitmap for valid pages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 1 0</td>
<td>0 1 1 0 0</td>
</tr>
</tbody>
</table>

Figure 3. An example PST (Page Status Table) entry

Figure 3 shows an example 8-bit PST entry designed for 4KB physical pages. Figure 3 represents that two logical pages (the second and the third one) are currently valid in the physical page number 100. Under this PST structure, up to five logical pages can be packed into a 4KB physical page. Our experiments show that about three compressed logical pages are stored in a single 4KB flash page on average for the most well-compressed workloads. Thus, we believe the 8-bit entry is sufficient for 4KB flash pages. If the page size is increased, we can add a few more bits to each PST entry.

3.6 Internal Fragmentation

The flash page size is fixed whereas the resulting data block size varies after compression. Unless we allow the compressed block to be stored in more than one page, internal fragmentation is unavoidable. The relative amount of internal fragmentation will be getting smaller as the page size becomes larger than the compression unit size. Considering the recent trend in NAND flash memory architecture where the page size grows progressively larger, the impact of internal fragmentation can be of minor significance, compared to the benefit of compression support.

Currently, zFTL does not implement any special scheme to reduce internal fragmentation. zFTL simply packs the incoming data in the order they are issued from the upper layer. We leave a more comprehensive analysis and possible optimization on internal fragmentation for future work.

3.7 Memory Requirement

The memory requirement of zFTL is comparable to other FTLs with page-level mapping. The use of block-level mapping can decrease the memory requirement by a factor of 64–128, but the increasing number of SSDs are adopting page-level mapping due to its superior performance and higher flexibility. Since other page-mapping FTLs also keep page-level address mapping information in memory (i.e., PMT in zFTL), only the memory used by PST is the added cost in zFTL, which requires 512KB for 2GB flash memory with 4KB page size.

If PMT and PST are too large to be accommodated in memory, zFTL may use the selective caching method used in DFTL [5], where the whole mapping table is stored in flash memory and only the needed part of the mapping table is loaded into memory.

4. EVALUATION

4.1 Experimental Setup

We evaluate the performance of zFTL on an x86-based Linux system equipped with Intel Core2Duo E8400 3.0GHz CPU, 4GB DDR2 DRAM, and 64-bit Ubuntu 9.04 distribution. zFTL is implemented as one of block devices in the MTD (Memory Technology Devices) layer of the Linux kernel 2.6.32.4, on top of which the ext4 file system is mounted. The compression support can be turned off anytime using the /proc interface. Instead of bare NAND flash chips, we use the MTD NANDSim module which emulates the behavior and timing of NAND flash memory with the host RAM. We configure the parameters of NANDSim to model a 2GB MLC NAND flash memory where the page size is 4KB and each erase block has 128 pages. The latency of read, write, and erase operation is set to 60usec, 800usec, and 1.5ms, respectively [22].

<table>
<thead>
<tr>
<th>Workload</th>
<th>Write Requests</th>
<th>Read Requests</th>
<th>Sectors Written</th>
<th>Sectors Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNTAR</td>
<td>116,922</td>
<td>15,722</td>
<td>935,376</td>
<td>125,776</td>
</tr>
<tr>
<td>COMPILE</td>
<td>112,775</td>
<td>32,739</td>
<td>902,200</td>
<td>261,912</td>
</tr>
<tr>
<td>TEMP</td>
<td>145,388</td>
<td>7</td>
<td>1,163,104</td>
<td>56</td>
</tr>
<tr>
<td>WINDOWS</td>
<td>45,788</td>
<td>4</td>
<td>366,304</td>
<td>32</td>
</tr>
<tr>
<td>OFFICE</td>
<td>243,503</td>
<td>8,284</td>
<td>1,948,024</td>
<td>66,272</td>
</tr>
</tbody>
</table>

Table 1 shows the basic information of five workloads used in this paper. UNTAR and COMPILE are the real workloads executed on ext4/zFTL, which untar and compile the source code of the Linux kernel 2.6.32.4, respectively. TEMP denotes the set of files downloaded from the Internet while the Firefox web browser visits such sites as Facebook, E-bay, Amazon, Yahoo, Youtube, etc. We periodically collect the files in the browser’s temporary directory and then copied them onto zFTL.

WINDOWS and OFFICE workloads are mainly used to investigate the compression ratios of the files used in Windows XP. Since zFTL is implemented only on the Linux platform, we first extract file system access traces using the ProcessMonitor tool [23] while Excel, Word, and Powerpoint from Microsoft Office 2007 are installed on Windows XP. Then, the installation process has been mimicked on Linux using the following steps: 1) The trace is postprocessed to identify the existing Windows files which are touched during the installation. Those files are copied to zFTL (WINDOWS). 2) File access traces obtained from Windows are replayed (OFFICE).

To model the aged file system, we initialize zFTL by running Postmark 1.51 [24] before each experiment. Postmark is configured with 25K files, 50K transactions, and file sizes ranging from 30KB to 80KB. The total amount of data written by Postmark is about 3GB. During this preconditioning phase, we turn off the compression support in zFTL.

4.2 Average Compression Ratio

Figure 4 shows the average compression ratios for each workload with Zlib and LZ77 algorithms. The compression ratio is defined as the ratio of the compressed block size to the...
uncompressed data block size (4KB)\(^1\). The compression ratio varies from workload to workload, but Zlib shows slightly better compression ratios than LZ77 in general. Workloads which manipulate text-based files such as UNTAR and COMPILE exhibit fairly good compression ratios as low as 27% with Zlib. On the other hand, TEMP shows the worst compression ratio since most files are JPEG files and movie clips which have been already compressed. We found that Windows files touched during the installation of Microsoft Office also reveal good compression ratios. The compression ratio of OFFICE is higher than that of WINDOWS by 29% (Zlib) or 24% (LZ77). This is because OFFICE handles many CAB files which are in the Microsoft compressed archive format.

UNTAR and COMPILE show very low WAFs under zFTL due to their low compression ratios. Since the amount of data written into NAND flash is reduced effectively, garbage collection hardly occurs. As a result, their WAFs are improved by 88% (UNTAR) and 92% (COMPILE) with the Zlib algorithm. The WAFs for TEMP, WINDOWS and OFFICE are also improved by 36%, 46% and 62%, respectively, with Zlib. As expected, TEMP shows the least performance improvement. The LZ77 algorithm performs slightly worse than Zlib, resulting in improvements in WAFs by 27% (TEMP) ~ 89% (COMPILE).

4.4 Garbage Collection Overhead
Figure 6 presents the total time spent for garbage collection. It is estimated by multiplying the number of flash read, write, and erase operations during garbage collection by the respective operational latency of MLC NAND flash memory. The final results are normalized to the values obtained when the compression support is disabled.

In UNTAR and COMPILE workloads, the garbage collection overhead is almost negligible because of good compression ratios. TEMP has the largest overhead, but it is still better than the case without any compression. We observe that the overall trend of Figure 6 is highly correlated to that of Figure 4.

4.5 Internal Fragmentation

\(^1\) The compressed block size includes the housekeeping information such as the number of compressed blocks and the offset of each compressed block, but does not include the wasted space in a page due to the internal fragmentation.
Figure 7 illustrates the average percentage of wasted space in a flash page. The amount of wasted space is measured when the contents of the flash write buffer is flushed into NAND flash memory. The overall tendency is that the percentage of wasted space gets smaller as the page size is increased from 4KB to 32KB. The sudden increases in UNTAR and COMPILE for the 32KB page size are due to the limit in the bitmap size of the PST entry. When the page size is 32KB, we expand the bitmap size to 32 bits so that each page can hold up to 32 compressed blocks. However, this was not sufficient for UNTAR and COMPILE.

One way to improve space utilization is to use multiple flash write buffers. With more than one flash write buffer, the incoming compressed block is more likely to find a flash write buffer that can accommodate its data. However, according to our experiments, this scheme reduces the wasted space only by up to 7% with the 4KB page size. The benefit gets even smaller as the page size becomes larger. Moreover, the use of multiple flash write buffers may harm the sequential read bandwidth since the sequentially-written data can reside in different flash pages.

5. CONCLUSION AND FUTURE WORK
Due to inherent characteristics of NAND flash memory which does not allow in-place update and wears out after repeated write/erase cycles, flash translation layers have been using a variety of techniques to enhance the overall performance and endurance. Many previous researches on flash translation layers have focused on efficient address mapping and garbage collection schemes. However, another orthogonal issue that can reduce the amount of data written into NAND flash memory is to support data compression inside the flash translation layer.

In this paper, we present zFTL, a flash translation layer which supports on-line, transparent data compression. We have examined several design issues to support data compression in flash translation layer, including some required extensions in address mapping and garbage collection. We have implemented zFTL in the MTD layer of the Linux kernel. Through the use of real and emulated workloads, we confirm that zFTL improves the write amplification factor by up to 92%.

Our future work includes the analysis of hardware compressor/decompressor engine in terms of cost, performance and energy consumption. We also plan to evaluate zFTL with more diverse workloads.

6. REFERENCES
[22] Samsung Elec., 2G8X Bit NAND Flash Memory, (K9GAG08U0M-P), 2006.