Real-Time Flash Translation Layer for NAND Flash Memory Storage Systems

Zhiwei Qin Yi Wang Duo Liu Zili Shao
Department of Computing
The Hong Kong Polytechnic University
Hung Hom, Kowloon, Hong Kong
{cszqin, csywang, csdliu, cszlshao}@comp.polyu.edu.hk

Abstract

Due to the variable garbage collection latency, NAND flash memory storage systems may suffer long system response time, especially when the flash memory is close to be full. Most of existing flash translation layer (FTL) schemes focus on improving the average response time but ignore to provide a desirable worst case response time upper bound. This paper proposes a Real-time Flash Translation Layer (RFTL) scheme to hide the long garbage collection latency while satisfying a worst case response time upper bound that achieves an ideal case. We achieve this by using a distributed partial garbage collection policy that enables RFTL to reclaim the space and to serve the write requests simultaneously. A new block-level address mapping approach is designed to guarantee enough free space to serve the write request arriving at any time period. Experimental results show that our scheme improves both the worst case system response time and the average system response time compared with previous work.

1 Introduction

No matter in mission-critical hard real-time systems such as aerospace [3] and military or in soft real-time systems such as iPads and smart phones, NAND flash memory has become an essential part due to its unique characteristics, such as non-volatility, low power-consumption and fast access time. However, in NAND flash, one page once be written cannot be overwritten until this page is erased (out-of-place update). The erase operation can only be performed in the unit of one block (bulk-erase). These properties make the response time become unpredictable. Most of existing FTL schemes focus on improving the average performance but ignore the real-time storage performance. In this paper, we propose a real-time FTL scheme which can provide a worst case response time upper bound that is close to an ideal case for I/O requests in NAND flash storage systems.

Flash translation layer is a block-device-emulation software layer that simulates NAND flash as a hard disk by hiding the “out-of-place update” and “bulk-erase” properties. One function of FTL is to do address mapping between a logical address in file systems to a physical address in flash media. Another important function is to reclaim the space by erasing obsolete blocks in flash, also known as *garbage collection*. Garbage collection will be invoked if there are not enough free space to serve the requests. Given a read/write request issued from file system, the best case response time is constant since no garbage collection is invoked. However, in the worst case, a request will be blocked by the time-consuming garbage collection. It consequently suffers a long latency which might be intolerable for mission-critical real-time applications. Therefore, how to design a service guaranteed FTL scheme for real-time applications becomes an important problem.

In the previous work, several techniques have been proposed to solve this problem. Chang et al. [7] is the first in proposing real-time garbage collection for flash memory storage systems, where predictable performance is guaranteed by ensuring enough free space is always available for write requests. Although a response time upper bound can be obtained, their approach suffers slow worst case response time and requires extra file system support. Choudhuri et al. [9] proposed a flash translation layer called GFTL to guarantee a response time upper bound. GFTL reduces the upper bound by adding extra blocks as the write buffer and using a partial block cleaning policy to hide the long garbage collection latency. In order to provide enough free space to serve write requests, the full blocks are centrally organized in a garbage collection queue, and the garbage collection are consecutively performed as long as the queue is not empty. GFTL guarantees a worst case response time for write request, however, it suffers a slower worst case response time for read requests. Moreover, it introduces a large amount of extra page copy operations which significantly degrade the average system response time. Since garbage collection dose not occur very often, a scheme should not sacrifice too much average response time when reducing the worst-case response time. We address this problem in this paper.

In this paper, we propose a real-time flash translation layer, called RFTL, which provides not only an ideal worst case response time upper bound but also faster average response time. A distributed partial garbage collection policy is applied in RFTL. Different from the centralized partial garbage collection policy [9] in which all full blocks are put into a queue and garbage collection is performed in a centralized manner, in RFTL, garbage collection is distributed into each logical block and a full block is reclaimed according to the arrival sequence of write requests in a distributed manner. The condition to invoke one partial garbage collection step is when a write request arrives, and the corresponding requested data block is full. Since a write request is served immediately after one partial garbage collection step, the worst case response time of a request is only the overhead to perform one partial garbage step. Moreover, in a logical block, the garbage collection of a full block is performed only when there is a write request to the logical block; therefore, many unnecessary valid page copies and block erase operations are avoided so as to significantly improve the average system response time. Compared with GFTL, our approach does need more flash memory space; however, it effectively reduces the more valuable RAM cost. To the best of our knowledge, this is the first work to reduce both the average response time and worst case response time by applying a distributed partial garbage collection policy in NAND flash memory storage systems.

We evaluate our scheme with a set of benchmarks running on a NAND flash memory simulator we developed under Linux kernel 2.6.17. The experimental results show that our scheme can achieve a 36.30% improvement in the worst case response time compared with GFTL. Moreover, we make a trade-off between the flash space and the average system response time. By using doubled flash space of GFTL, our scheme shows a 91.79% reduction in more valuable RAM space and a 67.06% improvement in average system response time compared with GFTL.

The rest of this paper is organized as follows. Section 2 shows background and related work. In Section 3, we give the problem formulation. Section 4 presents our scheme and the WCET analysis. In Section 5, we present the performance evaluation of our scheme and finally we give the conclusion and future work in Section 6.

2 Background and Related Work

In this section, we first present a NAND flash storage system architecture in Section 2.1. Then we show the related work and revisit the implementation of some representative FTL schemes in Section 2.2.

2.1 NAND Flash System Architecture

A NAND flash chip consists of multiple blocks, and each block is composed of a fix number of pages. A page is the basic unit for read/write operation, while a block is the minimum unit for erase operation. A page contains the data area and the spare area also known as the Out Of Band (OOB) area. The OOB area is used to store some house-keeping information (e.g., error correction code) of the corresponding page. The typical size of a page is 512 bytes for older, small block NAND flash, and 2KB for new large block NAND flash. A read/write request can read/write an entire page or an OOB area. Table 1 depicts the basic operation specifications for NAND flash [10].

Table 1. NAND flash specifications.

Characteristics	Samsung 16MB Small Block	Samsung 128MB Large Block
Block size	16KB	64KB
Page size	512B	2KB
OOB size	16B	64B
Read page	$36\mu s$	$25\mu s$
Read OOB	$10\mu s$	$25\mu s$
Write page	$200\mu s$	$300\mu s$
Erase	$2000\mu s$	$2000\mu s$

A typical NAND flash memory storage system architecture is shown in Figure 1. In NAND flash storage systems, the flash translation layer is a software layer built between the MTD (Memory Technology Device) layer and the file system layer. The MTD layer can do primitive read, write and erase operations on flash [1]. The flash translation layer provides three components: address translator, garbage collector, and wear-leveler. Address translator is responsible for address translation between a logical address applied in file system and a physical address adopted in flash memory chip. It maintains an address mapping table in RAM. According to the "out-of-place update" property, one page once be written cannot be overwritten. The rewritten data has to be written to a new free page, the old page becomes invalid (called an invalid-page) and the new page is considered as a valid-page. Garbage collector reclaims space by erasing obsolete blocks, which includes a series of page read, page write and block erase operations. In NAND flash memory, blocks have a limited erase lifetime. Wear-leveler is a component that distributes erase counts evenly across all blocks, so as to extend the lifetime of flash memory. Our work focuses on the FTL design for real-time data storage based on the architecture shown in Figure 1.

When file system layer issues a read or write request with a logical address (sector) to NAND flash memory, address translator locates the corresponding physical address $<$*block, page*$>$ by searching the address mapping table. This procedure is called *address translation*. The time cost

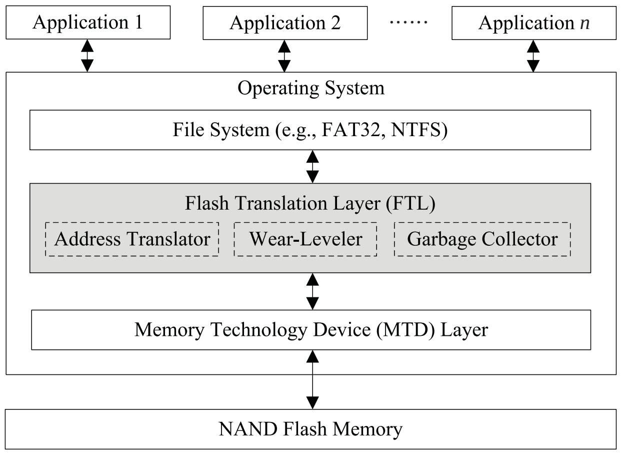

Figure 1. Architecture for NAND flash storage system.

in this procedure is the address translation overhead. According to the "out-of-place update" property, if a logical address is mapped to a physical address that contains valid data, the updated data should be written to a free physical page. The mapping table should then be updated due to the newly changed address mapping. In this process, we define *system response time* of a request as the time cost from the request issued from file system to the finishing time of the requested operation. System response time is determined by the address mapping approach and the garbage collection policy. It reflects the efficiency of the FTL scheme.

2.2 Related Work

In the past decades, several techniques have been proposed to improve the system performance of NAND flash storage systems [4, 6, 11, 12, 16]. Some techniques exploit different system architectures and some techniques explore the flash translation layer designs. In particular, Ha et al. [12] proposed a new architecture exploration methodology for NAND flash based multimedia card to improve the throughput and the cost for low-end embedded systems. Lee et al. [11] used NAND flash memory to store program codes, and proposed a new paging technology that allows program codes stored in NAND flash memory to be executed satisfying real-time requirements with the minimal usage of SRAM. Different from the above work, we aim to design a new FTL scheme to provide a guaranteed real-time service for data accessing in NAND flash storage systems.

In terms of FTL design, three kinds of address mapping schemes have been proposed: the page-level FTL [4, 10, 13], the block-level FTL [5, 15] and the hybrid-level FTL [8, 14, 16]. In page-level FTL [4], one logical page (sector) is mapped to one physical page. Since a large mapping table is involved, page-level FTL is inapplicable for RAM-constrained embedded systems. The garbage collection in page-level FTL is invoked when the NAND flash runs out of space, and each time only one victim block will be reclaimed. In general, the block with the least valid pages is taken as the victim block. The victim block will be erased after the valid pages are copied into a new free block. Suppose one block consists of π pages and the victim block has M valid pages ($\pi \geq M \geq 0$), the time overhead to reclaim the victim block is $M*(T_{rdpg}+T_{wrpg})+T_{er}$, where T_{rdpg} is the time to read a page, T_{wrpg} is the time to write a page, and T_{er} is the time to erase a block.

In block-level FTL schemes [5], a logical page number (LPN) is made up of a logical block number (LBN) and a block offset (BO). One logical block is mapped to a physical block (called *primary block*). In case of a rewrite operation (or if the primary block is full), a new physical block (called *replacement block*) is chosen to serve the write requests. Since the mapping table only maintains the mapping information between the blocks, it has lower RAM requirements. However, it suffers worse address translation efficiency. The garbage collection in a block-level FTL is invoked once both primary block and replacement are full. They will be erased after being merged into a new free block. Since two blocks are involved in this process, the garbage collection latency is much longer in the worst case compared with the one in page-level FTL.

In hybrid-level FTL schemes [8, 14, 16], physical blocks are logically partitioned into data blocks (primary blocks) and log blocks (replacement blocks). Data block is used to store the first written data, while the updated data is stored in log blocks. In a merge operation in hybrid-level FTL schemes, valid pages scattered on data block and its corresponding log blocks are copied into a free block. Hybrid-level FTL shows better address translation efficiency, however, it also suffers long worst case response time due to the two-blocks merge operations. As a good supplement for above schemes, we focuses on designing a hybrid-level FTL scheme with the objective to hide the long garbage collection latency and provide a deterministic response time.

3 Problem Formulation

The non-deterministic response time of requests in NAND flash memory is caused by the variable garbage collection latency. Figure 2 shows an illustration example of garbage collection (GC) process in page-level FTL schemes. For the sake of illustration, we assume that each block consists of 8 physical pages. In Figure 2, the victim block consists of 5 valid pages. These valid pages are copied to a new free block. After that, all pages in the victim block become invalid and the victim block is then erased for reuse. Based on the specifications of a small block NAND flash shown in Table 1, the time overhead to reclaim this block is $5*(36+200)+2000=3180\mu s$. Given a write request, the response time is $200\mu s$ if no garbage collection is triggered. Otherwise, the response time becomes $3380\mu s$ when the request is blocked by the garbage collection with 5 valid-page copy operations. Such long time latency limits the usage of NAND flash in real-time applications. Moreover, since the number of valid pages in different victim blocks are different, the time overhead to reclaim these blocks vary which makes the response time of the requests non-deterministic.

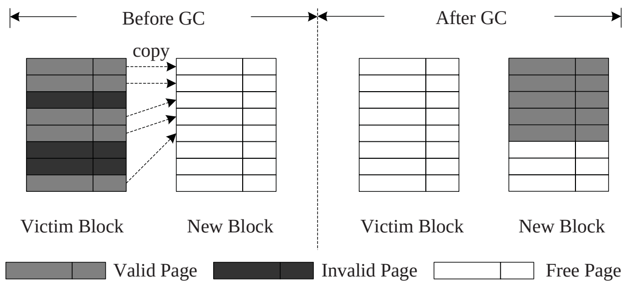

Figure 2. Illustration example of garbage collection.

In order to remove the unpredictability, we model the NAND flash storage system as follows. Each I/O request issued from file system to the FTL is modeled as an independent real-time task $T= \{p, e, d\}$, where p is the period, e is the execution time and d is the deadline. Without loss of generality, we assume that p is equal to d. Multiple I/O requests form a set of real-time tasks $V=\{T_1, T_2, ..., T_n\}$. There are two kinds of tasks in task set V: read request task $T_r=\{p_r, e_r, d_r\}$, and write request task $T_w=\{p_w, e_w, d_w\}$. p_r and p_w denote the frequency of a read or write request arriving from file system. e_r represents the time taken to search for a target page, read the data from the page and return a success or failure to the file system. e_w is the time overhead to search a free page to store the data. The values of e_r and e_w are determined by the specific FTL. A lower bound on p (denoted as $L(p)$) gives the maximum request arrival rate that an FTL can handle. The upper bound on e (denoted as $U(e)$) shows the worst case execution time of requests when no garbage collection is involved. From the perspective of file system, $L(p)$ represents the worst case response time when garbage collection is considered.

For comparison purpose, we first present a hypothetical ideal case as a baseline. In the ideal case, a read/write request task can be executed directly without any garbage collection involved. This is the best case scenario and both the execution time and the response time are constant. Here we only consider the flash operation time overhead since the address translation overhead in RAM operation is at least an order of magnitude less than the flash operation time. The upper bounds on $U(e)$ in the ideal case are shown in Table 2. In the table, T_{rdoob} represents the time to read an OOB of a page. In the worst case scenario, the execution of a read/write request task will be blocked by garbage collection. Note that, T_{er} is the longest atomic operation in flash media since the erase of one block cannot be interrupted. Therefore, T_{er} is the minimum time a request will be blocked and $L(p)$ should be T_{er} in the ideal case.

Table 2. Service guarantee bounds.

Bounds	Ideal	GFTL [9]	RFTL
$U(e_r)$	T_{rdpg}	$T_{rdpg}+\pi T_{rdoob}$	$T_{rdpg}+T_{rdoob}$
$U(e_w)$	T_{wrpg}	T_{wrpg}	$T_{wrpg}+T_{rdoob}$
$L(p)$	T_{er}	$T_{er}+max\{U(e_r), U(e_w)\}$	$max\{T_{er}+U(e_w), U(e_r)\}$

In this paper, we design a real time FTL scheme (called RFTL) which guarantees $U(e)$ for both reads and writes that are marginally T_{rdoob} larger than T_{er}. Our scheme provides service guarantees for requests that have a lower worst case response time ($L(p)$) compared with GFTL [9], since $T_{rdpg}+\pi T_{rdoob}$ tends to be greater than $T_{wrpg}+T_{rdoob}$ according to the NAND flash specifications shown in Table 1.

Based on the model and problem analysis, we formulate the problem as follows:

Given an NAND flash memory chip and a task set $V=\{T_1, T_2, ..., T_n\}$, how to design a FTL scheme that can jointly schedule the requests and corresponding garbage collection operations such that a request can be executed within an upper bound $L(p)$ that is close to T_{er}?

4 RFTL: Real-time Flash Translation Layer

In this section, we describes the technique details for our RFTL scheme. We first present the address mapping approach and task schedule policy in Section 4.1 and Section 4.2, respectively. Then we show the read operation, write operation and garbage collection policy of RFTL in Section 4.3 and Section 4.4, respectively. Finally, we give the WCET analysis in Section 4.5.

4.1 Address Mapping in RFTL

In RFTL, we use a hybrid-level mapping approach. A logical page number (LPN) is divided into a logical block number (LBN) and a block offset (BO). A block mapping table is used to map a logical block to three physical blocks: the *primary block*, the *replacement block* and the *buffer block* as shown in Figure 3. Three indices that direct to the next available page in each block are recorded in the table. The primary block is firstly used to serve the write requests, and the buffer block will serve the pending write requests when the primary block is full, while the replacement block provides a space to reclaim the primary block. These three blocks can periodically change their functions to provide guaranteed space for writes.

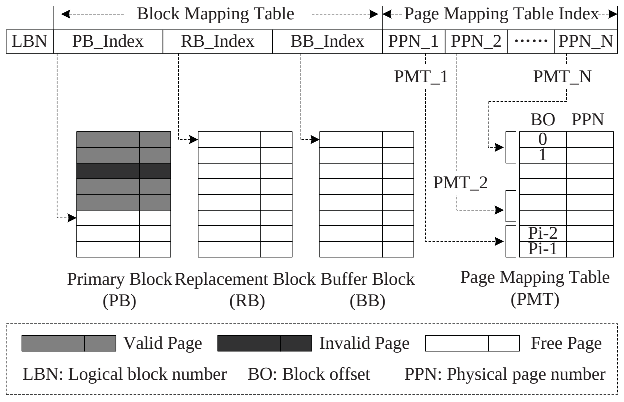

Figure 3. Architecture of RFTL.

For each logical block, a page-level mapping table is used to map a logical page to a physical page that may belong to one of these three physical blocks. In order to reduce the RAM cost, the page mapping table is divided into *N* small tables, and each small table is stored in the OOB area of the newly allocated page. Suppose each logical block and each physical block include π pages, the entire page-mapping table for a logical block has π entries. Assume the OOB area of a physical page can store α ($\pi \geq \alpha > 0$) entries of mapping slots, then the whole page mapping table is divided into *N* subtables according to the logical page number, where $N = \lfloor \pi/\alpha \rfloor$. The *N* page mapping table indices are recorded in the RAM. Using the page-level mapping table indices, RFTL can obtain the address mapping information rapidly by reading one OOB.

4.2 Task Schedule in RFTL

After obtaining the address mapping information, the read/write request should be serviced in three physical blocks. If no garbage collection is involved, RFTL will only execute this request in one period p. Otherwise, if the primary block is full and the garbage collection is invoked. The valid-page copy operations and the erase operation performed on the garbage collection are divided into partial steps, and the time taken to perform each step is no longer than the longest atomic operation in flash (that is the block erase operation T_{er}). In such scenario, RFTL will first execute the request and then serve one partial garbage collection step in one period p.

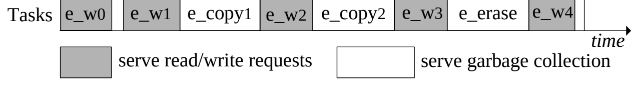

Figure 4. Task schedule in RFTL.

Figure 4 shows the task schedule policy of RFTL, in which the requests and the garbage collection can be alternatively scheduled. Five requests $w0$, $w1$, $w2$ $w3$, and $w4$ are mapped to the same primary block. $w0$ is scheduled directly since free space is available. When the primary block is full, the pending tasks are scheduled in each period p and the time cost to execute each task is e_w1, e_w2 and e_w3, respectively. In the time left for each period, the partial garbage collection operations of this primary block will be scheduled. In Figure 4, there are two copy operations and one erase operation. The time cost of these three operations are e_copy1, e_copy2 and e_erase, respectively. After garbage collection, the primary block becomes free and $w4$ can be scheduled.

4.3 Write Operation and Read Operation

A write request issued from file system is represented by a data and a logical page number (LPN), e.g., *write(D,126)*, where *D* is the data and *126* is the LPN. The LPN is translated to a LBN and a BO. Three physical blocks are mapped to the logical block with LBN. The first write to the LBN is written to the first free page of the primary block, and the pages in the primary block are allocated sequentially from page 0. After π writes, the primary block becomes full, the buffer block will then serve the coming write requests, and the distributed partial garbage collection will be invoked simultaneously to reclaim the primary block. The buffer block serves as the buffer for requests from the time the primary block becomes full until it is reclaimed. The valid pages in the primary block will be copied to the replacement block, where the copy operation can be interleaved with the requests. In the page copy process, a free page is guaranteed to be available in the buffer block to serve the requests simultaneously (to be explained in Section 4.5).

When a physical page is allocated to serve the write request, one mapping slot (BO, PBN) is formed. The corresponding subtable and the data are written to the OOB area and data area, respectively. A page table index is stored in RAM to keep track of the mapping information. For a rewrite (update) operation, the out-of-date mapping slot needs to be read out from the OOB of the page pointed to by the pointers in RAM. The corresponding mapping slot will be updated and then written to the OOB of the new page. The page table index in RAM will also direct to the new physical page. If a free page can always be guaranteed in the buffer block, the time to execute a write request is constant: T_{rdoob}+T_{wrpg} (one OOB read and one page write). The best case response time is also T_{rdoob}+T_{wrpg}. In the worst case that the partial garbage collection operation is scheduled, the worst case response time is T_{er}+T_{rdoob}+T_{wrpg}.

A read request issued from file system is represented by a logical page number (LPN), e.g., *read (36)*. The LPN is translated to a LBN and a BO. The corresponding LBN will be first searched in the block mapping table in RAM. Then the page mapping subtable for the requested BO can be obtained using the page table index in RAM. From the subtable, we can get the physical page which stores the requested data. Since no space is required in serving the read request, no partial garbage collection is invoked. Therefore, the best case response time and the worst case response time of a read request are the same T_{rdoob}+T_{rdpg}.

4.4 Distributed Garbage Collection

The garbage collection in RFTL is invoked once a primary block is full and a write request is issued to this primary block. Given a block with π pages, the garbage collection can be partitioned into k periods (steps) if all the π pages are valid:

$$k = \lceil \{\pi \times (T_{rdpg} + T_{wrpg} + 2T_{rdoob}) + T_{er}\}/T_{er} \rceil \quad (1)$$

In one period p, the write request will firstly be serviced, and the execution time is e_w, where e_w=T_{wrpg}+T_{rdoob}. After the request be serviced, the time left in this period is t, where $t \geq T_e$. In the time t, the garbage collection operations (valid-page copy or block erase) will be performed. For valid-page copy operations, suppose the maximum pages can be copied in this period is β, then:

$$\beta = \lfloor t/(T_{rdpg} + T_{wrpg} + 2T_{rdoob}) \rfloor \quad (2)$$

Figure 5 gives an example of the garbage collection process in RFTL. We assume β=4 and k=3, which means four valid-page copies can be finished in one period p and three periods are needed in the worst case. In Figure 5 (a), the primary block is full and garbage collection is triggered. Write request $w0$ is serviced in the first page of buffer block, meanwhile, four valid pages in the primary block are copied to the replacement block after *copy*0 as shown in Figure 5 (b). After $w1$ is serviced, all the valid pages in the primary block are copied into the replacement block by *copy*1 operation. The primary block is erased after the write request $w2$ is serviced.

Exchange Operation After the primary block is reclaimed, an exchange operation is performed to change the position of the primary block and the replacement block as shown in Figure 5 (d). The new primary block will serve the coming requests if free space is available (i.e.,$w3$ and $w4$). After the primary block is full, the coming requests are written to the buffer block (i.e.,$w5$ and $w6$). When the buffer block has only k (i.e., k=3) free pages left, the partial garbage collection of primary block is triggered again. The replacement block will store the valid pages from both the primary block and the buffer block. The partial garbage collection is interleaved with pending requests served in the buffer block (i.e.,$w7$, $w8$ and $w9$). After the buffer block is full, the primary block is free as shown in Figure 5 (e).

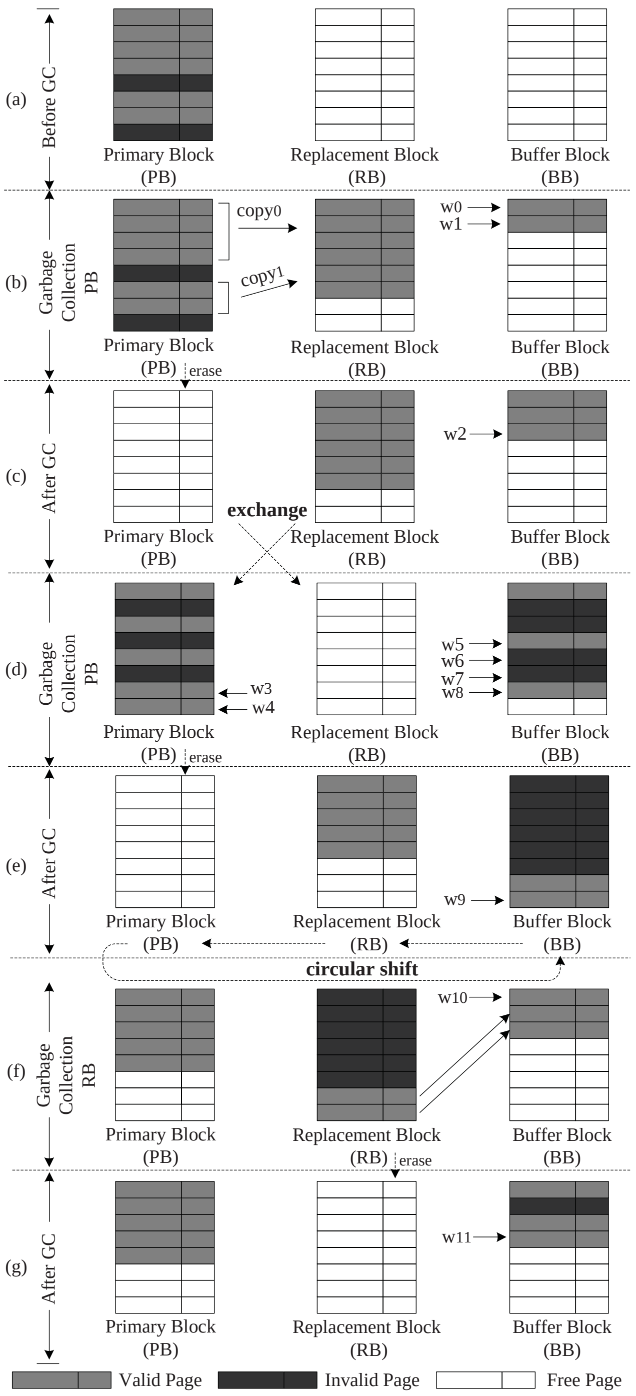

Figure 5. Garbage collection in RFTL.

Circular Shift Operation After the buffer block is full, a circular shift operation is taken to change the position of three blocks. The free primary block will be reallocated as a buffer block, and the original buffer block is transfered to a new replacement block. The original replacement block will serve as the new primary block as shown in Figure 5 (f). Partial garbage collection for replacement block is triggered. Since the replacement block has k valid pages, the garbage collection can be split into j partial steps:

$$j = \lceil k \times (T_{rdpg} + T_{wrpg} + 2T_{rdoob})/T_{er} + 1 \rceil \quad (3)$$

Figure 5 (g) shows an example of reclaiming replacement block when j equals to two. The replacement block becomes free after two write requests $w10$ and $w11$ are served in the buffer block. The primary block can serve the requests again if free pages are included. A new garbage collection will be invoked if the new primary block is full and a new request wants to access this block.

In RFTL, garbage collection of one physical block is partitioned into multiple independent steps, and each step is triggered by one request. If the requests arrive and want to access the same logical block, the partial steps are performed consecutively within the physical blocks mapped to the same logical block. Otherwise, if the requests want to access different logical blocks, the garbage collection operations are distributed to different logical blocks correspondingly. In Figure 5, the garbage collection of the primary block or replacement block is triggered and finished by consecutive requests which are mapped to the same logical block.

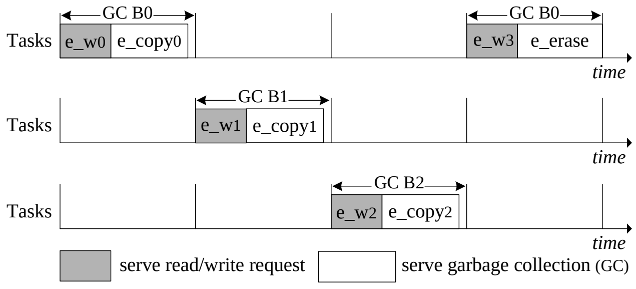

Figure 6. Distributed garbage collection.

Figure 6 gives an example of garbage collection which are distributed to different logical blocks by the requests mapped to different logical blocks. We suppose four requests $w0$, $w1$, $w2$, $w3$ arrive sequentially. Write requests $w0$ and $w3$ are mapped to primary block $B0$, and $w1$ is mapped to primary block $B1$ while $w3$ is mapped to primary block $B2$. In the first period, garbage collection of $B0$ is performed in which valid-page copy $copy0$ is executed after the schedule of request $w0$. In the second period, primary block $B1$ is reclaimed since the request $w1$ is mapped to it, and block erase operation $erase$ is executed. When it comes to period 4, primary block $B0$ is reclaimed again since the garbage collection is not finished in the first period. Two benefits can be achieved by distributed partial garbage collection. First, the long garbage collection latency can be fundamentally hidden, such that the worst case response time of requests can be reduced to $L(p)$, where $L(p)$=$max\{T_{rdpg}$+T_{rdoob}, T_{er}+T_{wrpg}+$T_{rdoob}\}$. Second, the garbage collection overhead can be reduced since the valid page numbers in one block may decrease when the garbage collection is distributed. In other words, the changing from reclaiming one block to a new block postpones the garbage collection of the old block. The postponed reclamation of the old block may reduce the valid page numbers within it, since a later rewrite operation may make the original valid page become invalid. The average system response time is consequently reduced due to the decreased garbage collection overhead.

4.5 WCET Analysis

Based on the distributed garbage collection policy, we can obtain the worst case response time for requests in RFTL is $L(p)$ if enough free space can be guaranteed. In order to verify that the block management in RFTL can provide enough space for all requests, we present the worst case analysis and give one theorem. The theorem gives the sufficient condition that a write request can be deterministically serviced.

Theorem 4.1. *The sufficient condition that deterministic service can be provided for each request is that: at least one free block and k free pages should be reserved when the distributed partial garbage collection is triggered.*

Proof. In the worst case, all pages in the victim block are valid pages. If the space reserved is less than one free block, at least one of the valid pages in the victim block has no place to be stored. If less than k free pages is provided, at least one pending write will be blocked. □

Based on Theorem 4.1, we can get two lemmas for our scheme. The first lemma shows the sufficient condition that deterministic service can be guaranteed when doing partial garbage collection for one block with k valid pages. The second lemma presents the minimum number of blocks that are needed to guarantee the deterministic service.

Lemma 4.1. *Given a victim block with k valid pages, the sufficient condition that partial garbage collection can work is that: at least k+j free pages should be reserved.*

Proof. In the worst case scenario, enough free space should be guaranteed to stored the k valid pages and the j pending writes which are interleaved with the partial garbage collection. Therefore, if less than $k+j$ space are provided, at least one valid page or one pending write will be blocked. □

In RFTL, the partial garbage collection of the replacement block is triggered after the circular shift operation. In the worst case, k valid pages are needed to be copied into buffer block. Since the buffer block can provide at least $2k+j$ free pages, the partial garbage collection can be guaranteed according to Lemma 4.1.

Lemma 4.2. *When distributed partial garbage collection is applied in block-level mapping schemes, the minimum number of blocks to guarantee deterministic service is 3.*

Proof. If one logical block is mapped to one physical block, no free space is provided to do partial garbage collection. It violates the sufficient condition in Theorem 4.1. If one logical block is mapped to two physical blocks, only one free block is provided. It also violates the sufficient condition in Theorem 4.1. Therefore, in order to provide deterministic service with distributed partial garbage collection, at least three blocks are needed □

In RFTL, we adopt a block-level mapping approach that one logical block is mapped to three physical blocks. Lemma 4.2 provides the guideline on how to design a deterministic FTL scheme with block-level mapping approach.

5 Evaluation

5.1 Experimental Setup

Table 3. Experimental setup.

Hardware	CPU	Intel Dual Core 2GHz
	Disk Space	200GB
	RAM	2GB
Simulation Environment	OS Kernel	Linux 2.6.17
	Flash Simulator	NAND flash simulator
	Flash Size	128/256/512MB

To evaluate the effectiveness of the proposed scheme, we developed a trace-driven NAND flash simulator under Linux kernel 2.6.17 and implemented three FTL schemes: GFTL [9], NFTL [5] and RFTL. NFTL scheme is a general purpose block-level FTL scheme. GFTL is a representative deterministic FTL scheme. Therefore, we compare our scheme with NFTL and GFTL. Table 3 summarizes our experimental setup. Three NAND flash memory chips with capacity 128MB, 256MB and 512MB separately are simulated. The framework of our simulation platform, as shown in Figure 7, consists of two modules: a NAND flash simulator providing basic read, write and erase capabilities; and a desired flash translation layer management scheme that can be executed on top of the NAND flash simulator. The traces along with various flash parameters, such as block size and page size, page read time and page write time etc., are fed into our simulation framework. The parameters in our simulation are based on the flash memory data sheet values shown in (Table 1).

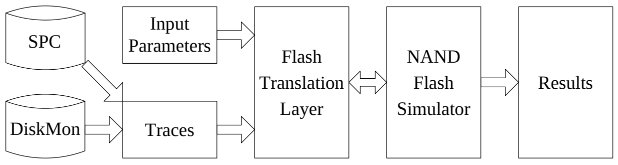

Figure 7. The framework of simulation platform.

We use the following benchmarks from both the real-world and the synthetic traces to study the system performance for different FTL schemes. *Multimedia* is a real-world trace we obtained from a PC with Windows XP on NTFS file system downloading and playing multimedia files (e.g., Movie, MP3). It consists of 1,633,269 write requests and 1,002,748 read requests. *Financial* is a well known write-dominant I/O trace obtained from an OLTP application running at a financial institution [2]. It consists of 4,099,354 write requests and 1,235,633 read requests. In order to perform a rigorous evaluation of different schemes, each read/write request in the traces is simulated with a periodicity of $L(p)$ without idle period involved.

5.2 Results and Discussion

In this section, we present the simulation results of the proposed RFTL scheme, GFTL scheme and NFTL scheme in terms of real time and average performance as well as the space overhead (RAM cost and flash memory cost).

Table 4 presents the performance of RFTL scheme for the two traces based on varying flash utilization and number of pages per block π. The first three columns under R_{best}, R_{avg} and R_{worst} denote the best case, the average and the worst case response time for read requests, respectively. The next three columns W_{best}, W_{avg} and W_{worst} represent the best case, the average and the worst case response time for write requests. The average response time for all requests, the total number of valid-page copy operations and the total number of erase operations are also measured which are denoted by T_{avg}, Σ_{cp} and Σ_{er} respectively.

Based on Table 4, we can observe that, the worst case response time for a read request is $50\mu s$, which is equal to $T_{rdoob}+T_{rdpg}$. For a write request, the worst case response time is $2325\mu s$, which is equal to $T_{er}+T_{rdoob}+T_{wrpg}$. The worst case response time for read request and write request is independent of the flash utilization and the flash size. It presents no variation when the flash utilization and the page size per block (π) vary. This observation shows that our scheme can provide guaranteed service for different flash specifications and different traces.

The average response time for read requests is close to the best case response time, and the average response time for write requests is close to the worst case response time. This is because that, little valid-page copy operations or

Table 4. RFTL performance.

Benchmarks	%	π	R_{best} (μs)	R_{avg} (μs)	R_{worst} (μs)	W_{best} (μs)	W_{avg} (μs)	W_{worst} (μs)	T_{avg} (μs)	Σ_{cp}	Σ_{er}
Multimedia	50	32	50	50	50	325	400	2,325	335	137,630	69,142
	50	64	50	50	50	325	359	2,325	298	66,508	33,296
	50	128	50	50	50	325	339	2,325	280	32,414	16,208
	100	32	50	50	50	325	419	2,325	341	270,903	205,297
	100	64	50	50	50	325	375	2,325	303	131,281	99,021
	100	128	50	50	50	325	353	2,325	285	64,295	48,367
Financial	50	32	50	50	50	325	389	2,325	274	31,822	26,943
	50	64	50	50	50	325	354	2,325	248	15,445	13,049
	50	128	50	50	50	325	338	2,325	236	7,488	6,285
	100	32	50	50	50	325	390	2,325	271	68,687	79,720
	100	64	50	50	50	325	355	2,325	245	33,409	38,714
	100	128	50	50	50	325	337	2,325	232	16,381	18,812

block erase operations are involved in one period p. This verifies that the distributed garbage collection can provide enough space to serve the sustained coming requests. The average response time for each trace is decreased while the number of valid-page copy and block erase operations are reduced with the increase of the flash size (e.g., π increases from 32 to 128). This is based on the fact that more free flash space will lead to less garbage collection when servicing the same amount of requests. Moreover, the valid-page copy and block erase operation are increased when the flash utilization is increased for a fixed flash size. This is due to the fact that free space becomes less when the flash continually serves the write request. More garbage collection will be invoked to reclaim the obsolete pages, which increases the average system response time.

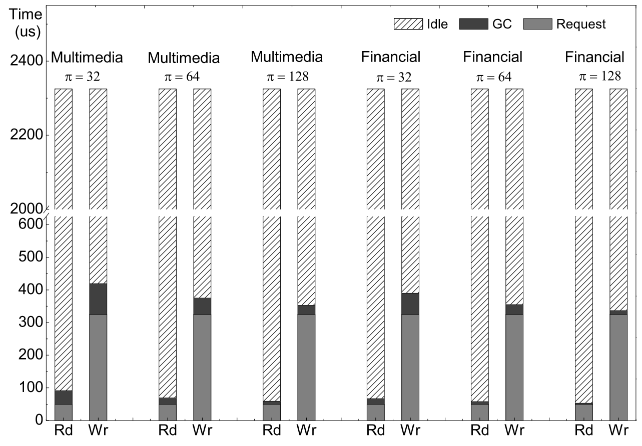

Figure 8. Average time distribution per period.

Figure 8 shows the distribution of request service time and the garbage collection (GC) overhead in one period p. The total length of a bar represents the upper bound of response time which is $L(p)$ as mentioned in Table 2. The "Request" bar denotes the execution time of request, and the "GC" bar represents the average time cost in garbage collection which includes a series of valid-page copy and block erase operations. The time left is the idle time. Since the total length of the bar is calculated under the worst case scenario, from the results we can see that, a large amount of time is idle in one period. The idle time increases when the value of π is increased. This is because more space are provided leading to less garbage collection overhead. Note that in case of a read request, the idle time left is much longer than that of a write request for both two traces. This is because the time cost to execute a page read is less than that of a page write. From the figure, we can also observe that the idle time for trace $Financial$ is longer than that of trace $Multimedia$. This is due to the fact that, trace $Financial$ follows higher temporal locality and more update operations occur resulting in less valid-page copy operations in garbage collection.

Table 5. Performance for RFTL, GFTL[9] and NFTL[5].

Traces	Metrics	FTL Schemes		
		RFTL	GFTL	NFTL
Multimedia	T_{worst} (μs)	2,325	3,650	4,335
	T_{avg} (μs)	303	525	321
	Σ_{cp}	1.31e5	5.38e5	3.95e5
	Σ_{er}	0.99e5	1.29e5	0.48e5
	Σ_{oob}	0.05e8	0.29e8	1.37e8
	$L(p)$ (μs)	2,325	3,650	4,335
Financial	T_{worst} (μs)	2,325	3,650	4,557
	T_{avg} (μs)	245	2,997	522
	Σ_{cp}	0.03e6	7.65e7	0.38e7
	Σ_{er}	0.38e5	6.60e5	1.23e5
	Σ_{oob}	0.02e8	0.40e8	2.86e8
	$L(p)$ (μs)	2,325	3,650	4,557

Table 5 compares the system performance of RFTL, GFTL and NFTL under the same flash size and space utilization ratio. Both RFTL and GFTL show great improvement in the worst case response time compared with NFTL scheme which is a general purpose FTL. In NFTL scheme, two block merge operations are involved and the blocked time of each request in the worst case is at least $2{*}T_{er}$. In GFTL scheme and RFTL scheme, one victim block reclamation is needed and the garbage collection is partitioned into multiple small steps. Therefore, they present lower worst case response time compared with NFTL. Note that RFTL achieves a 36.30% improvement in the worst case response time compared with that of GFTL, which means RFTL can accept requests at a higher arrival rate while providing read/write service guarantees. This is based on the fact that GFTL requires to search all the OOB area of one block in order to read the valid page. But in RFTL, the address mapping information can be obtained by reading one OOB area.

RFTL scheme shows better average response time compared with NFTL scheme while GFTL presents the longest average response time. In order to provide enough space to serve the real time requests, GFTL scheme invokes the garbage collection once a block becomes full. The reclamation of one block is performed in a concentrated manner, which incurs many unnecessary valid page copy and block erase overhead. As shown in Table 5, these extra overhead increases the average response time significantly compared with NFTL and RFTL. In RFTL scheme, the partial garbage collection is distributed to each logical block in an on-demand fashion. The valid page copy and block erase are performed only when needed. The delayed reclamation reduces the valid page copy number and block erase number. Therefore, RFTL achieves a 67.06% improvement in average response time compared with GFTL.

5.3 Overhead

In order to provide deterministic service, both GFTL and RFTL introduce extra flash space to serve as the write buffer for partial garbage collection. The number of buffer blocks required for GFTL is the same as that of data blocks, while RFTL needs double of data blocks to serve as replacement blocks and buffer blocks. Although RFTL has more extra overhead on flash space, it shows great reduction in much more valuable RAM space. Given a large block based 128MB NAND flash with 64 pages per block, RFTL requires 16KB (16B*1024) RAM space to store the block mapping table and page mapping table index. For GFTL scheme, the RAM cost is 195KB which consists of three parts: the block level mapping table for data blocks (3KB), the page mapping table for buffer blocks (64KB) and one block buffer in RAM (128KB). RFTL shows a 91.79% reduction in RAM cost compared with GFTL. In terms of reliability, RFTL can startup safely by scanning the flash media in which the mapping table is stored permanently.

6 Conclusion and Future Work

In this paper, we proposed a real time flash translation layer (called RFTL) for NAND flash memory storage systems which can provide real time service guarantees by hiding the long garbage collection latency. A novel hybrid-level address mapping approach and a distributed garbage collection policy are introduced. Experimental results show that our scheme can achieve a 36.30% improvement in the worst case response time upper bound for requests compared with GFTL. Moreover, we achieve a 67.06% reduction in average system response time and a 91.79% reduction in RAM cost compared with GFTL. In the future, how to further reduce the extra flash space overhead is an interesting problem. Moreover, how to fully utilize the free pages in buffer blocks to further improve the average response time for some specific applications is also one direction.

References

[1] Intel Corporation. Understanding the flash translation layer (FTL) specification. *http://developer.intel.com*.

[2] OLTP trace from umass trace repository. *http://traces.cs.umass.edu/index.php/storage/storage*.

[3] Airlines electronic engineering committee (AEEC). *ARINC Specification*, 651, 1991.

[4] A. Ban. Flash file system. *US patent 5,404,485*, 1995.

[5] A. Ban. Flash-memory translation layer for NAND flash (NFTL). *M-systems*, 1998.

[6] L.-P. Chang and T.-W. Kuo. An adaptive striping architecture for flash memory storage systems of embedded systems. In *RTAS '02*, pages 187–196, 2002.

[7] L.-P. Chang, T.-W. Kuo, and S.-W. Lo. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *TECS*, 3(4):837–863, 2004.

[8] H. Cho, D. Shin, and Y. I. Eom. KAST: K-associative sector translation for NAND flash memory in real-time systems. In *DATE'09*, pages 393 –398, 2009.

[9] S. Choudhuri and T. Givargis. Deterministic service guarantees for NAND flash using partial block cleaning. In *CODES+ISSS '08*, pages 19–24, 2008.

[10] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *ASPLOS '09*, pages 229–240, 2009.

[11] J.-C. Kim, D. Lee, C.-G. Lee, K. Kim, and E. Y. Ha. Real-time program execution on NAND flash memory for portable media players. In *RTSS '08*, pages 244–255, 2008.

[12] S. Kim, C. Park, and S. Ha. Architecture exploration of NAND flash-based multimedia card. In *DATE '08*, pages 218–233, 2008.

[13] Z. Qin, Y. Wang, D. Liu, and Z. Shao. A two-level caching mechanism for demand-based page-level address mapping in NAND flash memory storage systems. In *RTAS '11*, pages 179–188, 2011.

[14] Z. Qin, Y. Wang, D. Liu, Z. Shao, and Y. Guan. MNFTL: An efficient flash translation layer for MLC NAND flash memory storage systems. In *DAC '11*, pages 17–22, 2011.

[15] Y. Wang, D. Liu, M. Wang, Z. Qin, Z. Shao, and Y. Guan. RNFTL: a reuse-aware NAND flash translation layer for flash memory. In *LCTES '10*, pages 163–172, 2010.

[16] C.-H. Wu and T.-W. Kuo. An adaptive two-level management for the flash translation layer in embedded systems. In *ICCAD '06*, pages 601–606, 2006.