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

Technical Report No. NJU-SEG-2015-IJ-003

2015-IJ-003

Lazy-RTGC: A Real-Time Lazy Garbage Collection Mechanism with Jointly Optimizing Average and Worst Performance for NAND Flash Memory Storage Systems

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

Transactions on Design Automation of Electronic Systems 2015

Lazy-RTGC: A Real-Time Lazy Garbage Collection Mechanism with Jointly Optimizing Average and Worst Performance for NAND Flash Memory Storage Systems

QI ZHANG, XUANDONG LI, LINZHANG WANG, and TIAN ZHANG, Nanjing University
YI WANG, Shenzhen University and The Hong Kong Polytechnic University
ZILI SHAO, The Hong Kong Polytechnic University

Due to many attractive and unique properties, NAND flash memory has been widely adopted in mission-critical hard real-time systems and some soft real-time systems. However, the nondeterministic garbage collection operation in NAND flash memory makes it difficult to predict the system response time of each data request. This article presents *Lazy-RTGC*, a real-time lazy garbage collection mechanism for NAND flash memory storage systems. Lazy-RTGC adopts two design optimization techniques: on-demand page-level address mappings, and partial garbage collection. On-demand page-level address mappings can achieve high performance of address translation and can effectively manage the flash space with the minimum RAM cost. On the other hand, partial garbage collection can provide the guaranteed system response time. By adopting these techniques, Lazy-RTGC jointly optimizes both the average and the worst system response time, and provides a lower bound of reclaimed free space. Lazy-RTGC is implemented in FlashSim and compared with representative real-time NAND flash memory management schemes. Experimental results show that our technique can significantly improve both the average and worst system performance with very low extra flash-space requirements.

Categories and Subject Descriptors: B.3.2 [**Design Styles**]: Mass Storage; D.4.2 [**Storage Management**]: Garbage Collection; D.4.7 [**Organization and Design**]: Real-TIme Systems and Embedded Systems

General Terms: Design, Performance

Additional Key Words and Phrases: NAND flash memory, real-time system, garbage collection, storage systems

ACM Reference Format:
Qi Zhang, Xuandong Li, Linzhang Wang, Tian Zhang, Yi Wang, and Zili Shao. 2015. Lazy-RTGC: A real-time lazy garbage collection mechanism with jointly optimizing average and worst performance for NAND flash memory storage systems. ACM Trans. Des. Autom. Electron. Syst. 20, 3, Article 43 (June 2015), 32 pages.
DOI: http://dx.doi.org/10.1145/2746236

The work described in this article is partially supported by National Key Basic Research Program of China (2014CB340703) and the National Natural Science Foundation of China (no. 91318301, no. 6132141, 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 (ref. no. G-HK021/12), National 863 Program 2013AA013202, the Hong Kong Polytechnic University (4-ZZD7, G-YK24, G-YM10, and G-YN36), and Guangdong Natural Science Foundation (2014A030310269).
Authors' addresses: Q. Zhang, X. Li, L. Wang, T. Zhang, State Key Laboratory for Novel Software Technology, Department of Computer Science, Nanjing University, Nanjing, China; Y. Wang (corresponding author), College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China and Department of Computing, The Hong Kong Polytechnic University, Hong Kong; email: ssywang@comp.polyu.edu.hk; Z. Shao, Department of Computing, The Hong Kong Polytechnic University, Hong Kong.

DOI: http://dx.doi.org/10.1145/2746236

1. INTRODUCTION

NAND flash memory has many attractive properties, such as low power consumption, fast access time, and shock resistance. Therefore, NAND flash memory has been widely adopted in mission-critical hard real-time systems such as aerospace [AEEC 1991] and in soft real-time systems such as iPhones and tablets. Different from other computing systems in real-time systems, a NAND flash memory storage system needs to provide both the worst and the average system response time. However, due to the constraint of "out of place update" in flash memory, the number of valid page copy operations in the victim block is unpredictable. This unfavorable characteristic may negatively impact the predictability of system response time and make the management of the garbage collection process become the major performance bottleneck. In this article, we propose a real-time lazy garbage collection mechanism to achieve the real-time property by jointly optimizing the average and worst system performance in NAND flash memory storage systems.

In past decades, many studies have been conducted on the management of flash memory storage systems. A lot of work has been conducted on storage system architecture design [Wang et al. 2014; Chang et al. 2013a, 2014a; Hsieh et al. 2013, 2014; Huang et al. 2013], while others studied the flash translation-layer design [Wu and Kuo 2006; Lee et al. 2008; Chung et al. 2009; Wu and Lin 2012]. Several techniques have been proposed to improve the system performance of NAND flash memory storage systems [Hu et al. 2010; Jung et al. 2010; Guan et al. 2013; Huang et al. 2014a]. Different from prior works, our scheme aims to guarantee the worst system response time and, at the same time, optimize the average system performance in NAND flash memory storage systems. There are also many studies [Kim et al. 2000; Bacon et al. 2003; Chang and Wellings 2010] focusing on real-time garbage collection for computing systems with dynamic memory requirements. However, for NAND flash memory storage systems, the concept of garbage collection is different from that in the dynamic memory version due to many unique constraints.

Only a few works focus on the real-time garbage collection techniques for NAND flash storage systems. These works mainly focus on two directions: task-driven free-space replenishment [Chang et al. 2004] and partial garbage collection [Choudhuri and Givargis 2008]. The first direction promises to replenish several free pages to ensure that there will be enough free space to execute each real-time task. In order to provide a lower bound of reclaimed space, these techniques have to store the runtime information of each real-time task, which normally requires significant modification to existing file systems. In another direction, a partial garbage collection mechanism partitions one garbage collection process into several steps and distributes these steps to different time slots. The partial garbage collection mechanism needs to allocate some extra physical blocks as a write buffer. A queue has to be maintained to record the garbage collection information. In partial garbage collection schemes, the process of garbage collection is triggered very early, which may further incur lots of unnecessary garbage collection with a large number of extra valid page copies.

In order to solve the performance issue of partial garbage collection, Qin et al. [2012] proposed a real-time *flash translation layer* (FTL) called RFTL. In RFTL, the partial garbage collection is distributed to different logical blocks, and each logical block is pre-allocated to three physical blocks (i.e., primary block, buffer block, and replacement block). RFTL can significantly reduce the number of unnecessary garbage collection processes. However, the flash-space utilization of RFTL is very low and the garbage collection process is also triggered early due to its fixed physical block allocation. Although the previous schemes can satisfy real-time requirements, they make no specific attempt to provide a guaranteed average system response time. Moreover,

the space utilization in these schemes is another critical issue, as these schemes necessarily allocate a large amount of extra flash memory space to ensure real-time performance.

In this article, we present *Lazy-RTGC*, a real-time lazy garbage collection mechanism that can ensure guaranteed system response time for both the *worst case* and *average case* with the minimum space requirement. Lazy-RTGC adopts the page-level mapping scheme that can fully utilize the flash memory space and effectively postpone the garbage collection process as late as possible. Therefore, Lazy-RTGC does not need to pre-allocate a fixed number of physical space as the write buffer, which can provide optimized average system response time. Through the space configuration, Lazy-RTGC can guarantee the number of reclaimed free space more than a lower bound after each garbage collection. The entire process of garbage collection is divided into a set of partial garbage collections which can provide an upper bound of service time. The reclaimed free space from the previous set of partial garbage collections, considered as a free write buffer, can be further used in the next partial garbage collection set. As a result, our scheme can not only guarantee the reclaimed free space of garbage collection, but also provide a deterministic service time of garbage collection in the worst case.

We have performed analysis and the results show that the page-level mapping scheme is the best option to sufficiently manage the flash space and delay garbage collection. In order to reduce the large mapping table in the page-level address mapping scheme, Lazy-RTGC adopts a demand-based page-level mapping scheme that can significantly reduce the RAM footprint and achieve similar performance as block-level mapping schemes. Only the on-demand address mappings will be allocated and stored in the cache. Moreover, Lazy-RTGC requires no changes to the file system or NAND flash memory chip, so it is a general strategy that can be applied to any page-level address mapping schemes.

Lazy-RTGC is implemented in the FlashSim framework [Kim et al. 2009], and a set of benchmarks from both real-world and synthetic traces is used to evaluate the effectiveness of Lazy-RTGC. In the evaluation, we compare our scheme with FSR [Chang et al. 2004], GFTL [Choudhuri and Givargis 2008], RFTL [Qin et al. 2012], and the Pure-Page-Level mapping scheme [Ban 1995] in terms of system response time in the worst case, average system response time, valid page copies, block-erase counts, and the space utilization ratio. FSR is the *free-space replenishment* strategy, a main part in Chang et al. [2004]. It can guarantee the reclaimed free space after each garbage collection process. GFTL [Choudhuri and Givargis 2008] and RFTL [Qin et al. 2012] are representative schemes that adopt a partial garbage collection technique. The Pure-Page-Level scheme [Ban 1995] is a page-level address mapping scheme without applying any real-time mechanisms.

The experimental results show that our scheme can achieve better worst- and average-case system performance compared with previous work. For system response time in the worst case, our scheme can achieve 90.58% improvement compared with the pure-page-level FTL scheme. For average response time, our scheme can improve 94.08% and 66.54% average system performance compared with GFTL and RFTL, respectively. For the number of valid page copies, our scheme can achieve 95.36% and 86.11% reductions compared to GFTL and RFTL, respectively. Since many valid page copies are reduced, our scheme can also significantly reduce the number of block-erase counts. For space utilization, our scheme can achieve 87.5% space utilization, which is very close to GFTL and much higher compared to RFTL. Therefore, by costing small extra flash space, our scheme can not only provide an upper bound of the worst system response time, but also significantly improve the average system performance and endurance of NAND flash memory storage systems.

This article makes the following contributions.

—We present for the first time a real-time lazy garbage collection mechanism with joint optimization of average- and worst-case system response time.
—We adopt on-demand page-level address mapping that can delay the garbage collection process and provide guaranteed system response time with minimum space cost.
—We demonstrate the effectiveness of Lazy-RTGC by comparing it with representative works in the literature using a set of real traces.

The rest of this article is organized as follows. We give the models and clarify the problem in Section 2. Section 3 presents our technique details and task scheduling algorithm. We analyze the task schedulability on the time level and space level in Section 4. In Section 5, we extend our scheme to an on-demand page-level mapping scheme. In Section 6, we present the evaluation of our scheme. The related work in the literature is presented in Section 7. Finally, we give the conclusion and future work in Section 8.

2. MODELS AND PROBLEM STATEMENT

In this section, we formulate task models and clarify the problem we want to solve in this work.

2.1. Task Models

In this article, each I/O request issued from the file system to NAND flash chip is modeled as an independent real-time task $T = \{p_T, e_T, w_T, d_T, r_T\}$, where p_T, e_T, w_T, d_T, and r_T denote the period, execution time, and the maximum number of page writes per period, deadline, and the release time of the task, respectively. Without loss of generality, we assume that the deadline d of each task is equal to period p. There are two kinds of data request task: read task (T_r) and write task (T_w). Here w is equal to 0 if it is a read task; w is equal to 1 if it is a write task. p_T represents the frequency of read and write requests issued from the file system, while e_T denotes the time of executing a read or write request, which is determined by the specific NAND flash memory. The lower bound on p_T (denoted as $\mathcal{L}(p_T)$) determines the maximum arrival rate that the flash storage system can handle. The upper bound on e_T (denoted as $\mathcal{U}(e_T)$) represents the longest execution time of a request that would be served by the flash memory storage system.

The release time of the data request task depends on the request execution time and the location of the task in the queue. In our model, the garbage collection process is independent from the logical address of coming data tasks. That is, the execution of the data request task in the queue does not influence the current garbage collection process. Therefore we can calculate the release time of each task T_i as follows:

$$r_{T_i} = \sum_{j}^{n_w} e_{T_j} + \sum_{k}^{n_r} e_{T_k} + e_{T_i}. \tag{1}$$

In Eq. (1), n_w and n_r represent the number of write tasks and read tasks in queue before T_i, respectively. Therefore the release time of T_i is the sum of three execution times, that is, the execution time of read tasks before T_i, of write tasks before T_i, and of T_i.

Due to the constraint of "out-of-place update" in flash memory, garbage collector is used to reclaim free pages from obsolete invalid pages. In this article, we model the process of garbage collection as a garbage collection task $G = \{p_G, e_G, w_G, d_G, r_G\}$, where p_G and e_G represent the period and the total execution time, respectively. The

total execution time of a garbage collection task includes the time cost to copy valid pages in a victim block and the time cost to erase the victim block. w_G denotes the number of reclaimed pages after garbage collection. d_G and r_G, the same as defined in T, represent the deadline and release time of the garbage collection task. When the number of free pages is smaller than the predefined threshold, the garbage collector will select a victim block and schedule the corresponding garbage collection task for reclaiming the victim block. Since the atomic operations cannot be interrupted in the flash memory, the garbage collection task G and the data request task T exist in dependency. That is, the data request task cannot be executed until the completion of the scheduled garbage collection task.

Based on the task models, we give the definitions of system response time in NAND flash memory storage systems. There are two main processes during the system response time. One process is the execution time of a data request task T_i, which includes the time cost for logical-to-physical address translation (denoted by t_{addr}) and the time cost for the atomic operations (read or write operations). Another process is time cost to schedule garbage collection tasks before executing T_i. The system response time is given in Eq. (2).

$$t_{res} = t_{exec} + t_{addr} + t_{gc}. \tag{2}$$

In Eq. (2), t_{res}, t_{exec}, t_{addr}, and t_{gc} represent system response time, data request execution time, address translation time, and garbage collection time, respectively. Since the address translation overhead in the RAM is at least an order of magnitude less than the flash operation time, we mainly discuss t_{exec} and t_{gc} of the tasks, which correspond to e_T and e_G, respectively, in our task models. Furthermore, we define the response time for one data request task as follows.

Definition 2.1 (*System Response Time*). Given a data request task T_i the garbage collector schedules a set of garbage collection tasks $V_G = \{G_1, G_2, \ldots, G_n\}$ ($n = 0$ if there is no scheduled task G_j) before the execution of task T_i. The system response time for task T_i contains the execution time of task (e_{T_i}) and the total execution time of the scheduled garbage collection tasks (V_G). That is,

$$\mathcal{R}_{T_i} = e_{T_i} + \sum_{j=1}^{n} e_{G_j}, \qquad n \geq 0. \tag{3}$$

Without loss of generality, we assume the garbage collector schedules only one garbage collection task for each data request task. Then, we give the definitions of average response time and worst-case response time in our models as follows.

Definition 2.2 (*Average System Response Time*). Given a set of data request tasks $V_T = \{T_1, T_2, \ldots, T_n\}$, the average system response time is the arithmetic mean of system response time of the tasks in the V_T. That is,

$$\mathcal{R}_{avg} = \frac{1}{n} \sum_{i=1}^{n} \mathcal{R}_{T_i}, \qquad n > 0. \tag{4}$$

Definition 2.3 (*Worst-Case System Response Time*). The worst-case system response time in the flash memory is bounded by the worst-case execution time of the data request task ($\mathcal{U}(e_T)$) and the worst-case execution time of the garbage collection task ($\mathcal{U}(e_G)$). That is,

$$\begin{aligned} \mathcal{U}(\mathcal{R}_T) &= \mathcal{U}(e_T) + \mathcal{U}(e_G) \\ &= \max\{t_{rd}, t_{wr}\} + (\pi - 1)(t_{rd} + t_{wr}) + t_{er}. \end{aligned} \tag{5}$$

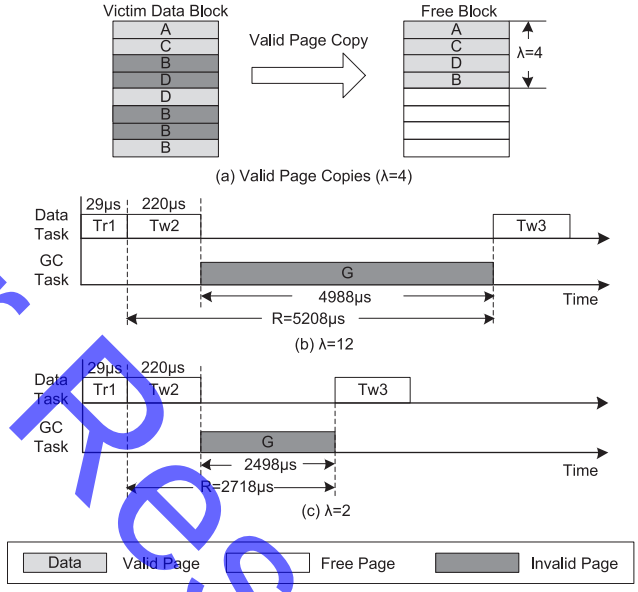


Fig. 1. An example of nondeterministic garbage collection.

In Eq. (5), t_{rd} and t_{wr} denote the execution time of reading and writing one page, respectively. π represents the total number of pages in one block. In the worst case, the garbage collector selects a victim block with $\pi - 1$ valid pages. We present the execution process of the garbage collection task in Section 2.2 and show why it causes nondeterministic system response time.

2.2. Nondeterministic Garbage Collection

A garbage collection task mainly consists of two subtasks: valid page copies and block erase. Valid page copy, also called *atomic copy* [Chang et al. 2004], reads the valid data in each valid page from the victim block and writes the data into another free page. After all valid pages are copied, the erase operation is invoked so that it can get the reclaimed free space. The total execution time (e_G) of one garbage collection task is defined as follows:

$$e_G = \lambda \times (t_{rd} + t_{wr}) + t_{er}. \tag{6}$$

In Eq. (6), λ denotes the number of valid pages in the victim block. Since λ is unpredictable in each victim block, the execution time of each garbage collection task is variable. Figure 1 shows an example of nondeterministic garbage collection. In Figure 1(a), suppose there is a victim block with four valid pages. Then, $\lambda = 4$ and all valid pages in the victim block are rewritten to another free block. Figure 1(b) and Figure 1(c) show the different system response time due to the nondeterministic value of λ. Suppose t_{wr}, t_{rd}, and t_{er} are $220\mu s$, $29\mu s$, and $2000\mu s$, respectively. When executing write task T_{w2}, the garbage collector triggers a garbage collection task G and the victim block has 12 valid pages ($\lambda = 12$). Thus, to reclaim such a block takes $12\times(29+220)+2000 = 4988\mu s$. Since T_{w2} needs to wait for the completion of G, the system response time for T_{w2} is $\mathcal{R} = 220 + 4988 = 5208\mu s$. In Figure 1(c), if $\lambda = 2$ in the victim block, the execution time of the garbage collection task is $2 \times (29 + 220) + 2000 = 2498\mu s$ and the system response time is only $\mathcal{R} = 2718\mu s$. Therefore λ causes unpredictable execution time of the garbage collection.

2.3. Problem Statement

Based on the task models and concepts, we further clarify the problem as follows.

Given a data request task set $V_T = \{T_1, T_2, T_3, \ldots, T_n\}$ and the garbage collection task set $V_G = \{G_1, G_2, \ldots, G_m\}$ on system demand, we want to obtain a task schedule in which each data request can be executed within the upper bound $\mathcal{U}(\mathcal{R}_T)$ in the worst case, and the average system response time can be guaranteed with low space cost.

3. LAZY-RTGC: A REAL-TIME LAZY GARBAGE COLLECTION MECHANISM

In this section, we present our real-time lazy garbage collection mechanism which jointly optimizes the worst and average system response time. The system architecture is presented in Section 3.1. The task scheduling scheme for Lazy-RTGC is introduced in Section 3.2 and the overhead analysis is presented in Section 3.4.

3.1. Page-Level Address Mappings in Lazy-RTGC

In the proposed Lazy-RTGC, the physical flash space can be partitioned into three areas: valid data area, invalid data area, and free area. The valid data area stores the latest data. When a data page is updated and the latest data is rewritten to another free page, the obsolete data belongs to the invalid data area. The free area contains free pages that can be utilized to store the updated data. Since Lazy-RTGC adopts a page-level address mapping scheme, these three areas are not predefined to specified physical blocks or pages. Therefore our scheme can maximize the space utilization. In NAND flash memory storage systems, the logical address space is normally smaller than the actual physical address space in the raw flash memory chip. This is because some physical blocks are utilized to manipulate the management of physical spaces in flash memory (e.g., translation blocks that store the address mapping tables, reserved blocks for bad block management, etc.). By taking advantage of these physical blocks, the size of the logical address space can help determine the lower bound of reclaimed free pages in the garbage collection.

In Lazy-RTGC, the page-level address mapping table is maintained in the RAM. Page-level address mapping can provide high performance of address translation and can effectively manage the flash space. Since there is no fixed physical flash space, as the write buffer and the trigger condition are independent from the logical address of the coming requests, any free space in the flash can be considered as the write buffer. Therefore Lazy-RTGC can delay the scheduling of garbage collection tasks as late as possible. That is, Lazy-RTGC will schedule garbage collection tasks only when the flash memory is about to run out of space. As the garbage collection process is the most time-consuming operation in flash memory management, postponing the garbage collection can significantly reduce the overhead that may impact each single data request. This could significantly improve the average system response time. Based on this observation, Lazy-RTGC adopts the page-level mapping scheme to achieve a real-time performance and further provides good average system performance in NAND flash memory storage systems.

3.2. Task Scheduling of Partial Garbage Collection Tasks

In our models, there are two kinds of tasks: the data request task (T) and garbage collection task (G). The flash memory controller serves the data request task to execute the atomic read or write operation. The garbage collector generates partial garbage collection tasks when the number of free pages in the flash memory is below the predefined threshold. If there is no garbage collection task, we schedule the data request tasks sequentially by coming order of the requests. Otherwise, the garbage collector

first selects a victim block and then generates several partial nonperiodic garbage collection tasks for reclaiming the victim block. Finally, the generated partial garbage collection tasks are scheduled to execute behind each write task one at a time until their executions are completed.

In order to hide the unpredictable service time of garbage collection, the partition of the garbage collection task is dependent on the entire garbage collection process of one victim block. We define a minimal deadline for each garbage collection task. Each G_j is executed within the deadline so that we can provide an upper bound of the worst system response time. Since the number of valid pages in the victim block is variable, one garbage collection task in previous schemes should be divided into several partial garbage collection tasks. Each partial garbage collection task executes many atomic operations, such as valid page copies or block erase, until the total execution time reaches the minimal deadline. Moreover, the block-erase operation, as the longest time cost operation, cannot be interrupted in the partial garbage collection task so that its execution time is the minimal deadline of each G_j. Therefore, the execution time of each garbage collection is guaranteed so that the worst system response time can be guaranteed as the minimal upper bound.

In Eq. (7), we define α as the number of valid copies in each partial garbage collection task, which is a constant specified to the flash storage systems.

$$\alpha = \left\lceil \frac{t_{er}}{t_{rd} + t_{wr}} \right\rceil. \tag{7}$$

ALGORITHM 1: Task Scheduling of Partial Garbage Collection Tasks

Input: A set of data request task ($V_T = \{T_1, T_2, \ldots T_n\}$), an empty set of garbage collection tasks ($V_G = \emptyset$), garbage collection threshold (ρ_{th}), and the number of free pages (Φ).
Output: A task schedule.

```
1  for each T_i ∈ V_T is executed do
2      if T_i ∈ T_w then
3          if V_G ≠ ∅ then
4              r_{G_j} ← r_{T_i} + e_{T_i};
5              remove_gc_task(G_j);
6          end
7          if Φ < ρ_th then
8              PBN_victim ← get_victim_block() ;
9              V_G ← generate_gc_tasks(PBN_victim) ;
10             r_{G_j} ← r_{T_i} + e_{T_i} ;
11             remove_gc_task(G_j) ;
12         end
13     end
14 end
```

Lazy-RTGC will not schedule garbage collection tasks after the execution of read tasks, since read tasks do not consume free pages. Thus Lazy-RTGC will not affect the read performance.

The task scheduling of partial garbage collection tasks is presented in Algorithm 1. The inputs of the algorithm contain a set of data request tasks, a set of garbage collection tasks, the threshold of garbage collection, and the number of free pages in the flash. The output is a task schedule of the garbage collection tasks, each with guaranteed execution time. Algorithm 1 handles each data request task from the data request set one by one and removes the task from the set after it is executed. In line 2, the type of data request task is identified to decide the garbage collection task schedule.

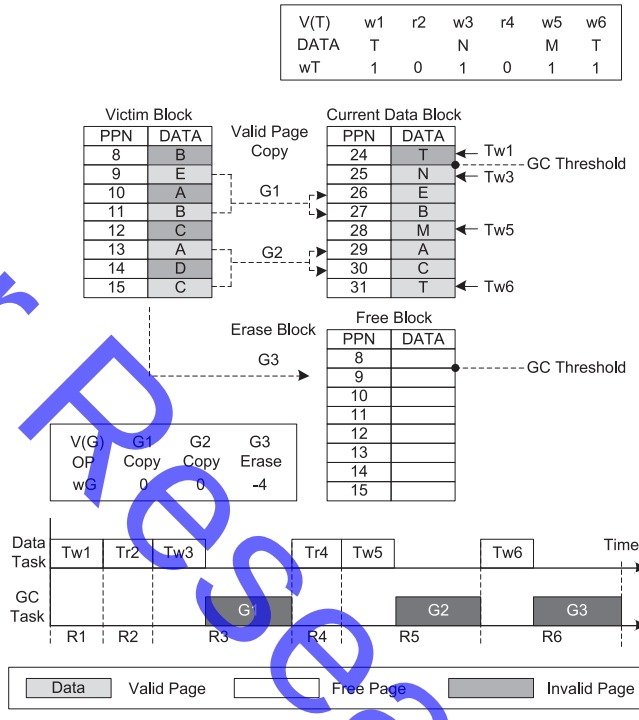


Fig. 2. An example task scheduling in lazy-RTGC.

As shown in lines 2 to 6 of Algorithm 1, if the current task is a write request and the garbage collection task set is not empty, it schedules one garbage collection task from the set and removes it once executed. The response time of the garbage collection task is the total time of the execution time of the write data request and the upper bound of the garbage collection task execution time (i.e., the erase block time). Otherwise, as shown in lines 7 to 12, if the garbage collection task set is empty and the number of free pages is lower than the threshold, our scheme will pick up one victim block and generate garbage collection tasks from the victim block. These generated garbage collection tasks are pushed into the set, where the valid page copy tasks are sorted by their first copied valid page number and the erase task is the last task. Then, our scheme selects the first garbage collection task from the set and schedules it behind the data request task. In Algorithm 1, our scheme schedules each data request task from the V_T and schedules the garbage collection task from V_G according to the type of data request task and the space usage of flash space. Suppose there are N data request tasks in the set of V_T, the algorithm will schedule all tasks one by one. Therefore the complexity of Algorithm 1 is $\mathcal{O}(N)$.

In order to avoid flash in the long-term worst case, Lazy-RTGC can be optimized by making use of system idle time. That is, Lazy-RTGC schedules the partial garbage collection task in system idle time, even though the flash does not meet the worst case. To reduce the impacts to average performance, we select that victim block with no valid pages and only schedule the partial garbage collection task after write tasks. As a result, Lazy-RTGC rarely meets the worst case and the system performance can be further improved.

Figure 2 shows an example task schedule generated from Lazy-RTGC. Suppose there is a set of data request tasks $V_T = \{T_{w1}, T_{r2}, T_{w3}, T_{r4}, T_{w5}, T_{w6}\}$, where T_{wi} represents

the write task and T_{ri} denotes the read task. When the flash storage system serves the request task T_{w3}, the garbage collector will invoke garbage collection as the number of free pages is smaller than the predefined threshold. Suppose $\alpha = 2$, which means each garbage collection task can execute at most two atomic copies. Since the selected victim block has 4 valid pages, the garbage collector generates 3 garbage collection tasks, G_0 and G_1 for valid page copies and G_2 for victim block erase. The garbage collector can reclaim free pages only after finishing the erase operation. Therefore $w_{G_1} = 0$, $w_{G_2} = 0$, and w_{G_2} is equal to $-(8-4) = -4$ (a negative value represents the reclaimed pages). G_0 is released after task T_{w3}. Since the deadline of G_1 is t_{er}, the T_{w3} can give response to the file system within $\mathcal{R} = t_{wr} + t_{er}$, which is the minimal upper bound of the worst system response time in a flash storage system. Task G_2 is scheduled after T_{w5}, which executes the same way as T_{w3}. After the completion of task G_3, the victim block is erased and becomes a free block that can be used to handle the coming tasks.

3.3. System Performance

In this section, we analyze the system performance of our scheme and compare it with representative real-time schemes. Given that the worst case does not happen frequently, the average system response time is becoming another important metric. The previous work in the literature mainly focuses on providing an upper bound of service time in flash storage systems, but ignores the average system response time. Therefore these real-time schemes suffer significant performance degradation even though they can guarantee the worst system performance.

Worst System Performance. The system response time in the worst case consists of the upper bound of a data request task and the deadline of a garbage collection task by using partial garbage collection. The free-space replenishment strategy in FSR [Chang et al. 2004] cannot guarantee the worst performance due to missing dependent real-time task information. GFTL, RFTL, and Lazy-RTGC can guarantee the worst-case response time by adopting a partial garbage collection technique. Due to the address mapping scheme, the upper bounds of the three schemes are different. GFTL [Choudhuri and Givargis 2008] uses a block-level mapping scheme, where the logical page number is written into the OOB area. There are extra OOB (*out of band*) read operations within the process of handling data request tasks so this impacts the upper bound of the worst-case response time. RFTL [Qin et al. 2012] uses a hybrid-level mapping scheme and the mapping table is partially stored in the OOB area so there are some OOB operations during address translation. Since our scheme adopts a page-level mapping scheme whose mapping table is maintained in the RAM, there are no extra OOB operations compared with GFTL and RFTL. Therefore Lazy-RTGC can provide the minimal upper bound of worst system response time.

Average System Performance. Garbage collection incurs the largest overhead in NAND flash memory storage systems due to the valid page copies and block erasing. Our scheme does not need to specify certain flash space as the write buffer. That is, any free space can be used as the write buffer due to the adoption of a page-level mapping scheme. The threshold of garbage collection is only related to the rest of the physical free space. The garbage-collection-triggered time is postponed as late as possible in Lazy-RTGC, which maintains high average performance.

Compared to our scheme, GFTL cannot provide good average performance. The main difference between GFTL and Lazy-RTGC is that our scheme can not only guarantee the worst-case response time, but also provide good average system performance. GFTL predefines a number of physical blocks as the write buffer and maintains a central garbage collection queue to decide which logical block is used for garbage collection. Once the primary block is full, the data should be written to the write buffer and invokes partial garbage collection for the primary block. The early garbage collection will cause

Table I. Service Guarantee Bounds of Ideal Case [Ban 1995], FSR [Chang et al. 2004], GFTL [Choudhuri and Givargis 2008], RFTL [Qin et al. 2012], and Lazy-RTGC

Bounds	$\mathcal{U}(e_r)$	$\mathcal{U}(e_w)$	$\mathcal{U}(\mathcal{R}_T)$	$\mathcal{U}(\lambda)$	$\mathcal{U}(\sigma)$
Ideal	t_{rdpg}	t_{wrpg}	t_{er}	π	0.99
FSR	t_{rdpg}	t_{wrpg}	$\mathcal{U}(e_T)+\mathcal{U}(e_G)$	$\sigma \times \pi$	N/A
GFTL	$t_{rdpg}+\pi t_{rdoob}$	t_{wrpg}	$t_{er}+max\{U(e_r),U(e_w)\}$	π	$1-[(\kappa+1)]/2\pi$
RFTL	$t_{rdpg}+t_{rdoob}$	$t_{wrpg}+t_{rdoob}$	$max\{U(e_r),t_{er}+U(e_w)\}$	π	1/3
Lazy-RTGC	t_{rdpg}	t_{wrpg}	$max\{U(e_r),t_{er}+U(e_w)\}$	$\sigma \times \pi$	$[(\pi-1)\alpha]/[(\alpha+1)\pi]$

lots of block erasing and valid page copies. As a result, GFTL suffers significant average system performance degradation.

RFTL pre-allocates three physical blocks to one logical block so that the execution of partial garbage collection is only related to the logical block. That is, once the primary physical block of the corresponding logical block is full, even if there exists free space in many physical blocks belonging to other logical blocks, GFTL and RFTL all trigger garbage collection. Therefore the garbage collection in GFTL and RFTL is invoked very early and the space utilization may be very low under unbalance workloads. As a result, average system performance is degraded and the high number of block-erase counts indirectly impacts the endurance of the flash memory. As average performance and the space utilization are also important since the worst case does not frequently happen, our scheme can not only provide an upper bound of execution time for each data request, but also provide better average performance and endurance compared to previous real-time flash schemes.

Table I shows the service guarantee bounds in different schemes. The symbols t_{rdpg}, t_{wrpg}, and t_{rdoob} denote the execution of page reading, page writing, and OOB reading time, respectively. σ is the ratio between logical and physical address space which used in the overprovisioning strategy. Through configuring σ, the reclaimed free space after each garbage collection is bounded. The upper bound of σ (denoted as $\mathcal{U}(\sigma)$) shows the maximum space utilization. For comparison purposes, we present a hypothetical ideal case as the baseline, where a read or write request task can be executed directly without triggering any garbage collection. Since the erase operation is the longest atomic operation in the flash and cannot be interrupted, the $\mathcal{U}(\mathcal{R}_T)$ in the ideal case is t_{er}. FSR is a representative scheme of a free-space replenishment strategy which can provide an upper bound of valid pages in the victim block (denoted as $\mathcal{U}(\lambda)$). However, FSR cannot provide the worst system response time and the upper bound of σ due to missing real-time task information so that its $\mathcal{U}(\mathcal{R}_T)$ is the theoretical worst-case value given in Eq. (5). GFTL schedules garbage collection tasks after the execution of a read or write task so that it impacts the read performance. The $\mathcal{U}(\sigma)$ in GFTL is $1-[(\kappa+1)]/2\pi$, where κ is the number of steps in partial garbage collection. Since GFTL cannot guarantee the valid pages in a victim block, in the worst case, $\kappa = [(\pi-1)t_{rdpg}+\pi t_{rdoob}+\pi t_{wrpg}]/t_{er}+1$. RFTL and our scheme only schedule garbage collection tasks after the completion of write tasks, so there is no read performance degradation. The $\mathcal{U}(\sigma)$ in RFTL is only 1/3 due to fixed block pre-allocation and that in Lazy-RTGC depends on the specification of flash. In the next section, we will present and analyze some properties and task schedulability to prove the bounds on time and space requirements.

3.4. Overhead Analysis

The resource overhead in Lazy-RTGC mainly comes from the RAM footprint and flash space. Due to the big page-level mapping table maintained in the RAM, Lazy-RTGC has large RAM-space consumption. For 1GB flash space, it requires 2MB RAM space to

store the mapping table. To solve this problem, several on-demand approaches [Gupta et al. 2009; Qin et al. 2011; Zhang et al. 2013] have been proposed. They can provide the page-level mapping performance but only cost RAM space similar to that in block-level mapping schemes. In order to guarantee the number of reclaimed free pages after each garbage collection, the logical address space is configured smaller than the entire physical flash space. Therefore Lazy-RTGC has flash space overhead. The space utilization only depends on the specification of the flash. For mainstream SLC NAND flash memory, Lazy-RTGC can achieve 87.5% space utilization. By adopting a page-level mapping scheme and partial garbage collection, the CPU resource consumption from address translation is close to that in the page-level mapping table. There are no further computing resource requirements in partial garbage collection, since it only defines the partial task start point and finish point. Therefore the CPU resource consumption is similar to the pure-page-level mapping scheme.

4. SCHEDULABILITY ANALYSIS

4.1. Bound of the Worst System Response Time

In this section, we analyze the bounds of the worst system response time of each data request task. In our task model, the entire process of garbage collection is divided into several partial garbage collection tasks, and each task G has the same deadline which is equal to the longest execution time of the atomic operations in the flash. We use λ to represent the number of valid pages in the victim block and use $\mathcal{N}(V_G)$ to denote the total number of generated garbage collection tasks. Then we can define $\mathcal{N}(V_G)$ as follows.

$$\mathcal{N}(V_G) = \left\lceil \frac{\lambda}{\alpha} \right\rceil + 1. \tag{8}$$

Based on Eq. (8), we can get some properties of partial garbage collection tasks.

Property 4.1. Since the erase operation is the longest atomic operation in NAND flash memory storage systems, the deadline of each garbage collection task (d_G) is equal to t_{er}.

Property 4.2. If $\lambda = 0$, the number of generated garbage collection tasks is equal to 1, which is the minimal number. That is, $\mathcal{N}(V_G) = 1$. For the worst case, $\lambda = \pi - 1$, where the victim block has the maximal number of valid pages, the number of generated garbage collection tasks also reaches the maximal value according to Eq. (8).

From the preceding properties, we propose Lemma 4.1 to analyze partial garbage collection tasks.

LEMMA 4.1. *The garbage collector can generate a finite number of partial garbage collection tasks from any garbage collection. The size of the garbage collection task set* ($\mathcal{N}(V_G)$) *has an upper bound which is only related to the specification of the NAND flash storage system.*

PROOF. Since each block has a fixed number of pages in the flash memory, the number of valid pages in the victim block has an upper bound of $\pi - 1$ (i.e., at least one invalid page). Moreover, Property 4.1 shows that t_{er} is the longest atomic operation execution time. t_{er} is the minimal value of deadline for each G_j. Since the garbage collector doesn't generate more partial garbage collection tasks until all previous tasks are scheduled, the upper bound of $\mathcal{N}(V_G)$ can be obtained, which is only related to π and α. Therefore

the upper bound of $\mathcal{N}(V_G)$ is only affected by the specification of the NAND flash storage system. □

Based on the properties and lemma of partial garbage collection tasks, we further propose a theorem to analyze the schedulability of data request tasks and partial garbage collection tasks in terms of the worst system response time.

THEOREM 4.2. *For a data request task (T_i), in the worst case, T_{wi} is a write task and V_G is not empty. Lazy-RTGC can schedule T_i and G_j so that $\mathcal{R}_{T_{wi}}$ can be guaranteed under a minimal upper bound.*

PROOF. Since T_i and G_j have dependency, the worst system response time is the period between the start time of T_i and the finishing time of G_j. According to Lemma 4.1, each G_j has a minimal deadline and the execution time of a write operation is the upper bound of e_{T_i}. Moreover, only one G_j in the V_G is scheduled to be executed behind T_i at one time. The upper bound of the system response time is bounded by e_{T_w} and t_{er}. Therefore we can schedule them within the minimal upper bound of the system response time in the worst case. □

4.2. Bound of the Reclaimed Free Space

In our task models of the NAND flash storage system, the schedulability of data request tasks and garbage collection tasks is not only related to the guaranteed system response time, but also to the free space in the flash. That is, if there is no free space to allow the execution of write operations or atomic copies, tasks cannot be scheduled and executed successfully. Therefore, in this section, the page requirement (w) in the task is discussed and we give the bounds of the space configuration to promise the tasks' schedulability on the space level.

Since the entire garbage collection task is divided into several partial garbage collection tasks and each task is executed behind one write task at one time, our scheme in fact delays the reclaiming time. On the other hand, by making use of a page-level mapping scheme, our scheme can fully use each free page in the flash space, that is, each free block can handle data from both the atomic copies in the garbage collection tasks and the write data request tasks. Therefore our scheme can improve the space utilization.

Here we summarize the property of the space requirement as follows.

Property 4.3. If $\mathcal{N}(V_G) = k$, the total free pages' cost on the dependent write tasks is $\sum_{i=1}^{k} w_{T_i}$, while the reclaimed free space after k partial garbage collection tasks is $\pi - \lambda$.

From Property 4.3, we can observe the space dependency between data request tasks and garbage collection tasks. That is, garbage collection tasks depend on data request tasks to be scheduled, while at the same time data request tasks depend on the reclaimed free space from the garbage collection tasks to be executed. Therefore it can be considered as a producer and consumer problem, where the garbage collection task is a producer to reclaim the free space while the data request task is a consumer to cost the reclaimed free space. When there is no free space in the flash, the data request task cannot be executed so that the corresponding garbage collection task cannot be scheduled either. Moreover, there is no reclaimed free space due to the failed scheduling of the garbage collection task. As a result, in this situation, the flash memory never has free space and no write task nor garbage collection task can be scheduled. In order to protect our task scheduling from such deadlocks, we need to promise the following

equations.

$$\sum_{i=1}^{k} w_{T_{wi}} \leq \sum_{j=1}^{k} (-w_{G_j}) \tag{9}$$

$$k \leq (\pi - \lambda)$$

Since w_{T_w} is equal to 1 for each write task, $\sum_{i=1}^{k} w_{T_{wi}}$ is equal to k and has an upper bound. $\sum_{j=1}^{k}(-w_{G_j})$ is the number of reclaimed free pages of one victim block, which is equal to $(\pi - \lambda)$. However, due to the unpredictable λ in each victim block, flash memory cannot give a bound of reclaimed free space. In our scheme, we adopt the strategy used in Chang et al. [2004] which limits the logical space address to guarantee the number of valid pages in each block. Therefore we propose the following theorem to analyze the guaranteed number of valid pages.

THEOREM 4.3. *Suppose the total number of data pages is denoted as Θ and the number of logical pages is bounded by Λ. If the garbage collector adopts a greedy policy to select a victim block, the number of valid pages in the victim block can be guaranteed to an upper bound (denoted as $\mathcal{U}(\lambda)$). That is, $\mathcal{U}(\lambda) = \lceil \frac{\Lambda}{\Theta} \times \pi \rceil$.*

PROOF. The garbage collector adopts a greedy policy to select that victim block with the least number of valid pages. Assume a victim block is picked out with $\lambda' = \mathcal{U}(\lambda) + 1$ valid pages and the flash space is fully used. Thus other blocks have at least λ' valid pages. Suppose there are N data blocks and $\Theta = \pi \times N$, $\Lambda = \lambda' N = \frac{\Lambda \times \pi \times N}{\Theta} + N = \Lambda + N$. Obviously, the number of logical pages contradicts the assumption in the theorem. Therefore $\mathcal{U}(\lambda)$ is the upper bound of the number of valid pages in the victim block. □

Since Theorem 4.3 gives an upper bound of valid pages in each victim block, the value of $(\pi - \lambda)$ can also provide a lower bound of the number of invalid pages. Therefore, we further propose the following theorem to analyze the schedulability of data request tasks and garbage collection tasks in terms of space requirements.

THEOREM 4.4. *The garbage collection tasks can be scheduled to execute after write tasks when and only when the lower bound of reclaimed space is greater than or equal to the upper bound of space requirement of dependent write tasks. That is, after scheduling partial garbage collection tasks in the V_G, the flash memory has enough reclaimed free space to schedule newly generated garbage collection tasks in the future.*

PROOF. According to Eq. (9), k is the free page cost of the write tasks with the scheduled garbage collection tasks. In Lemma 4.1, we show that $\mathcal{N}(V_G)$ has an upper bound. On the other hand, we prove the number of reclaimed free pages in each victim block has a lower bound in our scheme. If the upper bound of $\mathcal{N}(V_G)$ is always lower than the lower bound of the reclaimed free space, we can promise that there always exists enough space for scheduling data request tasks with garbage collection tasks. □

Since we limit the logical address space lower than the total flash space, our scheme also has a trade-off of flash space compared with previous works. In order to reduce such space overheads as much as possible, we set k to be equal to the lower bound of the reclaimed space. Then, we have the following equation by combining Eqs. (6) and (9).

$$\lambda \leq \left\lceil \frac{(\pi - 1)\alpha}{\alpha + 1} \right\rceil. \tag{10}$$

In order to simplify the representation, we use σ to denote the ratio of logical address space to total physical space (i.e., physical-space utilization). Therefore the upper bound

of λ analyzed in Theorem 4.3 can be represented as $\lceil \sigma \times \pi \rceil$. Further, we can get the following equation of the space configuration parameter of σ in our models.

$$\sigma \leq \frac{(\pi - 1)\alpha}{(\alpha + 1)\pi}. \tag{11}$$

Therefore, we can see that σ is only related to the specification parameters (α and π) of the flash memory. Moreover, we can get the relation between k and σ by combining Eqs. (10) and (11).

$$k \leq \max\left\{\left\lceil \frac{\sigma\pi}{\alpha + 1} \right\rceil, \lceil (1 - \sigma)\pi \rceil\right\}. \tag{12}$$

In order to delay scheduling garbage collection tasks as much as possible, we define the minimal threshold of starting to generate and schedule the partial garbage collection task compared with previous schemes. We give the following equation to define the garbage collection threshold (denoted as ρ_{th}). That is, we can execute write tasks without scheduled partial garbage collection tasks until the free space is under the ρ_{th}. Here $\mathcal{U}(k)$ and $\mathcal{U}(\lambda)$ represent the upper bound of free page costs and valid page copies during the garbage collection tasks, respectively.

$$\rho_{th} = \mathcal{U}(k) + \mathcal{U}(\lambda). \tag{13}$$

4.3. Space Configuration

Since the space configuration of Lazy-RTGC is only related to α and π, we select six representative flash memory chips to show the relationship between the space utilization and the specification. Table II shows the parameters of different NAND flash chips, and Table III presents the space utilization for different schemes using the parameters in Table II. NAND flash design can be categorized into SLC (*single-level cell*), MLC (*multilevel cell*), and TLC (*triple-level cell*) flash memory. SLC flash stores one bit value per cell, which can provide faster write performance and greater reliability. An MLC and a TLC cell can represent multiple values so as to provide high storage capacity with performance and reliability degradation. Lazy-RTGC can achieve about 87% space utilization in SLC NAND flash and meet the worst space utilization (about 49.6%) in Samsung 512MB MLC NAND flash. The space utilization is decided by the ratio between the sum of t_{rd} and t_{wr} and the t_{er}. Since the ratio is only 1 in Samsung MLC NAND flash, that is, each partial task can copy only one data page, Lazy-RTGC reaches the worst space utilization, which is about half of the entire flash. GFTL has better space utilization compared to our scheme, but suffers performance degradation. Since RFTL pre-allocates three physical blocks to one logical block, its space utilization is about 33.3%, not related to specification of flash. The space utilization of FSR is decided by the real-time task information so that it does not have a fixed upper bound. The ideal scheme has the highest space utilization since it uses the Pure-Page-Level mapping scheme without considering real-time properties. TLC NAND flash memory has high storage capacity but its performance is poor. GFTL and our scheme have low space utilization on a TLC NAND flash memory specification. From the theoretical value comparison, Lazy-RTGC shows better space utilization in SLC NAND flash than those in MLC and TLC NAND flash. Moreover, SLC NAND flash has good reliability and endurance so is more suitable to real-time embedded systems. Therefore Lazy-RTGC can be applied on SLC NAND flash for real-time systems.

5. EXTENSION TO ON-DEMAND ADDRESS MAPPING

Page-level address mapping in Lazy-RTGC can improve the average system performance by postponing garbage collection operations as late as possible. However, the big address mapping table costs large RAM space, which is not suitable for

Table II. Space Configurations in Different Flash Memory Chips

NAND Flash	$t_{rd}(\mu s)$	$t_{wr}(\mu s)$	$t_{er}(\mu s)$	π	$\mathcal{U}(\sigma)$
Spansion 512MB SLC NAND Flash [Spansion 2013]	25	200	2000	64	0.875
Toshiba 512MB SLC NAND Flash [Toshiba 2012]	25	300	3000	64	0.886
Samsung 512MB MLC NAND Flash [Samsung 2007]	60	800	1500	128	0.496
Micron 16GB MLC NAND Flash [Micron 2012]	50	1600	5500	256	0.747
Toshiba 2GB TLC NAND Flash [Toshiba 2008]	250	2700	4000	192	0.497

Table III. Space Utilization Comparison

NAND Flash	Ideal	FSR	RFTL	GFTL	Lazy-RTGC
Spansion 512MB SLC NAND Flash [Spansion 2013]	99%	N/A	33.3%	92.9%	87.5%
Toshiba 512MB SLC NAND Flash [Toshiba 2012]				92.9%	88.6%
Samsung 512MB MLC NAND Flash [Samsung 2007]				68.8%	49.6%
Micron 16GB MLC NAND Flash [Micron 2012]				84.2%	74.7%
Toshiba 2GB TLC NAND Flash [Toshiba 2008]				59.6%	49.7%

resource-constrained embedded systems. In this section, we present how to apply our scheme to on-demand page-level mapping.

5.1. Lazy-RTGC for On-Demand Page-Level Address Mapping

To solve the big RAM cost in page-level mapping, some on-demand approaches have been proposed [Gupta et al. 2009; Qin et all . 2011; Zang et al. 2013]. We select DFTL [Gupta et al. 2009], a representative on-demand scheme, to introduce how to apply our Lazy-RTGC scheme to on-demand page-level mapping.

In DFTL, there are two kinds of blocks: data blocks and translation blocks. The entire page-level mapping table is stored in translation blocks. Each translation page stores multiple consecutive mapping items from a fixed starting logical address. Frequently used mapping items are cached in a *cached mapping table* (CMT) in the RAM and there is a *global translation directory* (GTD) to track the translation pages in flash. The performance of DFTL is close to those of pure-page-level schemes, while the RAM space it requires is close to those of block-level mapping schemes. Thus it can be applied to resource-constrained embedded systems. However, by introducing translation blocks and CMT, it is more difficult to jointly optimize its worst-case response time and average response time.

Compared with pure-page-level address mapping schemes, DFTL triggers extra translation page operations in NAND flash. In particular, in the worst case, one write request incurs two extra read operations and one extra write operation. The reason is as follows: First, one translation page needs to be read in order to get the corresponding address mapping of the write request if we could not find the mapping information from CMT; second, by caching the new address mapping information into CMT, it may cause an eviction operation that will introduce one read and one write operation in order to write the updated mapping item back to the translation page. Similarly, in the worst case, one read request also incurs two extra read operations and one write operation.

In order to jointly optimize the average- and worst-case performance of DFTL, we apply Lazy-RTGC to manage both the cached mapping table and translation blocks. To make our scheme easily extended to other on-demand page-level schemes, we do not modify the data structures of DFTL. Our scheme includes three tasks, namely the data-block partial garbage collection task (denoted as *DG*), the translation-block partial garbage collection task (denoted as *TG*), and the translation page writeback task (denoted as *TW*). Basically, *DG* manages partial garbage collection for data blocks, *TG* manages partial garbage collection for translation blocks, and *TW* writes several

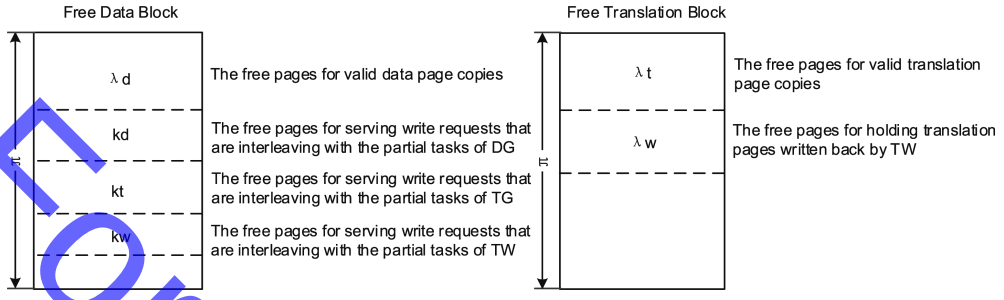


Fig. 3. An example of Lazy-RTGC for DFTL.

translation pages back to translation blocks by grouping all corresponding mapping items in CMT together so as to reduce the size of CMT. Our basic idea is to guarantee the following two conditions in the worst-case scenario.

(1) A predefined number of free pages in data blocks and a predefined number of free pages in translation blocks (both of the numbers are not larger than π) are good enough to hold all write requests during execution of the aforesaid three tasks.
(2) After the three tasks have been finished, one new free data block and one new free translation block are generated so there is always enough space for garbage collection, even in the worst-case scenario.

Figure 3 shows an example in which a data block and a translation block are used to provide free pages that can hold all write requests for data and translation pages, respectively, when the three tasks are executed.

Like in Lazy-RTGC, *DG*, *TG*, and *TW* are all executed in partial garbage collection manner in which each is divided into partial tasks that are scheduled to interleave with tasks that serve read/write requests (see Figure 4 for an example). In *TG*, Lazy-RTGC is applied in garbage collection for translation blocks, in which a garbage collection operation is divided into a partial task for copying valid translation pages and one for erasing the victim translation block. Moreover, the overprovisioning strategy is also applied in translation blocks. By configuring the space ratio of translation blocks, we can guarantee the maximum number of valid translation pages in a victim translation block so the number of partial tasks of *TG* can be bounded. In *DG*, to reduce extra update operations for translation pages, for both write requests and valid page copies, all address mappings are cached into CMT. Accordingly, *TW* is used to reduce the size of CMT by grouping related mapping items into their corresponding translation pages and writing back to translation blocks. Similar to partial garbage collection tasks, the upper bound of the execution time of *TW* is t_{er} and each *TW* task can update α translation pages. *DG*, *TG*, and *TW* are independently invoked based on their own thresholds. When all or any two of them are triggered at the same time, the precedence order is $DG > TG > TW$.

The worst-case scenario occurs when *DG*, *TG*, and *TW* all reach their thresholds at the same time. Based on the preceding precedence order, *DG* is first scheduled to be executed. Since all related address mapping information will be cached in CMT, *DG* will not introduce any updates for translation pages. Next, *TG* will be scheduled to be executed after *DG* has finished. In *TG*, as data blocks and translation blocks are separated, valid translation pages from a victim translation block will be copied to another translation block. Thus *TG* itself does not require data pages. However, free pages from data blocks are still needed to serve write requests during the execution of *TG*, because partial tasks of *TG* are interleaved with tasks to serve read/write

requests. Finally, *TW* is scheduled to write address mapping items back to translation pages in a batch manner. The number of *TW* tasks is decided by the size of CMT and the thresholds of *DG* and *TG*, which is discussed in detail in space utilization in Section 5.2. Free pages required by the three tasks can be provided as shown in Figure 3. Here, *DG* requires λ_d data pages for valid page copies while k_d data pages are used to serve write requests that are interleaving with the partial tasks of *DG*. And *TG* requires λ_t translation pages for valid translation page copies in translation blocks, and k_t data pages provide the space for write requests interleaving with the partial tasks of *TG*. Then *TW* writes λ_w translation pages back while k_w data pages are used for write requests. As discussed in Section 5.2, based on the overprovisioning strategy, we can guarantee $k_t + k_d + \lambda_d + k_w \leq \pi$ and $\lambda_t + \lambda_w \leq \pi$, so our scheme can work in the worst-case scenario.

Figure 4 shows a specific example. Suppose each block has eight pages and each partial garbage collection task can copy at most three valid pages. When there is only one free data block, the garbage collection of the data block is triggered. DG_1 and DG_2 are scheduled after T_{w1} and T_{w2}, respectively, to copy the valid data pages and erase the block. During the garbage collection, the updated address mapping items from valid page copies and write requests are cached in CMT. After the data-block garbage collection, since CMT is close to full and the translation block has not reached the threshold, Lazy-RTGC generates a *TW* task (write back TP19 and TP45) that is executed after T_{w4} to write back the corresponding updated mappings to the current translation block. Then, after T_{w4} and *TW*, the number of free pages in the translation block is smaller than the threshold. This will trigger translation-block garbage collection. After executing T_{w5} to write to PPN30, we schedule TG_1 to copy three translation pages (TP11, TP18, and TP14) to the current translation block. Then, in TG_2, the victim translation block is erased after T_{w7} that serves a write request. As a result, one new free data block and one new free translation block are reclaimed with our scheme.

5.2. Bound Analysis

In this section, we present bound analysis for applying Lazy-RTGC on DFTL. We start with the worst-case response time, then discuss the average-case response time, and finally analyze space utilization.

Worst-Case Response Time. Lazy-RTGC for DFTL includes two kinds of garbage collection operations: data-block garbage collection and translation-block garbage collection. Our scheme only schedules one kind of garbage collection operation at a time. After a victim data block is erased, we make use of remaining free pages to schedule the writeback tasks and partial garbage collection tasks on translation blocks. Moreover, our scheme caches the updated mapping items in CMT during the garbage collection so that it does not incur translation page updates. In the worst case, there is a cache miss to handle a write request. Accordingly, there is an extra read translation page operation to locate the address mapping information. Thus the worst-case response time is as shown in Eq. (14).

$$\mathcal{U}(\mathcal{R}_T) = \max\{U(e_r), t_{er} + U(e_w + e_r)\}. \tag{14}$$

Average-Case Response Time. Since the entire mapping table is stored in the translation blocks, the operations on the mapping item between the cache and translation blocks incur extra overhead. For the worst case of DFTL, there are an extra two translation page read and one translation page write operations attaching to one data page write request due to cache replacement. The upper bound of each data task without triggering garbage collection is presented in Eq. (15).

$$\mathcal{U}(\mathcal{R}_{avg}) = 2 \times (t_{wr} + t_{rd}). \tag{15}$$

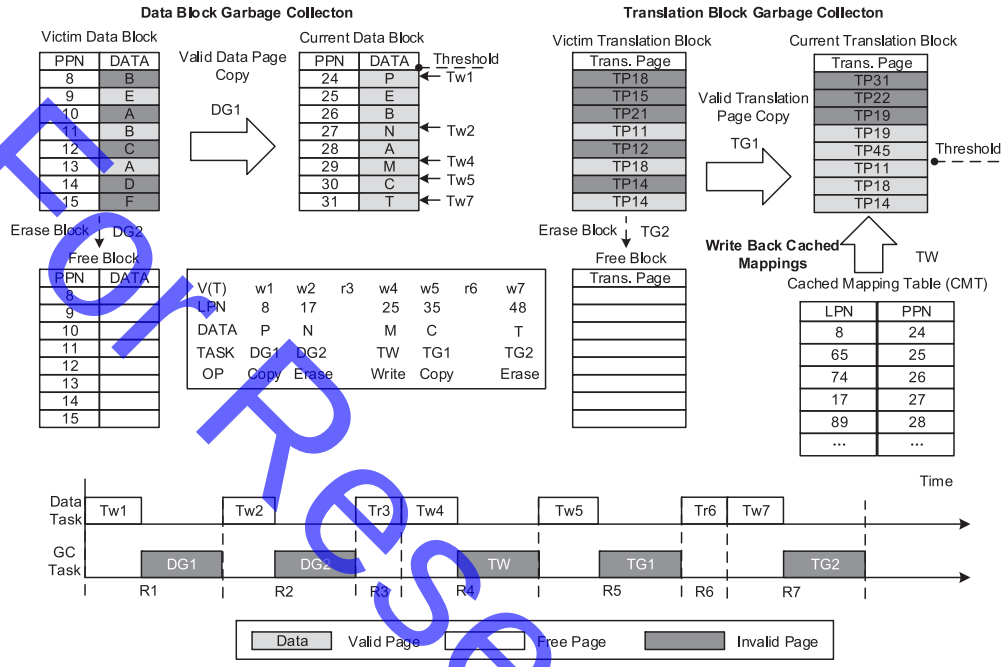


Fig. 4. An illustrative example to show how Lazy-RTGC for DFTL works.

Space utilization. To provide enough free space for scheduling two kinds of partial garbage collection operations, the overprovisioning strategy is applied for both data blocks and translation blocks. We define N_d as the number of physical data blocks and N_t as the number of translation blocks. Let σ_d be the ratio between the logical address space and N_d, where σ_t is represented as the ratio between N_t and predefined physical space for the translation block. σ_d decides the number of reclaimed data pages and σ_t is used for getting the upper bound of the valid translation page copies. λ_d and λ_t represent the guaranteed number of valid pages in the victim data block and translation block, respectively.

With demand-based approaches, translation blocks will occupy some flash space. If the size of each page is 2KB and the RAM cost for each page-level mapping item is 4 bytes, the physical space overhead is 1/512 (about 0.2%) of the entire space. The logical address space Λ_d is decided by $\lceil \sigma \times N_d \rceil$ and the corresponding number of translation blocks is $\lceil \Lambda_t = \Lambda_d/512 \rceil$. To guarantee the number of valid translation page copies, we also apply the overprovisioning strategy to translation blocks, that is, $N_t = \sigma_t \times \Lambda_t$. Here N is the total number of physical blocks in flash, $N = N_d + N_t$. To reclaim the free space for translation-block garbage collection, the value of k and the number of partial garbage collection tasks for the data block and translation block are defined as follows.

$$k_d + k_t \leq (\pi - \lambda)$$
$$\mathcal{N}(V_{G_d}) = \left\lceil \frac{\lambda_d}{\alpha} \right\rceil + 1; \mathcal{N}(V_{G_t}) = \left\lceil \frac{\lambda_t}{\alpha} \right\rceil + 1 \quad (16)$$

The garbage collection of translation blocks does not require free data pages but only translation pages, so we only need k_t extra free data pages to serve write requests that are interleaving with the partial tasks in a translation-block garbage collection task.

According to the prior equations, we can get the space configuration under on-demand page-level mapping in Eq. (17).

$$\sigma_d \leq \frac{(\pi - 2)\alpha - \lambda_t}{(\alpha + 1)\pi}$$
$$\lambda_t \leq \sigma_t \times \pi. \tag{17}$$

Here σ_t can be configured to a small value since the entire space for translation blocks is small. Then σ_d can be decided by the flash specification of α, π, and the configured σ_t. For example, in the experiments, the space utilization ratio of Lazy-RTGC for DFTL is about 80%.

On the other hand, we cache address mappings to CMT during the garbage collection. In the worst case, data write requests and valid data copies all cause cache misses so that π mapping items may be added to CMT. Each writeback task can update α translation pages and each translation-block garbage collection can update at least λ_t translation pages. Thus, the number of *TW* tasks is $(\pi - \lambda_d - k_d - k_t)$, which represents the number of free pages not scheduled for partial garbage collection in the last free data block. Then, the total number of updated translation pages (denoted as γ) from *TW* tasks and TG tasks can be calculated in the following equation:

$$\mathcal{L}(\gamma) = (\pi - 2)\alpha - (\alpha + 1)\sigma_d \times \pi. \tag{18}$$

In Eq. (18), σ_d is the space configuration parameter for overprovisioning, and α and π are only related to the flash specification. In the worst case, π cached mapping items are all from different translation pages. To balance the number of increased π mapping items and the γ writeback translation pages, each updated translation page should have at least π/γ mapping items from CMT. We define N_t as the number of all valid translation pages in the flash, which is decided by the logical address space. Then we can get an upper bound of the CMT size as follows:

$$L_{cmt} \leq \frac{\pi}{\gamma} \times N_t. \tag{19}$$

6. EVALUATION

In this section, we present the experimental setup and results with analysis to demonstrate the effectiveness of the proposed scheme. We compare Lazy-RTGC with the Pure-Page-Level [Ban 1995], FSR [Chang et al. 2004], GFTL [Choudhuri and Givargis 2008], and RFTL [Qin et al. 2012], techniques in terms of five performance metrics: system response time in the worst case, average system response time, valid page copies, block-erase counts, and the space utilization ratio.

6.1. Experimental Setup

The framework of our simulation platform, as shown in Figure 5, is based on Disksim [Bucy and Ganger 2003], a well-regarded disk-drive simulator. FlashSim [Kim et al. 2009], as a module of Disksim, is used to manage and supply basic operations of a flash memory chip. We adopt the FlashSim framework because it is a widely used simulation platform to evaluate the performance of FTL schemes. In the simulation framework, we implemented our scheme, FSR, GFTL, RFTL, and the Pure-Page-Level mapping scheme. We also applied our scheme on a demand-based page-level mapping scheme (called On-demand Lazy-RTGC) to reduce RAM cost. FSR is a representative scheme that can guarantee the reclaimed free space. However, it cannot satisfy real-time requirements. GFTL and RFTL are representative schemes adopting the partial garbage collection technique. The Pure-Page-Level scheme is the page-level address

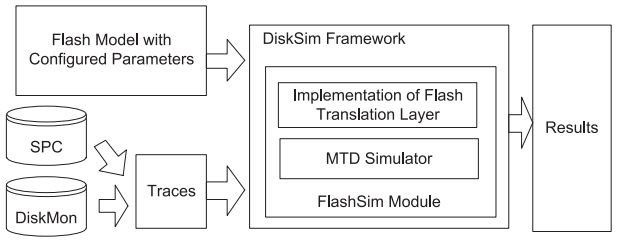


Fig. 5. The framework of simulation platform.

Table IV. Parameters of the NAND Flash Memory

Parameter	Value
Total capacity	32GB
The number of planes in the chip	8
The number of blocks per plane	2048
The number of pages per block	64
Page size	2KB
Page read latency	29.0μs
OOB read latency	29.0μs
Page write latency	220.9μs
Block erase latency	2000.0μs

Table V. Traces Used in the Simulation

Traces	Number of Requests	Write (%)	Average Request Size (KB)
Websearch	1,055,448	0.02	15.05
Financial	3,698,864	17.66	5.24
Copy File	670,412	71.89	42.30
Download File	1,730,415	67.21	41.10
Play Video	875,928	63.44	47.75

mapping scheme without applying any real-time mechanisms. In our simulation, a 32GB NAND flash memory is configured and the parameters are shown in Table IV.

We use a set of benchmarks from both real-world and synthetic traces to study the performance of different schemes. The traces used in our simulation are summarized in Table V.

Among them, Websearch [UMass 2013] is a read-dominant I/O trace obtained from Storage Performance Council (SPC), which has lots of read operations. Most of the read operations in Websearch are random data requests. Financial is an I/O trace with high sequential accesses from an OLTP [UMass 2013] application running at a financial institution. The logical address space in Financial is far smaller than the physical space of the simulated NAND flash. Copy File is a trace collected when copying files from one location to another. Copy File consists of a high ratio of write request tasks, including many sequential read and write operations. Download File is collected when downloading files from the network. It is also a write-dominant trace. Compared to the trace Copy File, it has more sequential write operations with a large number of requests. Play Video is collected under a hybrid workload of playing an online video in which the video player is reading the video data and, at the same time, downloading the data from the network. The trace Play Video contains many random read and write operations. These three traces are collected from a desktop running Diskmon [2013] with Windows XP on an NTFS file system. To make a fair comparison, performance

data is collected after the first garbage collection is triggered and there is a warm-up process that writes the data into the NAND flash before the simulation starts so that all read requests can read data from the simulator.

Compared to Pure-Page-Level, Lazy-RTGC adopts a different policy to trigger garbage collection. For Pure-Page-Level, the garbage collection can be delayed as late as possible. Since it does not use the overprovisioning strategy, the entire physical address space is mapped to the logical address space. Only one extra swap block is used as the buffer to hold the valid page copies during garbage collection. When there is no free data page in NAND flash memory, Pure-Page-Level triggers garbage collection operations to copy free pages to swap blocks and to reclaim free pages. FSR adopts the overprovisioning strategy where the logical address space is smaller than the physical space. The trigger condition in FSR is similar to that in Pure-Page-Level. GFTL and RFTL adopt a block-level and a hybrid-level mapping scheme, respectively. When the primary block for the corresponding logical block is full, garbage collection is triggered. Therefore, the trigger condition for these two schemes depends on the logical address of the coming data task. In Lazy-RTGC, by adopting a page-level mapping scheme, the garbage collection trigger time is delayed as late as possible compared to GFTL and RFTL. When there is only one free block in flash, partial garbage collection is triggered in our scheme. In all schemes, every garbage collection process reclaims one victim block when garbage collection is finished in the experiment.

6.2. Results and Discussion

In this section, we present the experimental results in terms of five performance metrics: system response time in the worst case, average system response time, valid page copies, block-erase counts, and the space utilization ratio. For each performance metric, we use six figures to represent the results from all five traces and the average results of these traces, respectively. We use *Pure-Page-Level*, *FSR*, *GFTL*, *RFTL*, *Lazy-RTGC*, and *On-demand+Lazy-RTGC* to represent the simulation results generated by the schemes in Ban [1995], Chang et al. [2004], Choudhuri and Givargis [2008], and Qin et al. [2012], the proposed Lazy-RTGC based on Pure-Page-Level, and Lazy-RTGC applied on an on-demand page-level mapping scheme, respectively.

(1) Worst-Case Response Time. The main objective of our scheme is to provide an upper bound of system response time. The experimental results are shown in Figure 6. The upper bound of worst-case response time in Pure-Page-Level is defined by Definition 5, that is, the garbage collection process needs to copy $\pi - 1$ valid pages. Moreover, due to the shortage of free pages, one data task may trigger multiple garbage collection operations. Therefore Pure-Page-Level may suffer bad worst-case response time. FSR can guarantee the number of reclaimed free pages but cannot guarantee the worst-case response time due to missing information of real-time tasks. Since our scheme adopts page-level address mapping and the mapping table is maintained in RAM, there is no OOB operation compared with GFTL and RFTL. Therefore our scheme can achieve the minimal upper bound of worst system response time. That is, $\mathcal{U}(\mathcal{R}_t) = t_{wrpg} + t_{er} = 2,220.9\mu s$. GFTL needs at most π OOB read operations to locate the page. In evaluation, GFTL reached this state in all traces so that the real upper bound is $\mathcal{U}(T) = t_{rd} + \pi t_{rdoob} + t_{er} = 3,885\mu s$. Therefore, our scheme can achieve a 42.83% reduction on the upper bound of worst system response time compared to GFTL. We can also achieve better performance (i.e., reduced one OOB read operation) compared to RFTL. As shown in Figure 6, our scheme can achieve 90.58% and 83.78% reductions on worst system response time compared with Pure-Page-Level and the FSR scheme, respectively.

In the on-demand Lazy-RTGC scheme, we cache the updated mapping items from data requests and valid page copies in the CMT so as not to incur any translation page

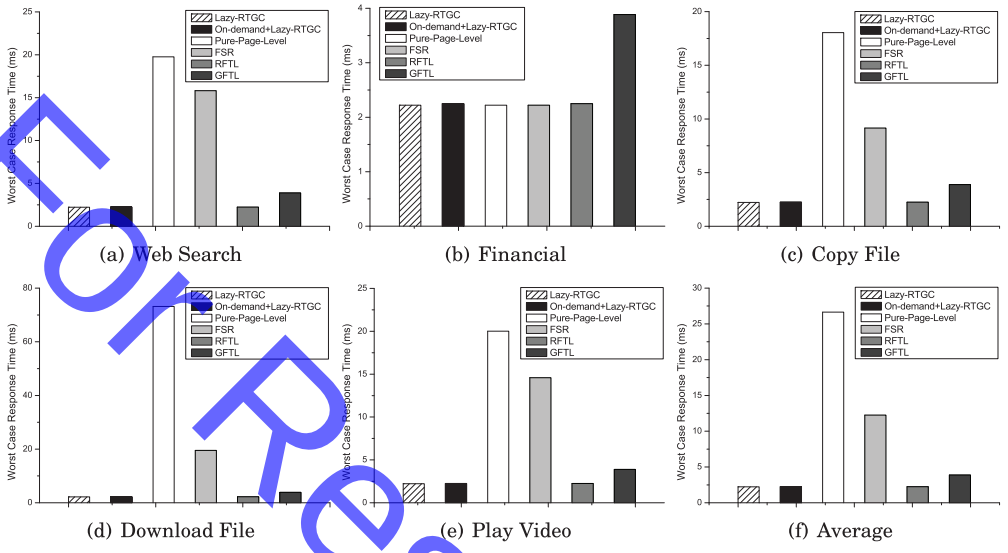


Fig. 6. Worst-case response time.

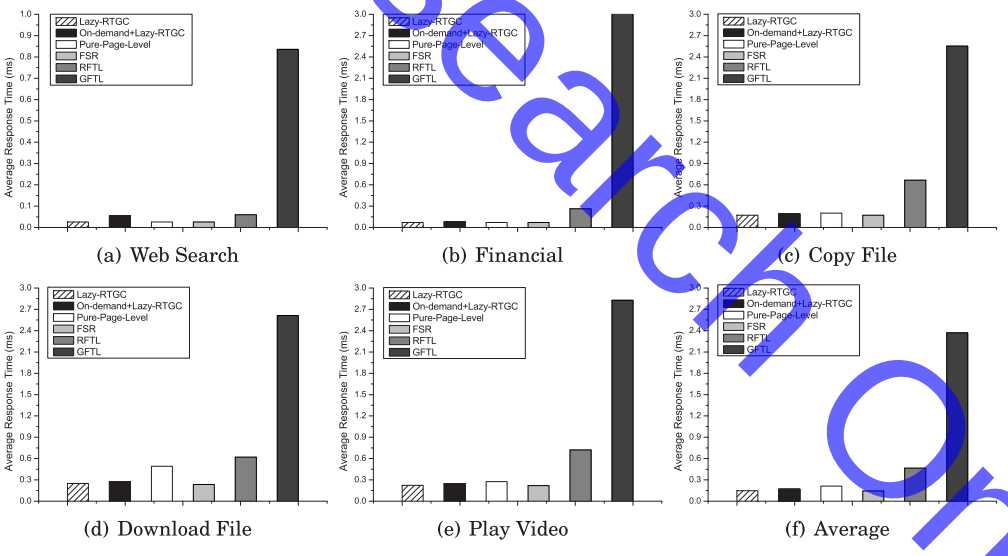
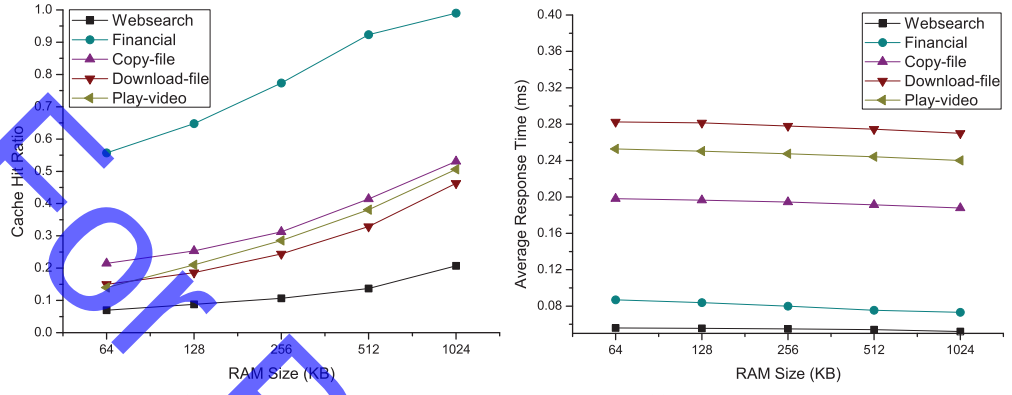


Fig. 7. Average response time.

operations during partial garbage collection. After garbage collection, we schedule writeback tasks to reduce the CMT size. For a single data request, it may need to read mapping information from the translation page in flash memory. Therefore the worst-case response time is slightly more than that in Lazy-RTGC.

(2) Average Response Time. Given that the worst case does not happen frequently, the average system response time is one of the most important metrics to represent system performance. The experimental results are shown in Figure 7. From the results, we can see that GFTL and RFTL suffer significant performance degradation compared with Pure-Page-Level and our scheme. Our scheme can achieve 94.56% and 50.84% improvements on average system response time compared with GFTL and RFTL,



(a) cache hit ratio (b) average response time

Fig. 8. The cache hit ratio and average response time for on-demand Lazy-RTGC with different RAM size configurations.

respectively. Since our scheme adopts a page-level address mapping scheme that can freely manage the data pages in the flash memory and sufficiently make use of the flash space, our scheme can achieve better average system response time. Compared to our scheme, GFTL adopts a block-level mapping scheme, and once some logical block is fully used, the corresponding physical block is added to the central garbage collection queue to do partial garbage collection. As a result, there is a large number of unnecessary garbage collections which are triggered very early. RFTL pre-allocates three physical blocks to each logical block. When the logical block is full, partial garbage collection is triggered within the allocated blocks. Therefore RFTL still triggers garbage collection early and requires lots of extra physical flash space. Compared with FSR and Pure-Page-Level that cannot guarantee real-time performance, our scheme can achieve similar average system performance.

In Lazy-RTGC, the logical address space is smaller than the physical. When garbage collection operations are triggered, the number of valid page copy operations can be guaranteed. Pure-Page-Level does not adopt the overprovisioning strategy, that is, the entire physical address space is mapped to logical address space. Only one block is used as the log buffer to hold valid page copies. Therefore, they may meet worst case when flash memory is fully utilized and few free pages can be reclaimed. As a result, the continuous garbage collection operations degrade the average performance in Pure-Page-Level. As shown in Figure 9, the average number of valid page copies during garbage collection in our scheme is smaller than that in Pure-Page-Level. Therefore more free pages are reclaimed after garbage collection, which can further postpone the next garbage collection triggering time. As a result, compared to Pure-Page-Level, the number of block-erase counts is also reduced in our scheme. As can be see from the results, our scheme has similar or even better average performance compared to Pure-Page-Level because of its smaller valid page copies and block-erase counts.

In On-demand Lazy-RTGC, as shown in Figure 8, we evaluate the average response time with different RAM cache sizes. Figure 8(a) presents the cache hit ratio with different RAM sizes over different benchmarks. Benchmark Web Search has lots of random read requests, so the hit ratio is below 20%. For benchmark Financial, the logical address space is small and contains a large number of sequential data requests. Therefore the hit ratio can achieve more than 50%. By making use of writeback tasks, for On-demand Lazy-RTGC, most cached mapping items can be written back to flash memory in a batch way. From the results in Figure 8(b), we see the average response

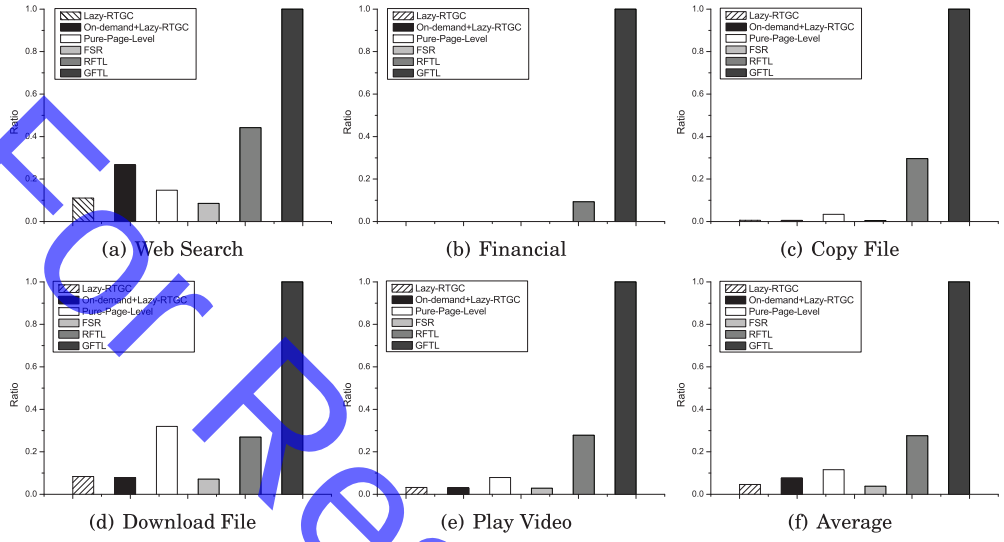


Fig. 9. The normalized number of valid page copies.

time with 1024KB RAM size over different benchmarks can achieve an average 7.50% improvement than that with 64KB RAM size. This is due to the fact that, with increase in RAM size, the number of translation page operations is reduced, and the worst-case response time is bounded. To make a fair comparison, for other performance metrics, we select 256KB as the CMT size in the experiments.

(3) Valid Page Copies. The number of valid page copies in garbage collection impacts the time consumption of the garbage collection process. By making use of a page-level address mapping scheme, Lazy-RTGC can fully use the free pages in the flash and trigger garbage collection as late as possible. Moreover, the logical address space is configured lower than the entire physical flash space. Thus there are more invalid pages in victim blocks when the flash memory is almost full. In order to reclaim enough free space to do both data garbage collection and translation-block garbage collection in On-demand Lazy-RTGC, the space utilization is lower than that in Lazy-RTGC. For Web Search, due to the cache replacement, 56.26% valid page copies are from translation pages. In other traces, about 3.98% valid page copies are translation page copies. By applying the overprovisioning strategy on translation blocks, there are few valid page copies in translation-block garbage collections. GFTL and RFTL adopt a block-level scheme so that the condition to trigger garbage collection depends on the logical address of the data request. That is, garbage collection is invoked when the allocated blocks are full, even though there are lots of free pages in the flash. Therefore GFTL and RFTL trigger garbage collection very early and there is a large amount of valid pages that need to be copied. To represent the results clearly, we normalized the experimental results and the results of GFTL are set to 1.

As shown in Figure 9, our scheme can achieve 95.36% and 86.11% reductions in valid page copies during the garbage collection compared to GFTL and RFTL, respectively. By adopting the overprovisioning strategy that limits the logical address space lower than the entire physical address space, there are more invalid pages in the victim block when running garbage collection compared to Pure-Page-Level. Moreover, as discussed in Section 4.2, the number of valid pages in a victim block has an upper bound while that for Pure-Page-Level is not predictable. Therefore our scheme can achieve a 60.51%

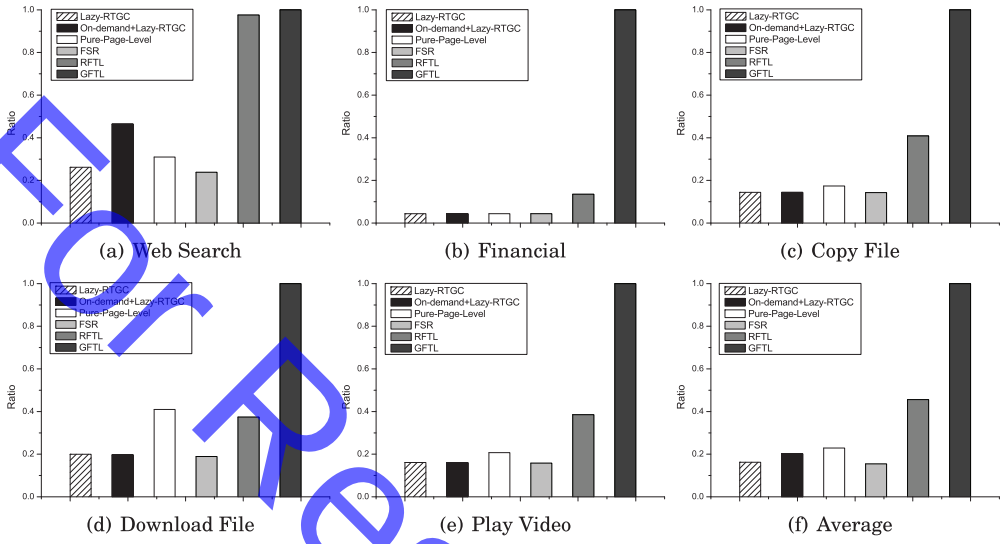


Fig. 10. The normalized number of block-erase counts.

reduction, on average, compared to Pure-Page-Level. Compared to FSR, our scheme has 21.56% more valid page copies since FSR has a lower space utilization ratio.

(4) Block-Erase Counts. The number of block-erase counts will influence the average system response time and the endurance of NAND flash memory. To show the results clearly, we normalized the experimental results and the results of GFTL are set to 1. As shown in Figure 10, our scheme can achieve 83.66% and 67.38% reductions in block-erase counts compared with GFTL and RFTL, respectively. That is because, for the 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. Thus these schemes will trigger lots of unnecessary garbage collection operations. Since our scheme, On-demand Lazy-RTGC, and FSR reduced the logical address space to guarantee the reclaimed free space, there are more reclaimed free pages after each garbage collection so that it can postpone the next garbage collection operation. As a result, our scheme has lesser number of block-erase counts compared with Pure-Page-Level and the number of block-erase counts is very close to the one in FSR. On-demand Lazy-RTGC contains two kinds of block, namely data block and translation block, and both of them will be erased due to shortage of free space. For Web Search, cache replacement incurs many translation-block garbage collections so the block-erase counts are increased 43.59% compared to Lazy-RTGC. In other traces, since the logical address space is reduced due to the overprovisioning strategy of translation blocks, the total number of block-erase counts in On-demand Lazy-RTGC is close to that in Lazy-RTGC.

(5) Space Utilization. In order to achieve the objective of guaranteeing the worst-case system response time, GFTL, RFTL, and our scheme all incur extra space overhead. In Lazy-RTGC, we can get the space utilization ratio σ according to Eq. (11). That is, the space utilization ratio of Lazy-RTGC is limited by $\mathcal{U}(\sigma) = \frac{(64-1)\times 8}{(8+1)\times 64} = 87.5\%$, where $\alpha = \lceil \frac{2000}{220+29} \rceil = 8$. According to Eq. (17) in On-demand Lazy-RTGC, we configure the $\sigma_t = 0.5$ first to pre-allocate more free blocks for the translation block and then we can get the space utilization ratio of On-demand Lazy-RTGC at about 80%. In GFTL, there is a central write buffer to serve the coming write requests when running partial garbage collection, whereas it exists in a distributed write buffer (i.e., buffer block) in RFTL for

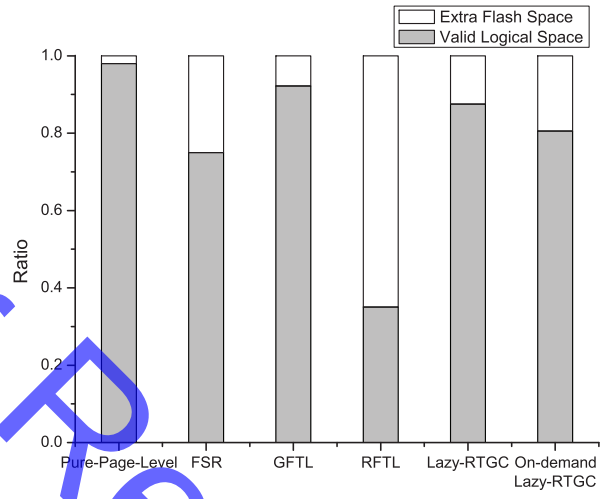


Fig. 11. The space utilization ratio.

each logical block. The buffer length in GFTL is limited by $N(k+1)/2$ so the utilization is about 92.18%. RFTL pre-allocated three physical blocks to one logical block; thus the space utilization is very low (about 33.3%). As shown in Figure 11, our scheme can achieve an 80.77% reduction on average in extra flash-space overhead compared with RFTL, but costs a little more space than GFTL. Since FSR cannot get real-time task information, we set 75% as the ratio between the logical space and physical space. From the experimental results, we see FSR has better average performance and block-erase counts. However, it cannot guarantee the worst-case system response time. Pure-Page-Level does not apply any real-time mechanism so that the space utilization is close to 100%.

6.3. Memory and Energy Overhead

In this section, we analyze the memory overhead caused by Lazy-RTGC and other representative schemes, that is, Pure-Page-Level, FSR, GFTL, RFTL, Lazy-RTGC, and On-demand Lazy-RTGC. Table VII presents the experimental results. In these schemes, Pure-Page-Level, FSR, Lazy-RTGC, and On-demand Lazy-RTGC adopt page-level address mapping in which the entire page-level mapping table is cached in the RAM space. This will incur large RAM-space cost. As shown in Table VII, for a 32GB NAND flash memory, the RAM cost for the Pure-Page-Level scheme is 64MB. FSR, Lazy-RTGC, and On-demand Lazy-RTGC adopt the overprovisioning strategy that limits the logical address space to be smaller than the physical address space. According to the different space utilization ratios, the RAM costs for FSR, Lazy-RTGC, and On-demand Lazy-RTGC are different. GFTL uses block-level address mapping, but requires a page-level address mapping table to manage the central write buffer. Therefore GFTL still needs about a 12.2MB RAM footprint to maintain the address mapping table. RFTL adopts a hybrid-level address mapping scheme and the entire mapping table is stored in the OOB area. Therefore, it can significantly reduce the RAM cost. On-demand Lazy-RTGC stores the entire page-level mapping table in flash memory and caches a small number of mapping items in RAM space, so reducing large RAM cost. From the experimental results, On-demand Lazy-RTGC can reduce 90.0% and 47.6% RAM overhead compared to Pure-Page-Level and GFTL, respectively.

Table VI. Parameters of CACTI Power Model [Shyamkumar et al. 2008]

Parameter	Value
Capacity (MB)	16
Output width (bits)	512
Number of banks	1
Number of read/write ports	1
Technology-node (nm)	65
Temperature (K)	360
SRAM cell/wordline technology flavor	ITRS HP
Peripheral/Global circuitry technology flavor	ITRS HP
Wire type inside/outside mat	Semi-global

Table VII. Memory and Energy Overhead Comparison

Metrics	Pure-Page-Level	FSR	RFTL	GFTL	Lazy-RTGC	On-demand Lazy-RTGC
RAM overhead (KB)	65,536	49,152	1,024	12,480	57,344	6,528
Energy consumption (nJ)	19.193	14.609	1.155	5.545	16.452	3.822

The energy consumption is affected by the size and number of accesses to RAM. There is research on power consumption analysis of storage devices [HP Laboratories 2009; Vidyabhushan et al. 2013]. We select CACTI 5.3 [HP Laboratories 2009] as the RAM power model. The parameters of the power model are listed in Table VI. We calculate the energy consumption by applying different RAM-space requirements from evaluated schemes to the power model and the results are shown in Table VII. Pure-Page-Level, FSR, and Lazy-RTGC adopt page-level mapping by which the entire page-level mapping table is cached in the RAM. The large RAM overhead leads to a large energy overhead in these schemes. RFTL stores the entire mapping table in the OOB area so its RAM overhead and energy consumption are very small. However, RFTL has very low space utilization ratio and its average performance is degraded due to a large number of garbage collections. GFTL uses the block-level mapping scheme to record primary block mapping and a page-level mapping table is used for the central write buffer. Though the RAM cost and energy cost are better than those of page-level mapping schemes, GFTL still costs 79.2% extra energy consumption compared to RFTL. To reduce the large energy consumption in Lazy-RTGC, we make use of the on-demand approach that stores the entire page-level mapping table in flash memory and only caches a small number of mapping items in the RAM. Therefore On-demand Lazy-RTGC can reduce 80.1% and 31.1% RAM energy overhead compared to Pure-Page-Level and GFTL, respectively.

7. RELATED WORKS

The design and architecture of *flash translation layer* (FTL) for flash storage systems has been extensively studied in the literature. Many schemes have been proposed to optimize the management of address mapping information [Wu et al. 2007; Lee et al. 2007; Liu et al. 2012; Zhang et al. 2013] and to solve endurance and power failure problems [Chang 2007; Wang et al. 2011; Guo et al. 2013; Chen et al. 2013; Chang et al. 2013b, 2015; Zhang et al. 2014; Huang et al. 2014a]. Only few works are about real-time NAND flash memory storage systems. Among them, Chang et al. [2004] proposed a real-time garbage collection mechanism. The mechanism needs to get real-time task information from the file system and allocate a garbage collection task to each real-time task for replenishing required free pages. In order to determine the reclaimed free

pages after each garbage collection process, the mechanism limits that logical address space which is smaller than the physical space, as the overprovisioning strategy. The number of valid pages in each data block can have an upper bound. Therefore this mechanism can guarantee that each real-time task has enough free pages to execute. However, it does not consider the average performance that can promise the deadline of each real-time task. Moreover, the mechanism needs extra real-time task information from the file system, which may require significant modification to current file systems.

Partial garbage collection is proposed in GFTL [Choudhuri and Givargis 2008]. In GFTL, a block-level mapping scheme is adopted, and there is a central garbage collection queue to record the block number needed to execute garbage collection. A write buffer is used to handle the coming request when the corresponding data block is full. Once the block is full, the block number will be added to the garbage collection queue and wait for garbage collection. Before the corresponding block is erased, the coming write request will be written to the write buffer. GFTL can guarantee that the buffer length has an acceptable upper bound. However, there are many unnecessary garbage collections and valid page copies. Many hot data blocks may be erased frequently while other blocks stay free for a long time.

To solve the problem, Qin et al. [2012] proposed a distributed partial garbage collection scheme called RFTL [2012]. In RFTL, each logical block is allocated to three physical blocks and the partial garbage collection process is distributed to the corresponding logical block. That is, the write request only triggers its own corresponding partial garbage collection and the data is moved within three pre-allocated physical blocks. RFTL can reduce the performance impact from partial garbage collection but cannot solve the unnecessary garbage collection problem, since cold free blocks are still not be used due to the block-level mapping scheme. Moreover, the low space utilization that costs 2/3 physical space is another problem in RFTL.

There is also much research on the architectures of *solid state drives* (SSDs) [Payer et al. 2009; Seong et al. 2010; Sun et al. 2010; Chang and Wen 2014; Chang et al. 2014b; Wu et al. 2013]. These works mainly focus on the optimization of parallel read or write without considering the garbage collection in SSDs. Lee et al. [2013] proposed a *pre-emptible garbage collection* (PGC) scheme for SSDs. PGC can identify pre-emption points that can minimize the pre-emption overhead, and can merge incoming I/O requests to enhance the performance of SSDs. However, PGC does not consider the mapping scheme impact performance. The objective of PGC is to improve the system performance instead of providing deterministic garbage collection. Therefore it may not provide effective solutions to our problem. There are also some write buffer and cache schemes proposed to improve write performance [Wu et al. 2006; Ding et al. 2007; Shi et al. 2013, 2014; Hu et al. 2013]. These techniques target at the management of data buffers during garbage collection. The objectives are to reduce the write activities. Craciunas et al. [2008] proposed a system call scheduling technique for resource and workload management in network- and disk-related system call scheduling [Huang et al. 2014b]. Although the scheduling technique can effectively control I/O process behavior with good real-time performance, it targets at hard disk drives instead of flash memory. Our technique can be combined with these techniques to provide deterministic garbage collection and optimize both the average and worst performance for NAND flash memory storage systems.

8. CONCLUSION

In this article, we have proposed a real-time lazy garbage collection mechanism for NAND flash memory storage systems. By making use of an on-demand page-level mapping scheme and a partial garbage collection technique, Lazy-RTGC can guarantee system response time in the worst case and further optimize the average system

performance. We have evaluated Lazy-RTGC using a set of benchmarks and compared with representative works. The simulation results show that our scheme can significantly improve both the average and worst system performance with very low extra flash-space requirements. In the future, the on-demand method will be optimized to further reduce the RAM-space cost, and we would like to make use of the system idle time to invoke garbage collection early and to avoid the flash memory staying in a long-term worst-case situation. We plan to apply our real-time lazy garbage collection mechanism to the parallelism architecture in solid-state drives. Moreover, Vidyabhushan et al. [2013] proposed FlashPower to model the power consumption of NAND flash memory. We also plan to use FlashPower to evaluate the power consumption of Lazy-RTGC and previous works.

REFERENCES

Airlines Electronic Engineering Committee (Aeec). 1991. Arinc Specification 651. http://infostore.saiglobal.com/store/details.aspx?ProductID=797110.

David F. Bacon, Perry Cheng, and V. T. Rajan. 2003. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03)*. ACM Press, New York, 285–298.

Amir Ban. 1995. Flash file system. US patent 5,404,485.

John S. Bucy and Gregory R. Ganger. 2003. The DiskSim simulation environment version 3.0 reference manual. Tech. rep., CMU-CS-03-102, Carnegie Mellon University.

Che-Wei Chang, Chuan-Yue Yang, Yuan-Hao Chang, and Tei-Wei Kuo. 2014a. Booting time minimization for real-time embedded systems with non-volatile memory. *IEEE Trans. Comput.* 63, 4, 847–859.

Da-Wei Chang, Hsin-Hung Chen, Dau-Jieu Yang, and Hsung-Pin Chang. 2014b. BLAS: Block-level adaptive striping for solid-state drives. *ACM Trans. Des. Autom. Electron. Syst.* 19, 2.

Hung-Sheng Chang, Yuan-Hao Chang, Pi-Cheng Hsiu, Tei-Wei Kuo, and Hsiang-Pang Li. 2015. Marching-based wear-leveling for PCM-based storage systems. *ACM Trans. Des. Autom. Electron. Syst.* 20, 2.

Li-Pin Chang. 2007. On efficient wear leveling for large-scale flash-memory storage systems. In *proceedings of the ACM Symposium on Applied Computing (SAC'07)*. ACM Press, New York, 1126–1130.

Li-Pin Chang, Tung-Yang Chou, and Li-Chun Huang. 2013a. An adaptive, low-cost wear-leveling algorithm for multichannel solid-state disks. *ACM Trans. Embed. Comput. Syst.* 13, 3.

Yuan-Hao Chang, Ming-Chang Yang, Tei-Wei Kuo, and Ren-Hung Hwang. 2013b. A reliability enhancement design under the flash translation layer for MLC-based flash-memory storage systems. *ACM Trans. Embed. Comput. Syst.* 13, 1.

Li-Pin Chang, Tei-Wei Kuo, and Shi-Wu Lo. 2004. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Trans. Embed. Comput. Syst.* 3, 4, 837–863.

Li-Pin Chang and Chen-Yi Wen. 2014. Reducing asynchrony in channel garbage-collection for improving internal parallelism of multichannel solid-state disks. *ACM Trans. Embed. Comput. Syst.* 13, 2.

Yang Chang and A. Wellings. 2010. Garbage collection for flexible hard real-time systems. *IEEE Trans. Comput.* 59, 8, 1063–1075.

Renhai Chen, Yi Wang, and Zili Shao. 2013. DHeating: Dispersed heating repair for self-healing NAND flash memory. In *Proceedings of the IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'13)*. 1–10.

Siddharth Choudhuri and Tony Givargis. 2008. Deterministic service guarantees for NAND flash using partial block cleaning. In *Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'08)*. ACM Press, New York, 19–24.

Tae-Sun Chung, Dong-Joo Park, Sangwon Park, Dong-Ho Lee, Sang-Won Lee, and Ha-Joo Song. 2009. A survey of flash translation layer. *J. Syst. Archit.* 55, 5–6, 332–343.

Silviu S. Craciunas, Christoph M. Kirsch, and Harald Rock. 2008. I/O resource management through system call scheduling. *SIGOPS Oper. Syst. Rev.* 42, 5, 44–54.

Xiaoning Ding, Song Jiang, and Feng Chen. 2007. A buffer cache management scheme exploiting both temporal and spatial localities. *ACM Trans. Storage* 3, 2.

DiskMon. 2013. DiskMon for Windows. http://technet.microsoft.com/en-us/sysinternals/bb896646.aspx.

Yong Guan, Guohui Wang, Yi Wang, Renhai Chen, and Zili Shao. 2013. BLog: Block-level log-block management for NAND flash memory storage systems. In *Proceedings of the 14th ACM SIGPLAN/SIGBED*

Conference on Languages, Compilers and Tools for Embedded Systems (LCTES'13). ACM Press, New York, 111–120.

Jie Guo, Jun Yang, Youtao Zhang, and Yiran Chen. 2013. Low cost power failure protection for MLC NAND flash storage systems with PRAM/DRAM hybrid buffer. In *Proceedings of Design, Automation Test in Europe Conference Exhibition (DATE'13)*. 859–864.

Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. 2009. DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. ACM Press, New York, 229–240.

HP Laboratories. 2009. CACTI: An integrated cache and memory access time, cycle time, area, leakage, and dynamic power model. http://www.hpl.hp.com/research/cacti/.

Jen-Wei Hsieh, Yuan-Hao Chang, and Yuan-Sheng Chu. 2013. Implementation strategy for downgraded flash-memory storage devices. *ACM Trans. Embed. Comput. Syst.* 12, 1.

Jen-Wei Hsieh, Kam-Yiu Lam, Po-Chun Huang, Yuan-Hao Chang, and Jiantao Wang. 2014. Block-based multi-version b¡sup¿+¡/sup¿-tree for flash-based embedded database systems. *IEEE Trans. Comput.* 99, 1.

Jingtong Hu, Chun Jason Xue, Wei-Che Tseng, Y. He, Meikang Qiu, and E. H.-M. Sha. 2010. Reducing write activities on non-volatile memories in embedded CMPs via data migration and recomputation. In *Proceedings of the 47th ACM/IEEE Design Automation Conference (DAC'10)*. 350–355.

Jingtong Hu, Chun Jason Xue, Qingfeng Zhuge, Wei-Che Tseng, and Edwin H.-M. Sha. 2013. Write activity reduction on non-volatile main memories for embedded chip multiprocessors. *ACM Trans. Embed. Comput. Syst.* 12, 3.

Po-Chun Huang, Yuan-Hao Chang, and Tei-Wei Kuo. 2013. An index-based management scheme with adaptive caching for huge-scale low-cost embedded flash storages. *ACM Trans. Des. Autom. Electron. Syst.* 18, 4.

Po-Chun Huang, Yuan-Hao Chang, Kam-Yiu Lam, Jian-Tao Wang, and Chien-Chin Huang. 2014a. Garbage collection for multiversion index in flash-based embedded databases. *ACM Trans. Des. Autom. Electron. Syst.* 19, 3.

Yazhi Huang, Liang Shi, Jianhua Li, Qingan Li, and C. J. Xue. 2014b. WCET-aware re-scheduling register allocation for real-time embedded systems with clustered VLIW architecture. *IEEE Trans. VLSI. Syst.* 22, 1, 168–180.

Dawoon Jung, Jeong-Uk Kang, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. 2010. Superblock FTL: A superblock-based flash translation layer with a hybrid address translation scheme. *ACM Trans. Embed. Comput. Syst.* 9, 1.

Taehyoun Kim, Naehyuck Chang, and Heonshik Shin. 2000. Bounding worst case garbage collection time for embedded real-time systems. In *Proceedings of the 6th IEEE Real-Time Technology and Applications Symposium (RTAS'00)*. 46–55.

Youngjae Kim, B. Tauras, A. Gupta, and B. Urgaonkar. 2009. FlashSim: A simulator for NAND flash-based solid-state drives. In *Proceedings of the 1st International Conference on Advances in System Simulation (SIMUL'09)*. 125–131.

Junghee Lee, Youngjae Kim, G. M. Shipman, S. Oral, and Jongman Kim. 2013. Preemptible I/O scheduling of garbage collection for solid state drives. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 32, 2, 247–260.

Sungjin Lee, Dongkun Shin, Young-Jin Kim, and Jihong Kim. 2008. LAST: Locality-aware sector translation for NAND flash memory-based storage systems. *ACM SIGOPS Oper. Syst. Rev.* 42, 6, 36–42.

Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. 2007. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Trans. Embed. Comput. Syst.* 6, 3.

Duo Liu, Yi Wang, Zhiwei Qin, Zili Shao, and Yong Guan. 2012. A space reuse strategy for flash translation layers in SLC NAND flash memory storage systems. *IEEE Trans. VLSI. Syst.* 20, 6, 1094–1107.

Micron. 2012. Micron NAND flash memory MT29E128G08CECAB datasheet. http://www.micron.com/products/nand-flash.

Hannes Payer, Ma Sanvido, Zvonimir Z. Bandic, and Christoph M. Kirsch. 2009. Combo drive: Optimizing cost and performance in a heterogeneous storage device. In *Proceedings of the 1st Workshop on Integrating Solid-State Memory into the Storage Hierarchy*. Vol. 1, 1–8.

Zhiwei Qin, Yi Wang, Duo Liu, and Zili Shao. 2011. A two-level caching mechanism for demand-based page-level address mapping in NAND flash memory storage systems. In *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'11)*. 157–166.

Zhiwei Qin, Yi Wang, Duo Liu, and Zili Shao. 2012. Real-time flash translation layer for NAND flash memory storage systems. In *Proceedings of the IEEE 18th Real-Time and Embedded Technology and Applications Symposium (RTAS'12)*. 35–44.

Samsung Electronics. 2007. Samsung K9G4G08U0A(v1.0)-4Gb MLC NAND flash data sheet. http://www.samsung.com/global/business/semiconductor/minisite/SSD/us/html/why/MlcNandFlash.html?gclid=CjwKEAjwvbGqBRCs3eH4o5C74CYSJAB3TODsbwzLCNWlEgf1SPOqaPTqPBZzdL8Sf3uHtustWH9vYdxoCK9vw_wcB.

Yoon Jae Seong, Eyec Hyun Nam, Jin Hyuk Yoon, Hongseok Kim, Jin Yong Choi, Sookwan Lee, Young Hyun Bae, Jaejin Lee, Yookun Cho, and Sang-Lyul Min. 2010. Hydra: A block-mapped parallel flash memory solid-state disk architecture. *IEEE Trans. Comput.* 59, 7, 905–921.

Liang Shi, Jianhua Li, Qingan Li, C. J. Xue, Chengmo Yang, and Xuehai Zhou. 2014. A unified write buffer cache management scheme for flash memory. *IEEE Trans. VLSI. Syst.* 22, 12, 2779–2792.

Liang Shi, Jianhua Li, Chun Jason Xue, and Xuehai Zhou. 2013. Cooperating virtual memory and write buffer management for flash-based storage systems. *IEEE Trans. VLSI. Syst.* 21, 4, 706–719.

Thoziyoor Shyamkumar, Muralimanohar Naveen, Ahn Jung, Ho, and P. Jouppi Norman. 2008. CACTI 5.1. http://www.hpl.hp.com/techreports/2008/HPL-2008-20.html.

Spansion. 2013. Spansion SLC NAND flash memory for embedded S34ML04G1 data sheet. http://www.spansion.com/Support/Datasheets/S34ML01G1_04G1.pdf.

Guangyu Sun, Yongsoo Joo, Yibo Chen, Dimin Niu, Yuan Xie, Yiran Chen, and Hai Li. 2010. A hybrid solid-state storage architecture for the performance, energy consumption, and lifetime improvement. In *Proceedings of the IEEE 16th International Symposium on High Performance Computer Architecture (HPCA'10)*. 1–12.

Toshiba. 2008. Toshiba CMOS NAND E2PROM (multi-level-cell) TC58NVG4T2ETA00 datasheet. https://www.toshiba.com/taec/components/Datasheet/TC58NVG0S3ETA00.pdf.

Toshiba. 2012. TOSHIBA NAND flash memory (SLC large capacity) TC58NVG3S0FBAID datasheet. http://media.digikey.com/pdf/Data%20Sheets/Toshiba%20PDFs/NAND_Flash_Memory(SLC_Large_Capacity)_Web.pdf.

UMASS. 2013. Trace from UMass trace repository. http://traces.cs.umass.edu/index.php/Storage/Storage.

Mohan Vidyabhushan, Trevor Bunker, Laura Grupp, Sudhanva Gurumurthi, Mircea R. Stan, and Steven Swanson. 2013. Modeling power consumption of NAND flash memories using FlashPower. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 32, 7, 1031–1044.

Yi Wang, Duo Liu, Zhiwei Qin, and Zili Shao. 2011. An endurance-enhanced flash translation layer via reuse for NAND flash memory storage systems. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE'11)*. 1–6.

Yi Wang, Zili Shao, H. C. B. Chan, L. A. D. Bathen, and N. D. Dutt. 2014. A reliability enhanced address mapping strategy for three-dimensional (3-D) NAND flash memory. *IEEE Trans. VLSI. Syst.* 22, 11, 2402–2410.

Chin-Hsien Wu and Tei-Wei Kuo. 2006. An adaptive two-level management for the flash translation layer in embedded systems. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'06)*. 601–606.

Chin-Hsien Wu, Tei-Wei Kuo, and Li Ping Chang. 2007. An efficient B-tree layer implementation for flash-memory storage systems. *ACM Trans. Embed. Comput. Syst.* 6, 3.

Chin-Hsien Wu, Tei-Wei Kuo, and Chia-Lin Yang. 2006. A space-efficient caching mechanism for flash-memory address translation. In *Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06)*.

Chin-Hsien Wu and Hsin-Hung Lin. 2012. Timing analysis of system initialization and crash recovery for a segment-based flash translation layer. *ACM Trans. Des. Autom. Electron. Syst.* 17, 2.

Guanying Wu, Xubin He, Ningde Xie, and Tong Zhang. 2013. Exploiting workload dynamics to improve SSD read latency via differentiated error correction codes. *ACM Trans. Des. Autom. Electron. Syst.* 18, 4.

Chi Zhang, Yi Wang, Tianzheng Wang, Renhai Chen, Duo Liu, and Zili Shao. 2014. Deterministic crash recovery for NAND flash based storage systems. In *Proceedings of the 51st Annual Design Automation Conference (DAC'14)*. ACM Press, New York.

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

Received July 2013; revised December 2014; accepted March 2015