Software Engineering Group
Department of Computer Science
Nanjing University
http://seg.nju.edu.cn

Technical Report No. NJU-SEG-2015-IC-002

2015-IC-002

Optimizing Deterministic Garbage Collection in NAND Flash Storage Systems

Qi Zhang, Xuandong Li, Linzhang Wang, Tian Zhang, Yi Wang, Zili Shao

Real-Time and Embedded Technology and Applications Symposium 2015

Optimizing Deterministic Garbage Collection in NAND Flash Storage Systems

Qi Zhang Xuandong Li Linzhang Wang Tian Zhang
State Key Laboratory for Novel Software Technology
Nanjing University

Yi Wang Zili Shao
Department of Computing
The Hong Kong Polytechnic University

***Abstract*—NAND flash has been widely adopted as storage devices in real-time embedded systems. However, garbage collection is needed to reclaim space and introduces a lot of time overhead. As *the worst system latency* is determined by the worst-case execution time of garbage collection in NAND flash, it is important to optimize garbage collection so as to give a deterministic worst system latency. On the other hand, since the garbage collection does not happen very often, optimizing garbage collection should not bring too much overhead to *the average system latency*.**

This paper presents for the first time a worst-case and average-case joint optimization scheme for garbage collection in NAND flash. With our scheme, garbage collection can be postponed to the latest stage so improves the average system latency. By combining partial garbage collection and over-provisioning, our scheme can guarantee that one free block is enough to hold all pages from both write requests and valid-page copies. The experiments have been conducted on a real embedded platform and the results show that our technique can improve both worst-case and average-case system latency compared with the previous works.

I. Introduction

NAND flash has been widely adopted as storage devices in real-time embedded systems due to its small size, low power consumption, high density and good shock resistance. However, NAND flash storage systems may adversely impact the worst-case system performance in real-time systems. In particular, garbage collection is needed to reclaim space in NAND flash. Compared with read/write operations, a garbage collection operation takes much longer time in which we need to copy valid pages (page is the basic unit for read and write operations) from a victim block to a free block (block is the basic unit for erase operations, and a block contains many pages) and then erase the victim block. Therefore, it is important to optimize garbage collection, as it determines the worst system latency in NAND flash. On the other hand, garbage collection does not happen very often, and it should be triggered only when a NAND flash system does not have enough free space. Thus, optimizing garbage collection should not bring too much overhead to the average system performance in NAND flash. This paper addresses worst-case and average-case optimization for garbage collection in NAND flash.

Deterministic garbage collection in NAND flash has been studied in the previous works. The problem that garbage collection in NAND flash causes deadline misses in real-time embedded systems is first studied in [1], in which a real-time garbage collection mechanism is proposed to reduce the worst-case execution time of garbage collection. In their mechanism, "over-provisioning" is applied to map a small logical space (the capacity that users are able to see and use) to a big physical space (the capacity that a NAND flash really has). By doing this, in a block, there must exist a certain amount of invalid pages, because multiple physical pages will be mapped to one logical page. In such a way, the number of valid-page copies can be guaranteed to be less than a fixed value based on an over-provisioning ratio; therefore, the worst-case execution time of garbage collection can be reduced. However, by performing both valid-page copies and block erase in one garbage collection operation, the execution time of garbage collection is still very long.

In [2], partial garbage collection is proposed to divide a garbage collection operation into several small steps and the time of each step is not longer than that of one erase operation. By interleaving each partial garbage collection step with the service of a read/write request, the worst system latency is deterministic, as the longest operation time in NAND flash is to erase a block in which it cannot respond to I/O requests. However, with this scheme, once a block is full, it will be put into a garbage collection queue, and partial garbage collection is performed if the queue is not empty. As garbage collection is triggered very early, a lot of unnecessary erase operations are introduced. Thus, the average system performance is not good. To solve this problem, a distributed partial garbage collection scheme is proposed in [3], in which the garbage collection queue is distributed to each logical block, and garbage collection is managed by each logical block in a distributed manner. The distributed partial garbage collection can effectively postpone garbage collection so as to improve the average system performance. However, its space overhead is too big, and it does not completely solve the average performance degradation from the early garbage collections.

In this paper, we study the worst-case and average-case joint optimization for garbage collection in NAND flash (called WAO-GC). In our scheme, garbage collection can be postponed to the latest stage with only a few free blocks left. As valid pages may become invalid later in NAND flash, postponing garbage collection can avoid unnecessary valid-page copies and erase operations, thereby optimizing the average system latency. By combining partial garbage collection and over-provisioning, our scheme can achieve *deterministic optimizing worst system latency*, while guaranteeing that one free block is big enough to hold all pages from both write requests and valid-page copies when a victim block is reclaimed via the partial garbage collection. We conduct experiments using both real applications on the evaluation board and standard benchmarks which are commonly used in the research community.

We implemented our scheme as a block device driver, and evaluate it with a set of benchmarks and real applications on a real embedded platform. FTLs with real-time garbage collection schemes including Real-time GC [1], GFTL [2], and RFTL [3] are implemented in evaluation. The experimental results show that our scheme can effectively improve the worst-case and average-case system latency compared with the above FTLs.

The remainder of this paper is organized as follows. Section II introduces the background. The worst-case and average-case system latency are discussed in Section III. Our garbage collection scheme is presented in Section IV. Section V presents the experimental results. The related work is introduced in VI. The conclusion and future work are presented in Section VII.

II. Background

A. NAND Flash Storage System and FTL

In recent years, many emerging memory techniques [4], [5], [6], [7], [8], [9] have been proposed and NAND flash memory has been widely adopted in many kinds of embedded systems. A typical NAND flash storage system usually includes two layers, the flash translation layer (FTL) and the memory technology device (MTD) layer. The MTD layer provides primitive functions such as read, write, and erase that directly operate on a flash memory system. The FTL emulates a flash memory system as a block device so that the file systems can access the flash memory transparently. The FTL usually provides three components: address translator, garbage collector, and wear-leveler. In an FTL, the address translator translates addresses between logical addresses and physical addresses based on a mapping table; the garbage collector reclaims space by erasing obsolete blocks in which there exist invalid data; the wear-leveler is an optional component that distributes write or erase operations evenly across all blocks, so the lifetime of a flash memory system can be improved. Many FTL schemes [10], [11], [12], [13], [14] have been designed to improve the NAND flash storage system.

B. Garbage Collection

One of the major functions in an FTL is to perform garbage collection to reclaim space. In a garbage collection operation, for a victim block, its valid pages are copied (read out and written to) to a free block and the victim block is erased for reuse.

Figure 1(a) shows an example of a garbage collection operation. As the victim block contains 4 valid pages, in the garbage collection, we need to copy them to a free block, in which a page copy consists of one page read and one page write, and then the victim block is erased. Using the parameters in Table II, it takes about $4 \times (25 + 200) + 1500 = 2400\mu s$ to finish this garbage collection operation by ignoring other negligible time overhead and only considering the major time from read (25 μs per page), write (200 μs per page) and erase (1500 μs per block) operations.

To shorten the latency introduced by garbage collection, the partial garbage collection mechanism is proposed in GFTL[2]. With this mechanism, for the same garbage collection operation in Figure 1(a), the operation is divided into several small steps, and each step interleaves with the service of a read or write request. The time of each step is not longer than that of one erase operation which is the longest operation time in which it cannot respond to I/O requests. By doing this, as shown in Figure 1(b), I/O requests can be responded in no more than 1500 μs.

III. Worst-Case and Average-Case System Latency

In this section, we first define the latency of a data request, and then formulate the worst-case and average-case system latency.

In a NAND flash storage system, I/O requests are served by an FTL that fetch data from the NAND flash and return them to the file system. In the WCET (Worst-Case Execution Time) analysis of a real-time task, we need to consider the storage system latency of a request when the task sends requests to NAND flash storage systems. Given a read or write request, we define **the storage system latency of the request** as the time period from the point when the request is issued from the file system to the point when the request has been served and the result has been returned from the FTL. To simplify the definition of data request, we define each data request in FTL has only one page read/write operation in this paper. The storage system latency of a request is mainly decided by the read/write page operation and the garbage collection, whose execution time are several orders of magnitude larger than these of other operations of the FTL. Thus, we consider the execution time of read/write page operation and block erase operation when discussing the system latency of a request in this paper.

The latency of a request may vary a lot. For example, using the parameters in Table II, if a read request arrives when the system is idle, its system latency is 25 μs. However, if it arrives just after the garbage collection starts as shown in Figure 1(c), with the conventional garbage collection or the partial garbage collection, its latency will be $2400 + 25 = 2425$ μs or $225 + 25 = 250$ μs. For a NAND flash storage system, we define **the worst system latency** and **the average system latency** as the longest latency time and the average latency for all read and write requests, respectively.

As shown in Figure 1(c), garbage collection plays a very important role influences the latency of a request. Let t_{er} be the block erase time, t_{wr} be the page write time, and t_{rd} be the page read time. *The worst system latency* can be minimized by the partial garbage collection, and its optimal value can be represented as follows:

$$Optimal\_WCET = t_{er} + \max\{t_{wr}, t_{rd}\} = t_{er} + t_{wr} \quad (1)$$

The average system latency is influenced by the garbage collection operations that happened during read and write requests, and can be calculated as follows:

$$\frac{N_{rd} \times t_{rd} + N_{wr} \times t_{wr} + \sum_{i=1 \to N_{GC}} t_{GC_i}}{N_{rd} + N_{wr}} \quad (2)$$

In the above equation, N_{rd} and N_{wr} are the numbers of read and write requests, respectively, N_{GC} is the number of the garbage collection operations involved, and t_{GC_i} is the execution time of the ith garbage collection. The optimal average system latency is achieved when there is no any

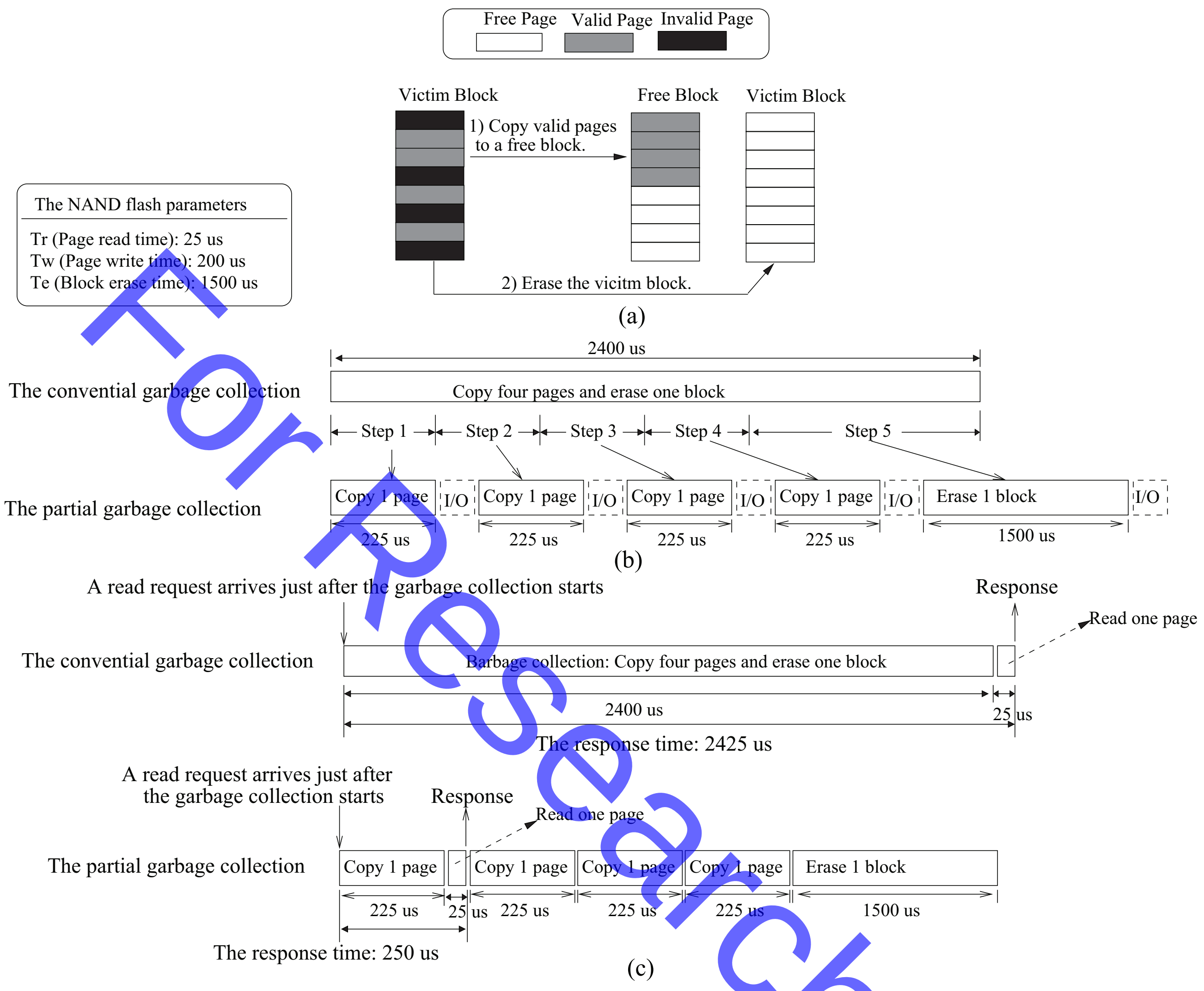

Free Page
Valid Page
Invalid Page
Victim Block
Free Block
Victim Block
1) Copy valid pages to a free block.
2) Erase the vicitm block.
The NAND flash parameters
Tr (Page read time): 25 us
Tw (Page write time): 200 us
Te (Block erase time): 1500 us
(a)
2400 us
The convential garbage collection
Copy four pages and erase one block
Step 1
Step 2
Step 3
Step 4
Step 5
The partial garbage collection
Copy 1 page
I/O
Erase 1 block
225 us
1500 us
(b)
A read request arrives just after the garbage collection starts
Response
Read one page
The convential garbage collection
Garbage collection: Copy four pages and erase one block
2400 us
25 us
The response time: 2425 us
A read request arrives just after the garbage collection starts
Response
Read one page
The partial garbage collection
Copy 1 page
Erase 1 block
225 us
25 us
1500 us
The response time: 250 us
(c)

Fig. 1. (a) Illustration of a garbage collection operation. (b) Illustration of the partial garbage collection, by which the garbage collection operation in (a) is divided into small steps and each step is interleaving with the service of a read/write request. (c) The different response times of a read request with the conventional garbage collection in (a) and the partial garbage collection in (b), respectively (using the NAND flash parameters in Table II).

garbage collection operation during all write and read requests, and it can be represented as follows:

$$Optimal\_Average = \frac{N_{rd} \times t_{rd} + N_{wr} \times t_{wr}}{N_{rd} + N_{wr}} \tag{3}$$

IV. Optimizing Deterministic Garbage Collection in NAND Flash Storage Systems

In this section, we present our optimizing deterministic garbage collection scheme. We first introduce the basic idea in Section IV-A and then present our scheme in Section IV-B. The implementation in FTLs is introduced in Section IV-C. Finally, we analyze the performance in Section IV-D.

A. Overview

To archive deterministic *the worst system latency*, We apply the partial garbage collection scheme by which each read or write request can only be influenced by at most one partial garbage collection step, so t_{GC_i} is not bigger than t_{er} in Equation 2. In order to optimize *the average system latency*, as shown in Equation 2, we need to reduce the number of the garbage collection operations happening during the process of serving read and write requests. Furthermore, we should reduce the execution time of the garbage collection involving in read or write requests. In our scheme, garbage collection can be postponed to the latest stage when there is about to out of space. As valid pages may become invalid later in NAND flash, postponing garbage collection can avoid unnecessary valid-page copies and erase operations so as to reduce the garbage collection overhead. With the above two strategies, the average system latency can be improved and at the same time we can give a deterministic worst system latency.

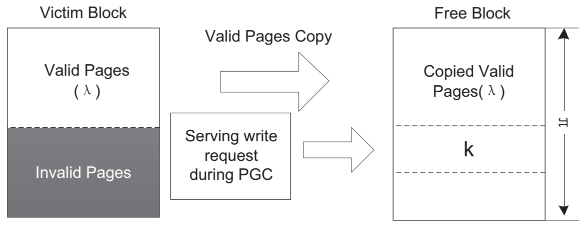


Fig. 2. Illustration of the basic idea of our garbage collection scheme.

Figure 2 shows the basic idea of our garbage collection scheme. First, when a victim block is selected, our scheme can guarantee that it contains at most λ valid pages. Second, the victim block will be reclaimed through the partial garbage collection. Suppose coping λ valid pages and erase a victim block requires k partial garbage collection steps and each step will be executed after each write data request, the entire garbage collection process will be completed after k write requests so we need to store these k pages. Let π be the page number in one block. As long as

$$k + \lambda \leq \pi$$

is satisfied, we can use one free block to hold both λ valid pages from the victim block, and k pages from write requests during reclaiming the victim block. After k partial garbage collection steps, the victim block becomes free, so we can always provide one free block for garbage collection.

B. Garbage Collection Scheme

In our scheme, we combine the partial garbage collection with the over-provisioning. Partial garbage collection can be invoked in the system idle time to avoid being out of free space quickly. However, the system idle time depends on the workload and requires operating system information. In this paper, we consider the extreme worst case and want to postpone the garbage collection as late as possible. As shown above, to achieve the joint worst-case and average-case optimization, the key is to satisfy $k + \lambda \leq \pi$. With the partial garbage collection, we first obtain the maximum valid pages allowed in a victim block (the maximum value of λ). k is the number of partial steps needed to reclaim a victim block with λ valid pages, so we have the following equations:

$$\alpha = \lfloor \frac{t_{er}}{t_{rd} + t_{wr}} \rfloor \tag{4}$$

$$k = \lceil \frac{\lambda}{\alpha} \rceil + 1 \tag{5}$$

Equation 4 shows how many page copies (represented by α) we can finished in one partial step. With the partial garbage collection, the longest atom time of a partial step is the time to erase a block. Therefore, α can be obtained by dividing t_{er} (the time to erase one block) with the summation of t_{rd} and t_{wr} (the time to read and write one page for one page copy). Then Equation 5 can be obtained that represents how many partial steps are needed to reclaim one block with λ valid pages (λ/α valid-page copies and one erase operation).

As a result, based on $k + \lambda \leq \pi$, we have:

$$\lceil \frac{\lambda}{\alpha} \rceil + 1 + \lambda \leq \pi \tag{6}$$

In a NAND flash storage system, α is fixed. Thus, based on Equation 6, we can obtain the maximum value of λ, i.e., the maximum valid pages allowed in a victim block. By making use of over-provisioning strategy, our scheme can guarantee the number of valid pages in victim block is less than the maximum allowed value of λ (i.e., the upper bound of λ, represented by $\mathcal{U}(\lambda)$).

In our over-provisioning strategy, a big physical space in a NAND flash storage system is mapped to a small logical space in the file system. As shown in Figure 3, there is a 3 GB logical address space mapping to the entire 4 GB physical address space. When the flash space is almost full, there are at least a number of invalid pages in the flash (1GB in the Figure 3). Suppose we use σ to present the ratio between the logical and physical space, $\sigma = \Lambda/N$, where Λ represents the number of logical pages from the file system, while N denotes the total number of data pages in the physical space. When the flash is almost full (i.e., only one free block left), by adopting greedy strategy to select the victim block contains the least valid pages, the upper bound of the λ can be calculated using the following equation.

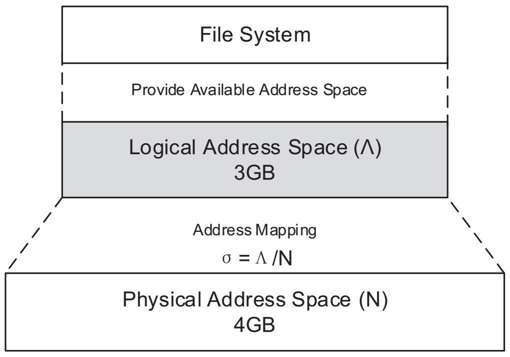

File System
Provide Available Address Space
Logical Address Space (Λ)
3GB
Address Mapping
σ = Λ /N
Physical Address Space (N)
4GB

Fig. 3. The over-provisioning strategy.

$$\mathcal{U}(\lambda) = \lceil \sigma \times \pi \rceil \tag{7}$$

Here we present a simple proof. Suppose there are M data blocks and each block has π pages, according to the definition, the total number of physical data pages is $N = \pi \times M$ and the total number of valid pages is $\sigma \times \pi \times M$. By selecting the victim block which has the least number of valid pages (denoted as λ_{min}), there should be at least λ_{min} valid pages in each of the rest data blocks. Suppose the λ_{min} is more than $\mathcal{U}(\lambda)$ (i.e., $\mathcal{U}(\lambda) + 1$), the total number of valid page is $\lceil \sigma \times \pi \rceil \times M + M$. The value is more than the pre-defined value of $\sigma \times \pi \times M$, which causes a contradiction. As a result, the λ_{min} is no more than the $\mathcal{U}(\lambda)$.

The value of $\mathcal{U}(\lambda)$ is independent to the workload and only related to the space configuration (σ) and the flash specification (π). Therefore, our scheme can guarantee the maximum value of λ from over-provisioning strategy. By combining Equations 5 and 6, we can get the relationship between partial garbage collection and space configuration in Equation 8. The upper bound of σ only depends on the constants α and π. Therefore, our scheme can satisfy $k + \lambda \leq \pi$ only when the ratio of logical address space and physical address space is configured lower than the upper bound of $\mathcal{U}(\sigma)$.

$$\mathcal{U}(\sigma) = \frac{(\pi - 1)\alpha}{(\alpha + 1)\pi} \tag{8}$$

Figure 4 shows an example of our garbage collection scheme. Suppose there are 4 data blocks, each of which has 8 data pages ($\pi = 8$), and execution time of read (t_{rd}), write (t_{wr}), and erase operation (t_{er}) are $60\mu s$, $600\mu s$, and $1500\mu s$, respectively. Therefore, $\alpha = \lfloor 1500/(60 + 600) \rfloor = 2$ and the upper bound of σ is $((\pi - 1)\alpha)/((\alpha + 1)\pi) = (7 \times 2)/(3 \times 8) \approx 0.583$, which means the logical address space is at most 58.3% of the physical address space. We defines $\sigma = 0.5$ in this

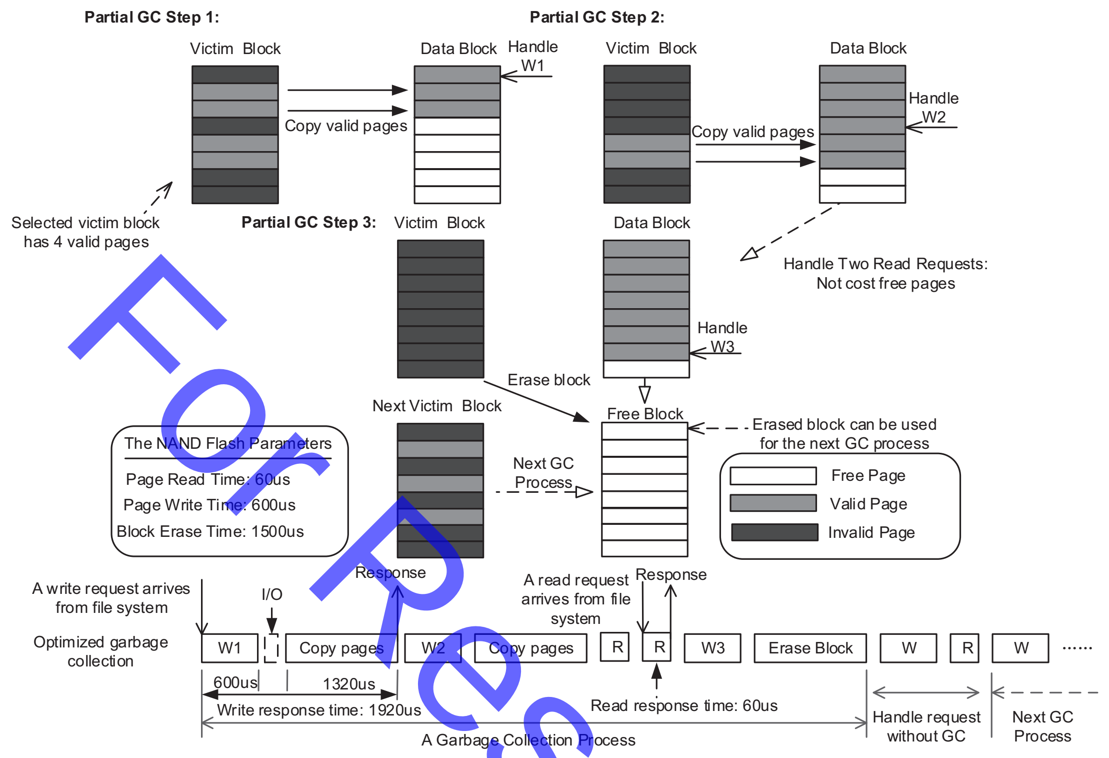


Fig. 4. An example to illustrate our garbage collection process

example, thus, when the flash is full, we can see the total number of valid pages is at most $(4 \times 8) \times 0.5 = 16$. If selecting a victim block by adopting greedy strategy, there is at most 4 valid pages in the block. Therefore, we can generate at most $k = \lambda / \alpha + 1 = 4/2 + 1 = 3$ partial garbage collection steps. As shown in the example, when there is one free block left in the flash and the after handling the write request $W1$ to the first page, our scheme executes partial step 1 to copy first two valid pages. We schedule the partial step 2 as the step 1, and then there are two coming read requests. Since the read requests will not cost free data pages, our scheme will not schedule partial steps to read requests. Finally, the last partial step 3 will erase the victim block to reclaim a new free block. The entire garbage collection process costs $k + \lambda = 3 + 4 = 7$ pages but can reclaim a free block which has 8 data pages. The next data request will not trigger any partial steps until there is only one free block left again and at that time, the last reclaimed block can be used for the next garbage collection process. Therefore, it can satisfy the $k + \lambda \leq \pi$ by using only one free data block and there is the lowest impact to data requests bounding the worst case system latency of write request to the lowest level, which can achieve jointly optimized the worst-case and average-case latency.

C. Implementation in FTLs

As an important component, our scheme will work with the address translator and wear-leveler in FTLs. We will discuss the implementation of FTLs to provide optimizing worst-case and average-case.

Address Translator: Address translation scheme will impact the data storing and garbage collection strategy. There are many address mapping schemes in FTLs[15], [16], [17], [18], [13], [19], which can be categorized to block-level mapping, page-level mapping and hybrid-level mapping scheme. In order to achieve the optimizing deterministic worst-case performance, our scheme adopts partial garbage collection technique and uses one block to handle the valid page copies and the coming data requests. Since the logical addresses from victim blocks and data requests are unpredictable, it requires address translator can freely translate the logical address to physical address and allocate data to any free pages. To achieve the optimizing average-case latency, the garbage collection should be postponed as late as possible. That requires the garbage collection trigger point independent from the logical address of the coming data request. On the other hand, over-provisioning can guarantee the reclaimed free pages based on global victim block selection in the full usage of flash memory. Therefore, data in our scheme can be stored in any data block and the page-level mapping information should be recorded.

The block-level mapping and hybrid-mapping schemes are not applicable to our garbage collection scheme because they cannot record the fine-granularity address mapping information. While, in page-level mapping scheme, data blocks can store data from any logical address since each local page to physical page mapping is recorded. As a result, the physical space of flash is fully utilized and the garbage collection can be triggered when there is only one free block. Therefore, address translator is implemented by adopting page-level address mapping scheme, which can provide intrinsic high performance and help our garbage collection scheme to achieve worst-case and average-case optimization. The drawback of page-level mapping scheme is the RAM space cost. There have been several on-demand approaches [20], [21], [18], and our scheme can adopt these approaches to significantly reduce the RAM cost.

]

Wear-leveler: Wear-leveler influences the endurance of the flash memory and relates to garbage collection strategy. Since our scheme optimizes the average-case performance by postponing garbage collection as late as possible, the erase counts of each block is improved. In order to balance the erase counts of each data block, our scheme will also check the block erase counts when selecting the victim block. When multiple blocks have the same least number of valid pages, our wear-leveler will select the block that has lower number of erase counts as the victim block. For many cold blocks that are not updated frequently, we do some block swaps between the hot block and cold block when the number of left free blocks does not meet the garbage collection threshold. Therefore, our scheme does not incur much garbage collection overhead and can improve the lifetime by balancing the block erase counts.

D. Performance Analysis

In this section, we analyze the system performance and space utilization of our scheme and give comparison with the representative real-time schemes.

The Worst System Performance: The system latency in the worst case consists of the data request execution time and the upper bound of garbage collection execution time. As shown in Table I, we use $\mathcal{U}(e_r)$ and $\mathcal{U}(e_w)$ to represent the page read operation time and write operation time, respectively. GFTL [2] uses block-level mapping scheme, where the logical page number is written into the OOB area. There are many OOB read operations when handling one data request. RFTL [3] uses hybrid-level mapping scheme and the mapping table is partially stored in the OOB area so that it also exists some OOB operations. $\mathcal{U}(t)$ denotes the upper bound of the system latency in the worst case and the value of Ideal and RTGC scheme depends on the upper bound of the entire garbage collection process execution time. GFTL schedules partial garbage collection step to any data request so impacts the read performance. Since the our scheme adopts page-level mapping scheme whose mapping table is maintained in the RAM, there is no extra OOB operations compared with GFTL and RFTL. Therefore, WAO-GC can improve the upper bound of the worst system latency.

Average System Performance: Garbage collection incurs the largest overhead in NAND flash memory storage systems due to the valid page copies and block erasing. WAO-GC adopts page-level mapping scheme that can fully use each page in the flash and delay the partial garbage collection only when there is about to out of space. Compared to our scheme, GFTL predefines a number of physical blocks as the write buffer and maintains a central garbage collection queue to decide which logical block executing garbage collection. RFTL pre-allocates three physical blocks (i.e., primary block, replacement block, and buffer block) to one logical block so that the execution of partial garbage collection is limited to the corresponding logical block. That is, once the primary physical block of the corresponding logical block is full, even there exists free space in many physical blocks belonging to other logical blocks, GFTL and RFTL all trigger garbage collection. $Threshold$ in Table I represents the garbage collection trigger condition. Ideal and our scheme trigger garbage collection only when there is one free block left in the flash. GFTL and RFTL trigger garbage collection when the physical block allocated to the logical block number (LBN) from the request is full. RTGC triggers garbage collection according to the space usage from the real-time tasks so cannot give a fixed threshold and buffer length without tasks information. Therefore, the garbage collection in GFTL or RFTL is invoked very early and the space utilization may be very low under the unbalance workload. As a result, the average system performance is degraded by such an early invoked garbage collection and the high number of block erase counts indirectly impacts on the endurance of the flash memory.

V. Evaluation

To evaluate the effectiveness of the proposed optimizing deterministic garbage collection scheme, we have conducted a set of experiments on a real embedded platform. In this section, we first introduce the experimental environment and performance metrics. Then we present experimental results and discussion. We compare WAO-GC with Pure-Page-Level scheme [22], RTGC [1], GFTL [2], and RFTL [3], in terms of four performance metrics: system latency in the worst case, average system latency, valid page copies, and block erase counts.

A. Experimental Environment and Performance Metrics

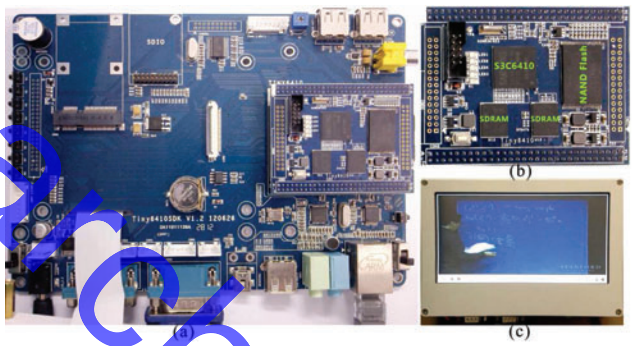


Fig. 5. Experimental platform. (a) A top view of the platform. (b) The core board that integrates an ARM 11 process core, 256 MB SDRAM, and an 8Gb NAND flash memory. (c) Video playing test.

We use an embedded developing board to conduct experiments. Both the proposed WAO-GC technique and representative schemes (i.e., Pure-Page-Level FTL [22], RTGC [1], GFTL [2], and RFTL [3]) are implemented in the embedded developing board. Figure 5 illustrates the embedded developing board. The board employs an ARM 11 processor core (Samsung S3C6410) with ARMv6 architecture. The ARM processor core is running at 532 MHz, and it consists of a 16 KB instruction cache and a 16KB data cache. The platform has a core board (Figure 5(b)) with an 8Gb NAND flash memory (K9K8G08U0B[23]) and 256 MB SDRAM, and a mother board with some physical interfaces.

Figure 6 illustrates the framework of our experimental platform. We conduct experiments using both real applications on the evaluation board and standard benchmarks which are commonly used in the research community. Specifically, we build a *NFS* (Net File System) [24] and perform file operations on the evaluation board. They can reflect the real workload of the system in accessing the NAND flash memory chip.

We also use standard benchmarks: Bonnie [25], Postmark [26] and Tiobench [27] to evaluate the proposed approach.

TABLE I. SERVICE GUARANTEE BOUNDS OF IDEAL CASE [22], RTGC [1], GFTL [2], RFTL [3], AND THE PROPOSED WAO-GC

Bounds	Ideal	RTGC	GFTL	RFTL	WAO-GC
$\mathcal{U}(e_r)$	t_{rdpg}	t_{rdpg}	$t_{rdpg} + \pi t_{rdoob}$	$t_{rdpg} + t_{rdoob}$	t_{rdpg}
$\mathcal{U}(e_w)$	t_{wrpg}	t_{wrpg}	t_{wrpg}	$t_{wrpg} + t_{rdoob}$	t_{wrpg}
$\mathcal{U}(\lambda)$	π	$\sigma \times \pi$	π	π	$\sigma \times \pi$
$\mathcal{U}(t)$	$U(e_w) + \mathcal{U}(e_G)$	$\mathcal{U}(e_w) + \mathcal{U}(e_G)$	$t_{er} + max\{U(e_r), U(e_w)\}$	$max\{U(e_r), t_{er} + U(e_w)\}$	$max\{U(e_r), t_{er} + U(e_w)\}$
$Threshold$	1	N/A	$isFull(LBN)$	$isFull(LBN)$	1
L_{buf}	N/A	N/A	$N(k+1)/2$	$2 \times \pi \times N$	k

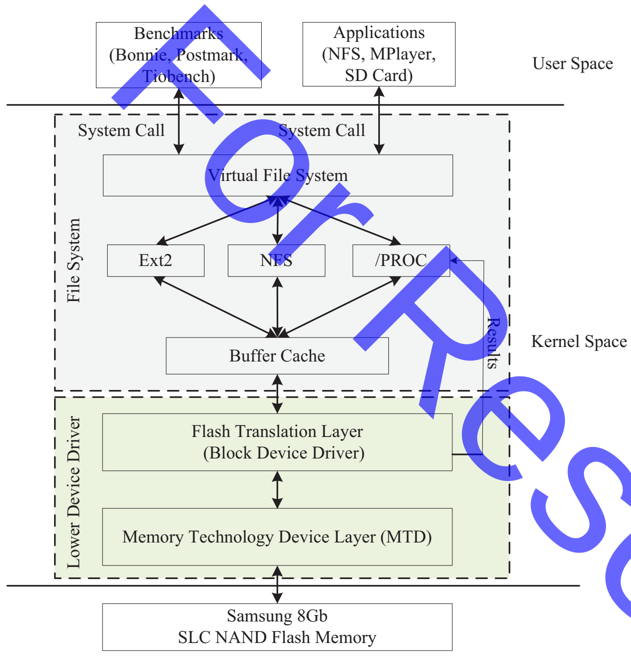

Benchmarks (Bonnie, Postmark, Tiobench)
Applications (NFS, MPlayer, SD Card)
User Space
System Call
System Call
Virtual File System
File System
Ext2
NFS
/PROC
Results
Buffer Cache
Kernel Space
Lower Device Driver
Flash Translation Layer (Block Device Driver)
Memory Technology Device Layer (MTD)
Samsung 8Gb SLC NAND Flash Memory

Fig. 6. The framework of experimental platform.

These benchmarks have different features. Benchmark Bonnie can perform a number of file seeks and meta-data operations to test the file system performance. Benchmark Tiobench uses many threads concurrently accessing a specified file directory, while benchmark Postmark creates a large amount of files under a specified file directory. VFS (Virtual File System) is used to hide different file system features and provide generic interfaces for user space programs. When our applications or benchmarks work under a specified file directory, the file operations are passed to the file system through the system call. After the file system receives these requests, it interprets these requests and issues requests to the lower device driver, such as FTL, mainly in terms of a sector (or page) reading and writing. Buffer cache is adopted by the file system to improve the file system performance. If the buffer cache could not handle requests or the system is ideal, the victim or cached data will be transferred to the lower FTL. FTL maps these requests to the physical NAND flash memory with the help of MTD (Memory Technology Device). Our experimental results are generated in the FTL layer. Since user could not access kernel space data directly, we make use of /PROC file system to build communication between user space and kernel space by means of creating a specified file under the */proc* file directory. Our test results are obtained through /PROC file system.

We implemented Pure-Page-Level FTL, RTGC, GFTL, RFTL and our scheme as block device drivers and ported ARM Linux 2.6.38 on this embedded platform. The Linux kernel loads these drivers as kernel modules implemented between the file system and the MTD layer. After these FTLs are inserted into the kernel, the corresponding device files will be created under the */dev* directory. Then with the help of file system formatting tools such as *mkfs.ext2*, the file system information is written in the NAND flash memory. By mounting these device files to a specified file directory, the NAND flash memory can be operated through normal file operations, such as file creation, file reading, etc. The basic parameters of the NAND flash memory is shown in Table II and we use the Linux kernel function *do_gettimeofday* to measure the system latency. Due to the operating system handling time, the latency time from the kernel function may not be consistent with the pure NAND flash memory operations executing time. For fair comparison, we format the entire NAND flash memory first before every experimental evaluation.

TABLE II. PARAMETERS OF THE NAND FLASH MEMORY (K9K8G08U0B[23]).

Parameter	Value
Total capacity	8Gb
The number of planes per element	4
The number of blocks per plane	2048
The number of pages per block	64
Page size	2KB
Block size	128KB
Endurance	100K P/E Cycles
Page read latency	$25\mu s$
Page write latency	$200\mu s$
Block erase latency	$1500\mu s$

We use the following metrics to evaluate the performance of our approach: 1) *System latency in the worst case*. It is the longest system latency when handling data requests from the benchmarks and applications. 2) *Average system latency*. We divide the total request latency by the counts to get the average system latency. 3) *Valid page copies*. We record the number of valid page copies to show the overhead of the garbage collections. 4) *Block erase counts*. We also measure the number of block erase counts to show the frequency of the garbage collection.

B. *Results and Discussion*

In this section, we present the experimental results in terms of four performance metrics: system latency in the worst case, average system latency, valid page copies and block erase counts. we use *Pure-Page-Level*, *RTGC*, *GFTL*, *RFTL*, and *WAO-GC* to represent the evaluation results generated by the schemes in [22], [1], [2], [3], and the proposed scheme, respectively.

1) Worst Case System Latency: By making use of partial garbage collection technology, GFTL, RFTL, and our scheme can guarantee the system latency of the data request in the

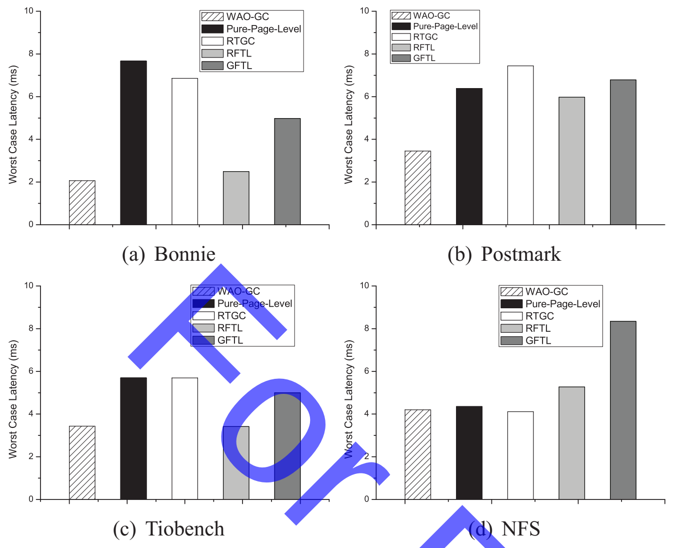


Fig. 7. Worst Case Latency

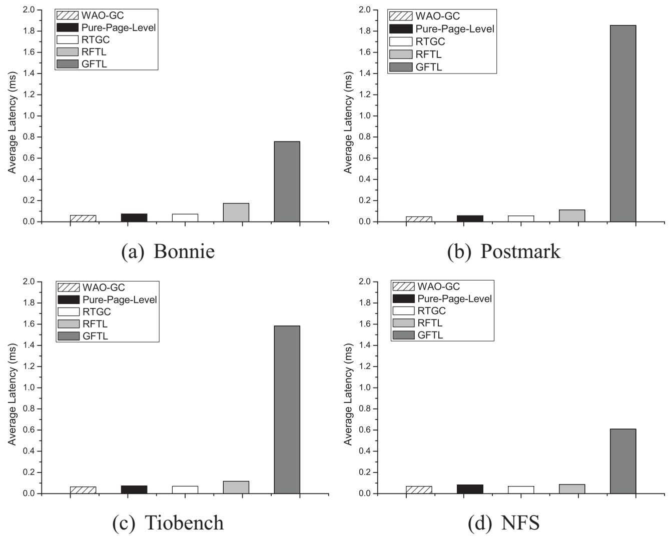


Fig. 8. Average System Latency

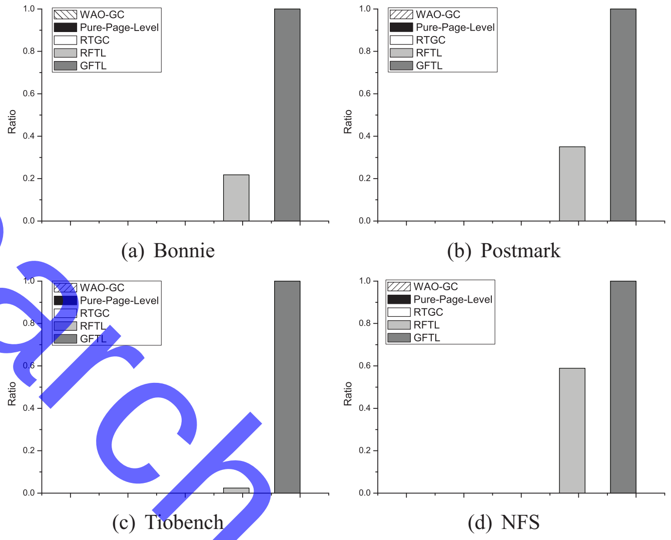
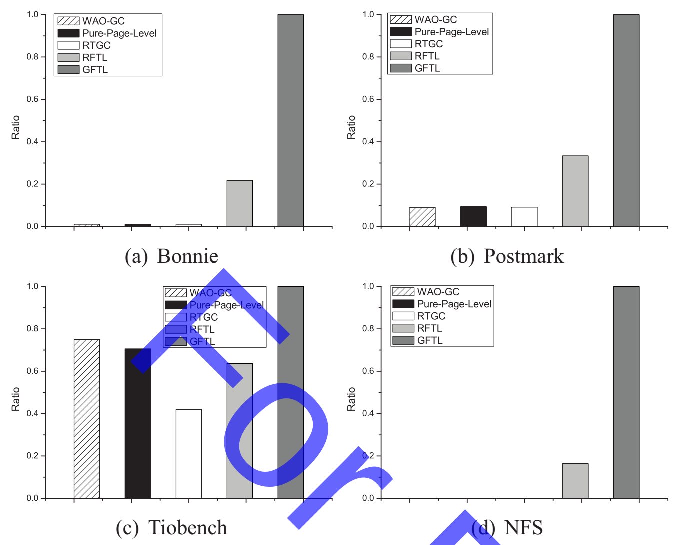


Fig. 9. The Normalized Number of Valid Page Copies

worst case. As shown in Figure 7, WAO-GC can achieve lower system latency in the worst case compared to GFTL, RFTL. That because, both GFTL and RFTL has extra OOB operations to get the real mapping information, while WAO-GC maintains all page-level mappings into RAM. The benefits on worst case system latency in our scheme are mainly from the page-level mapping scheme, which will also incur large RAM cost. In the experimental results, WAO-GC can improve 47.14% and 19.80% on system latency in the worst case compared to GFTL and RFTL, respectively. Pure-Page-Level cannot provide a deterministic garbage collection execution time so that their worst system latency is worse than that in our scheme. Since the running operation system is a general Linux and is not modified to support real-time tasks, RTGC cannot reclaim free pages for each real-time task so cannot provide deterministic data request serve time. However, RTGC adopts over-provisioning strategy that can reduce the number of valid pages in the victim block. Therefore, the worst system latency of RTGC is lower than that in Pure-Page-Level FTL scheme. Compared to Pure-Page-Level and RTGC, WAO-GC can archive 40.51% and 40.24% reduction on worst case system latency, respectively.

2) Average System Latency: Given that the worst case does not happen frequently, optimizing garbage collection for giving a deterministic worst case system latency should not bring too much overhead to the average system latency. Therefore the average system latency is one of the most important metrics represent the system performance. The experimental results are shown in Figure 8. From the results, GFTL and RFTL suffer from significantly average performance degradation compared with Pure-Page-Level scheme and our scheme. That is because our scheme adopts page-level address mapping scheme that can freely manage the data and postpone the partial garbage collection as late as possible. Compare to our scheme, GFTL adopts block-level mapping scheme and once a logical block is fully used, the corresponding physical block is added to central garbage collection queue to do partial garbage collection. As a result, there is a large number of unnecessary and early triggered garbage collections. RFTL pre-allocates three physical blocks to one logical block and when the logical block is full, the partial garbage collection is triggered within the allocated blocks. Therefore, RFTL also triggers garbage collection early and requires lots of extra physical flash space. In experimental results, our scheme can achieve an average 47.01% and 93.48% reduction on average system latency compared to RFTL and GFTL and even sightly better than that in Pure-Page-Level mapping scheme due to over-provisioning strategy that can reduce valid page copies in victim block.

3) Valid Page Copies: The number of valid page copies in garbage collection decides the time overhead of the garbage collection process. By making use of page-level address mapping scheme, WAO-GC can fully use the free pages in the flash and postpone the garbage collection. Moreover, the over-provisioning strategy limits the logical address space so that reduce the number of valid pages in the victim block. As a result, there are more invalid pages in victim blocks when the flash memory is almost full. In GFTL and RFTL, once the logical block is full, the partial garbage collection is triggered, even though there may exist many free blocks belongs to other logical blocks. The early garbage collections reduce the chance to invalid the page on handling the data request in the future. As shown in Figure 9, GFTL and RFTL have a lot number of



(a) Bonnie (b) Postmark (c) Tiobench (d) NFS

Fig. 10. The Normalized Number of Block Erase Counts

valid page copies while Pure-Page-Level scheme, RTGC and our scheme which adopt page-level mapping scheme have very low valid page copies overhead. Compare to Pure-Page-Level scheme, both RTGC and our scheme can archive fewer number of valid page copies by using over-provisioning.

4) Block Erase Counts: The number of block erase counts will influence the average system response time and the endurance of the NAND flash memory. As shown in the Figure 10, our scheme can significantly reduce 50.08% and 71.64% block erase counts compared with GFTL and RFTL, respectively. That is because, for central partial garbage collection policy in GFTL and distributed partial garbage collection policy in RFTL, the condition to trigger garbage collection depends on the usage of logical blocks. There is a lot of unnecessary garbage collection operations in these schemes. Since RTGC reduces the logical address space to guarantee the reclaimed free space, it can archive lower block erase counts compared to Pure-Page-Level scheme. We observe that the number of block erase counts of our scheme is very close to the one in RTGC.

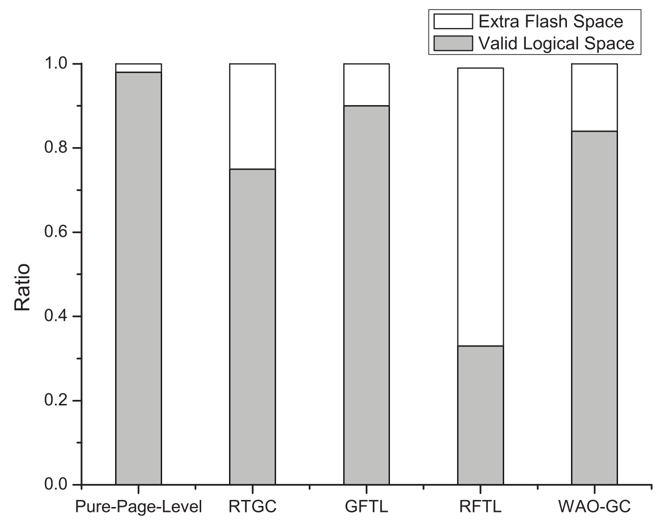


Fig. 11. The Space Utilization Ratio.

C. Overhead

In order to provide deterministic garbage collection and optimizing average performance, both our scheme and previous schemes cost extra flash space as the write buffer or using for over-provisioning. Due to different address mapping schemes, the RAM space overhead are also different. Our scheme can get the space utilization ratio σ according to the space configuration. In the experiment, as shown in Figure 11 the space utilization is $\mathcal{U}(\sigma) = ((64-1)\times 6)/((6+1)\times 64) = 84.38\%$, where $\alpha = 6$. Our scheme costs about 15.62% flash space. In GFTL, there is a central write buffer to serve the coming write requests when running partial garbage collection. While in RFTL, it exists a distributed write buffer (i.e., buffer block) for each logical block. The buffer length in GFTL is limited by $N(k+1)/2$ so the flash space overhead is about 10.16%. RFTL pre-allocated three physical blocks to one logical block thus it costs about 66.7% physical address space. Since RTGC cannot get the real-time task information, we set $\sigma = 0.75$ as the ratio between the logical space and physical space. Pure-Page-Level scheme does not apply any optimizing mechanisms so that the space utilization is closed to 100%. Since our scheme adopts page-level mapping scheme, the RAM overhead of our scheme is larger than those adopt block-level or hybrid-level mapping schemes. Although our scheme has physical space and RAM space cost, it can not only guarantee the serve time under worst case, but also optimize the average system latency compared with previous works.

VI. Related Works

In the previous studies, there are several research works about real-time NAND flash memory storage systems. Kuo et al [1] proposed a real-time garbage collection mechanism. The mechanism needs to get the real-time task information from file system and allocate garbage collection tasks to each real-time task for replenishing required free pages. In order to determine the reclaimed free pages after each garbage collection operations, the mechanism limits the logical address space be smaller than physical space. As the over-provisioning strategy, therefore, RTGC can guarantee each real-time task has enough free pages to execute. However, it does not consider about the average performance. Moreover, it needs extra real-time task information from the file system which may cause significant modifications to current file system so it is not a general scheme for NAND flash memory storage systems.

Partial garbage collection is proposed in GFTL [2] and block-level mapping scheme is used. In GFTL, there is a central garbage collection queue to record the block number needed to execute garbage collection and a write buffer to handle the coming request when the corresponding data block is full. Once the block is full, the block number will be added into the garbage collection queue and wait to run garbage collection. Before the corresponding block is erased, the coming write request will be written to the buffer. GFTL can guarantee the buffer length has an acceptable upper bound. However, there are many unnecessary garbage collections with a lot of valid page copies. Many hot data block may be erased frequently while other block may stay free for a long time. To solve the problem, Qin et al proposed a distributed partial garbage collection scheme called RFTL [3]. In RFTL, each logical block owns three physical blocks and the partial garbage collection process is distributed to the corresponding logical block. RFTL can improve the average system performance compared to GFTL but it cannot solve performance degradation from unnecessary garbage collections. Moreover, the low space utilization that costs more than 60% physical space is another problem in RFTL.

Lee et al [28] proposed a preemptible garbage collection (PGC) for Solid State Drives. In their scheme, PGC can identify preemption points that can minimize the preemption overhead. Furthermore, PGC can merge incoming I/O requests to enhance the performance of SSDs. However, it requires many techniques supported by SSDs. The target of PGC is to improve the system performance instead of providing deterministic garbage collection. Therefore, PGC cannot optimize the worst case system latency.

VII. Conclusion

In this paper, we have proposed a worst-case and average-case joint optimization scheme for garbage collection in NAND flash. By making use of partial garbage collection technique and over-provisioning strategy, our scheme can give an optimizing deterministic system latency in the worst case and further optimize the average system performance. We have evaluated our scheme using a set of benchmarks and compared with representative works. The experimental results show that our scheme can improve both the average and the worst system performance with very low extra flash space requirements.

Acknowledgments

This work is supported by the National Key Basic Research Program of China(2014CB340703) and the National Natural Science Foundation of China (No.91318301, No.61321491, No.61272103, No.61373049), the grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (GRF 152138/14E), the Germany/Hong Kong Joint Research Scheme sponsored by the Research Grants Council of Hong Kong and the Germany Academic Exchange Service of Germany (Reference No.G_HK021/12), National 863 Program (2013AA013202), and the Hong Kong Polytechnic University (4-ZZD7,G-YK24, G-YM10 and G-YN36).

References

[1] Li-Pin Chang, Tei-Wei Kuo, and Shi-Wu Lo, "Real-Time Garbage Collection for Flash-Memory Storage Systems of Real-Time Embedded Systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 3, no. 4, pp. 837–863, 11 2004.

[2] Siddharth Choudhuri and Tony Givargis, "Deterministic service guarantees for NAND flash using partial block cleaning," in *CODES+ISSS'08*, 2008, pp. 19–24.

[3] Zhiwei Qin, Yi Wang, Duo Liu, and Zili Shao, "Real-Time Flash Translation Layer for NAND Flash Memory Storage Systems," in *RTAS'12*, 2012.

[4] Yiran Chen, Weng-Fai Wong, Hai Li, and Cheng-Kok Koh, "Processor caches built using multi-level spin-transfer torque ram cells," in *ISLPED'11*, Aug 2011, pp. 73–78.

[5] Yaojun Zhang, Xiaobin Wang, and Yiran Chen, "Stt-ram cell design optimization for persistent and non-persistent error rate reduction: A statistical design view," in *ICCAD '11*, Piscataway, NJ, USA, 2011, ICCAD '11, pp. 471–477, IEEE Press.

[6] Hong-Phuc Trinh, Weisheng Zhao, J.-O. Klein, Yue Zhang, D. Ravelsona, and C. Chappert, "Magnetic adder based on racetrack memory," *Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 60, no. 6, pp. 1469–1477, June 2013.

[7] Yiqun Wang, Yongpan Liu, Yumeng Liu, Daming Zhang, Shuangchen Li, Baiko Sai, Mei-Fang Chiang, and Huazhong Yang, "A compression-based area-efficient recovery architecture for nonvolatile processors," in *DATE'12*, March 2012, pp. 1519–1524.

[8] Boxun Li, Yu Wang, Yiran Chen, H.H. Li, and Huazhong Yang, "Ice: Inline calibration for memristor crossbar-based computing engine," in *DATE'14*, March 2014, pp. 1–4.

[9] Weisheng Zhao, S. Chaudhuri, C. Accoto, J.-O. Klein, C. Chappert, and P. Mazoyer, "Cross-point architecture for spin-transfer torque magnetic random access memory," *Nanotechnology, IEEE Transactions on*, vol. 11, no. 5, pp. 907–917, Sept 2012.

[10] Jingtong Hu, Chun Jason Xue, Qingfeng Zhuge, Wei-Che Tseng, and Edwin H.-M. Sha, "Write activity reduction on non-volatile main memories for embedded chip multiprocessors," *ACM Trans. Embed. Comput. Syst.*, vol. 12, no. 3, pp. 77:1–77:27, Apr. 2013.

[11] Jingtong Hu, C.J. Xue, Qingfeng Zhuge, Wei-Che Tseng, and E.H. Sha, "Data allocation optimization for hybrid scratch pad memory with sram and nonvolatile memory," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 21, no. 6, pp. 1094–1102, June 2013.

[12] Liang Shi, C.J. Xue, and Xuehai Zhou, "Cooperating write buffer cache and virtual memory management for flash memory based systems," in *RTAS'11*, April 2011, pp. 147–156.

[13] Zhiwei Qin, Yi Wang, Duo Liu, Zili Shao, and Yong Guan, "MNFTL: an efficient flash translation layer for MLC NAND flash memory storage systems," in *DAC'11*, 2011, pp. 17–22.

[14] Tianzheng Wang, Duo Liu, Zili Shao, and Chengmo Yang, "Write-activity-aware page table management for pcm-based embedded systems," in *ASP-DAC'12*, Jan 2012, pp. 317–322.

[15] Chin-Hsien Wu, Tei-Wei Kuo, and Li Ping Chang, "An efficient b-tree layer implementation for flash-memory storage systems," *ACM Transcations on Embedded Computing Systems*, vol. 6, no. 3, July 2007.

[16] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sang-won Park, and Ha-Joo Song, "A log buffer-based flash translation layer using fully-associative sector translation," *ACM Transcations on Embedded Computing Systems*, vol. 6, no. 3, July 2007.

[17] Duo Liu, Yi Wang, Zhiwei Qin, Zili Shao, and Yong Guan, "A space reuse strategy for flash translation layers in slc nand flash memory storage systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, no. 6, pp. 1094–1107, 2012.

[18] Qi Zhang, Xuandong Li, Linzhang Wang, Tian Zhang, Yi Wang, and Zili Shao, "Optimizing translation information management in nand flash memory storage systems," in *Proceedings of 18th Asia and South Pacific Design Automation Conference*, 2013, ASP-DAC '13, pp. 326–331.

[19] Zhiwei Qin, Yi Wang, Duo Liu, and Zili Shao, "Demand-based block-level address mapping in large-scale NAND flash storage systems," in *CODES+ISSS'10*, 2010, pp. 173–182.

[20] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar, "DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings," in *ASPLOS'09*, 2009, pp. 229–240.

[21] Zhiwei Qin, Yi Wang, Duo Liu, and Zili Shao, "A Two-Level Caching Mechanism for Demand-Based Page-Level Address Mapping in NAND Flash Memory Storage Systems," in *RTAS'11*, 2011, pp. 157–166.

[22] Amir Ban, "Flash file system," *US patent 5,404,485*, 1995.

[23] "Samsung electronics.samsung K9K8G08U0B(v0.1)-8Gb SLC NAND Flash data sheet," July 2008.

[24] "NFS," *http://nfs.sourceforge.net/nfs-howto/*, 2013.

[25] B. Tim, "Bonnie," *http://www.garloff.de/kurt/linux/bonnie/*, 2013.

[26] J. Katcher, "Postmark: A new file system benchmark[r]," *Technical Report TR3022, Network Appliance*, 1997.

[27] M. Kuoppala, "Tiobench-threaded I/O bench for linux[j]," 2002.

[28] Lee Junghee, Kim Youngjae, M. Shipman Galen, Oral Sarp, and Kim Jongman, "Preemptible i/o scheduling of garbage collection for solid state drives," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 2, pp. 247–260, 2013.