Reinforcement Learning-Assisted Garbage Collection to Mitigate Long-Tail Latency in SSD

WONKYUNG KANG, Seoul National University
DONGKUN SHIN, Sungkyunkwan University
SUNGJOO YOO, Seoul National University

NAND flash memory is widely used in various systems, ranging from real-time embedded systems to enterprise server systems. Because the flash memory has erase-before-write characteristics, we need flash-memory management methods, i.e., address translation and garbage collection. In particular, garbage collection (GC) incurs long-tail latency, e.g., 100 times higher latency than the average latency at the 99th percentile. Thus, real-time and quality-critical systems fail to meet the given requirements such as deadline and QoS constraints. In this study, we propose a novel method of GC based on reinforcement learning. The objective is to reduce the long-tail latency by exploiting the idle time in the storage system. To improve the efficiency of the reinforcement learning-assisted GC scheme, we present new optimization methods that exploit fine-grained GC to further reduce the long-tail latency. The experimental results with real workloads show that our technique significantly reduces the long-tail latency by 29–36% at the 99.99th percentile compared to state-of-the-art schemes.

CCS Concepts: • **Computing methodologies → Reinforcement learning**; • **Computer systems organization → Firmware**; • **Hardware → Memory and dense storage**;

Additional Key Words and Phrases: Flash storage system, SSD, garbage collection, long-tail latency, reinforcement learning

ACM Reference format:
Wonkyung Kang, Dongkun Shin, and Sungjoo Yoo. 2017. Reinforcement Learning-Assisted Garbage Collection to Mitigate Long-Tail Latency in SSD. *ACM Trans. Embed. Comput. Syst.* 16, 5s, Article 134 (September 2017), 20 pages.
https://doi.org/10.1145/3126537

1 INTRODUCTION

Flash memory storages are widely used in embedded systems, and consumer and enterprise-server systems. Flash memory has two principal issues: (1) erase-before-write (write once) property, and (2) endurance problem. To address the erase-before-write property, a flash-translation layer (FTL)

This article was presented in the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES) 2017 and appears as part of the ESWEEK-TECS special issue.
This work was supported by National Research Foundation of Korea (NRF-2016M3A7B4909604).
Authors' addresses: W. Kang and S. Yoo (corresponding author), Department of Computer Science and Engineering, Seoul National University, 1, Gwanak-ro, Gwanak-gu, Seoul, 08826, Republic of Korea; emails: wkkang@gmail.com, sungjoo.yoo@gmail.com; D. Shin, Department of Software, Sungkyunkwan University, 2066, Seobu-ro Jangan-gu, Suwon, Gyeonggi-do, 16419, Republic of Korea; email: dongkun@skku.edu.

https://doi.org/10.1145/3126537

is employed. Currently, a page-level mapping [11] is being widely used to reduce the write latency induced by write-once and bulk-erase properties of flash memory storages. In the page-level mapping, when writing new data, FTL assigns a new free page, and subsequently, writes data to the newly assigned free page. Thereafter, it updates the address-mapping information between the logical and the physical addresses. If the free blocks are insufficient, they are obtained by reclaiming the unused space in the used blocks. To do that, the valid pages of the victim block are copied to a new block. The victim block is then erased to obtain a free block. This procedure is called garbage collection (GC). The GC induces a long-latency problem because the page-copy and block-erase operations are time consuming.

GC latency increases as the capacity of flash memory increases. It is mainly due to the fact that the block size (number of pages per block) increases as the capacity of Flash memory increases. GC latency is determined by the time for valid page copy and block erase. Thus, as block size gets increased, GC latency also increases. According to our analysis, the block size has a strong impact on long tail latency. Especially, the block size gets increased from 2D to 3D NAND flash memory, e.g., 256 pages/block in 2D planner NAND flash memory [9] and 768 pages/block in 3D NAND flash memory [8]. Even in 3D NAND flash memory, the block size is expected to continue to increase [23, 24]. Thus, the long write latency problem incurred by GC can become more serious in 3D NAND flash memory-based storage. Note that the long write latency due to GC can increase not only write latency but also read latency since GC can stall the service of subsequent read requests.

A long tail is observed in the distribution of the write latency because of the GC. For instance, the latency at the 99^{th} percentile can be 100x higher than the average latency [22]. Such a long-tail latency causes a significant problem in real-time embedded and enterprise-server systems which need to meet the real-time and quality of service (QoS) requirements.

In this study, we propose a reinforcement learning-assisted GC technique to reduce the long-tail latency. The proposed technique is a new approach to exploit the idle time in the storage with reinforcement learning.

The contributions of this study are as follows.

- To the best of the authors' knowledge, this is the first approach of reinforcement learning-assisted idle time-aware GC.
- The proposed reinforcement learning-assisted solution helps determine the number of GC operations to be executed to exploit the varying idle time while avoiding the long-tail latency due to the GC.
- We also present an optimization scheme that aggressively performs fine-grained GC to prepare free blocks in advance, thereby reducing the blockage due to the GC, which significantly reduces the long-tail latency.

The rest of this paper is organized as follows. Section 2 reviews previous GC techniques. Section 3 explains the motivation behind our study and the problem discussed herein. Section 4 describes the background of flash-storage systems and reinforcement learning. Section 5 presents the proposed method. Section 6 gives the experimental results. Section 7 concludes this paper.

2 RELATED WORK

Several techniques have been proposed to improve the GC performance [1, 2, 5, 13–17]. Wei et al. 0identified workload characteristics per address range and assigned page or block-level mapping based on identifying the workload [13]. Similarly, Jang et al. classified the data into three types such as hot, cold, and warm and allocated blocks such that a block is assigned to the same type of data, which improves the GC performance [14].

Studies have been conducted on approaches that utilize the idle time and workload prediction. Han et al. predicted the future workload and controlled the number of victim blocks [15]. The victim blocks are selected based on the age, utilization, and erase counts. The number of reclaimed blocks is then determined by predicting the history of the request count and rate. Lin et al. predicted the future workload and obtained the number of victim blocks based on the predicted workload, erase count, and invalidation period [16].

Only a few studies have been conducted on real-time GC for flash-storage systems. Chang et al. proposed a free-page replenishment mechanism wherein the real-time tasks were prevented from being blocked due to insufficient number of free pages. Assuming the write behavior of a real-time task is known, the number of GC operations and the maximum quantum for GC operation are determined to meet the real-time constraints [17].

Choudhuri et al. proposed GFTL, which helps perform partial GC to ensure fixed upper bounds in the latency of storage access by eliminating the source of non-determinism [5]. Qin et al. proposed a distributed partial GC policy in the RFTL, which tries to hide the long-tail latency due to the GC. Periodically, the method helps perform partial GC and exploit buffer blocks to store the write data obtained during the GC operation, thereby reducing the GC-induced blockage [2].

Zhang et al. proposed a lazy GC method termed LazyRTGC. In this method, a page-level mapping is employed to fully utilize the flash memory space and postpone the GC as much as possible. To employ the idle time of the system, LazyRTGC schedules a partial GC after serving write requests [1]. However, a fixed policy is employed to utilize the idle time. Thus, as demonstrated in our experiments, it does not consider the duration of the idle time determined by the dynamic behavior of the storage access thereby losing an opportunity to further exploit the idle time.

Reinforcement learning has been widely used in a broad range of problems including robot control and resource allocation in data center. In [25], Ipek et al. proposed a self-optimizing DRAM controller design based on reinforcement learning. This memory controller sees the system state and predicts the long-term performance impact of each action it can perform. In this way, this controller learns to optimize its scheduling policy to offer maximum performance. In [26], Wang et al. proposed deriving a near-optimal power management policy using reinforcement learning and Bayesian classification. In [27], Peled et al. proposed context-based prefetcher using reinforcement learning.

3 PROBLEM AND MOTIVATION

3.1 Long-Tail Problem in Flash Storage Access Latency

Figure 1 shows the latency comparison for a storage trace called home2 (used in our experiments) between an ideal storage without a GC overhead and a real one with page-level mapping. The figure shows that the response time is short for the majority of the storage accesses. It is less than 1 ms for approximately 85% of the accesses. However, the latency difference between the median and the 99^{th} percentile is a factor of 100. As mentioned before, such a long-tail latency is a serious problem in real-time and quality-critical systems. For instance, the server storage typically needs to provide a minimum 7.5 ms of write latency for 99.99% of the storage accesses [21]. Considering that the GC latency continues to increase due to the increasing block size, it is important to reduce the long-tail latency for such real-time and quality-critical systems.

3.2 Idle Time in Flash Storage

Figure 2 shows the distribution of the request interval time for 60K requests the real-world workloads used in our experiments. The x-axis represents the inter-request interval time, and the y-axis represents the frequency of the request in each bin. As the figure shows, the storage system has

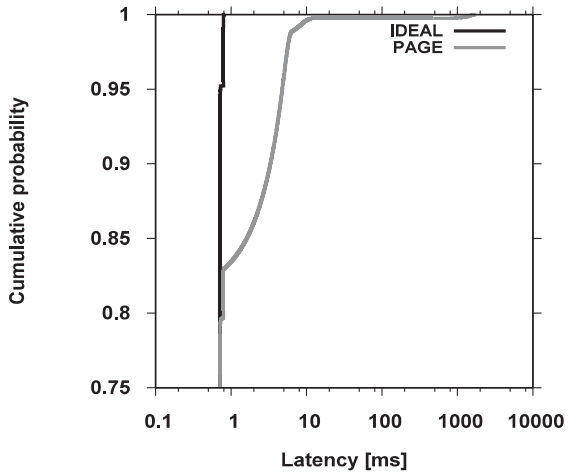

Fig. 1. Long tail latency problem: trace home2.

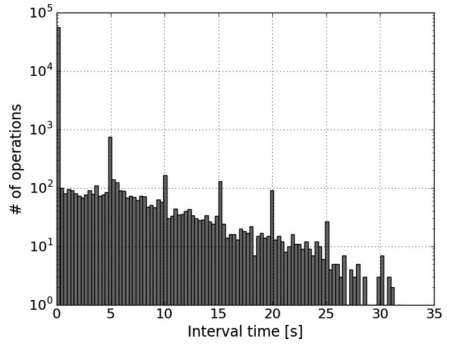

Fig. 2. Inter-request interval distribution.

frequent and long idle periods. Such an idle time can be exploited to perform GC operations. In idle time-aware GC methods [15, 16], it is important to determine how many GC operations need to be performed for a given idle time. The difficulty of this problem is that the length of the current idle period is unknown. To address this problem, several techniques exist [15, 16]. These techniques use fixed policies determined at the design time. Thus, they are limited in adapting to the dynamically changing storage access behavior because of the different program runs or phases. In this study, we propose an RL-assisted adaptive GC method, which learns the storage access behavior online and adjusts the GC to it to reduce the long-tail latency.

4 BACKGROUND

4.1 SSD Architecture and Garbage Collection

Solid-state drives (SSDs) are one of the flash-storage systems widely used in consumer and enterprise systems. Figure 3 shows the internal architecture of the SSD. It comprises flash-memory chip packages for data storage, a controller, and DRAM for the buffer. The controller is connected to the host interface (e.g., SATA) and flash-memory packages. To exploit the parallelism of multiple flash memories to maximize the performance, multiple flash-memory interface channels are

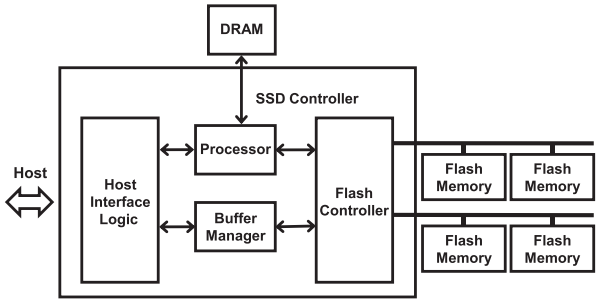

Fig. 3. SSD internal architecture diagram.

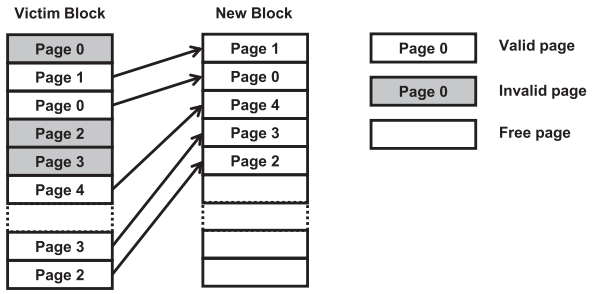

Fig. 4. Garbage collection.

employed in the flash controller. The DRAM stores the address mapping table and read/write data for caching and buffering [12].

Figure 4 illustrates the GC operation. GC is typically triggered if the number of free blocks is less than a certain threshold, e.g., 5% of the total number of blocks. To reduce the cost of page copy, a block having the lowest number of valid pages is typically selected as a victim block. As shown in the figure, the valid pages (e.g., pages 1, 0, 4, 3, and 2) are read from the victim block and written to a free block, which is called the valid page copy. After copying all the valid pages, the victim block is erased to obtain a free block. The GC latency depends on the number of valid pages in the victim blocks, which is proportional to the block size, i.e., the number of pages in a block. As the 3D NAND flash memory becomes more popular, the block size increases rapidly, thereby increasing the GC latency, which can make the GC-induced long-tail write-latency problem more severe in the 3D NAND flash memory. Note that the GC can increase the latency of the read access and that of the write access because the flash memory is blocked during the GC operation. Specifically, the plane under the valid page copy or block erase is blocked to subsequent accesses, which increases the read or write latency of the blocked plane. Note that a flash memory die contains two or four planes. A plane consists of a large number of blocks. Each plane can be accessed independently.

4.2 Reinforcement Learning

Figure 5 shows a simplified view of the reinforcement learning (RL). The agent (e.g., the GC scheduler in the SSD controller) has a set of actions. In our work, the actions are defined to be partial GC operations, e.g., 5 page copies or erase operation. Thus, a policy selects one of possible actions whether it is page copy or erase operation. The environment has states e.g., active and idle states. Based on the current state, the policy of the agent tries to maximize the immediate reward,

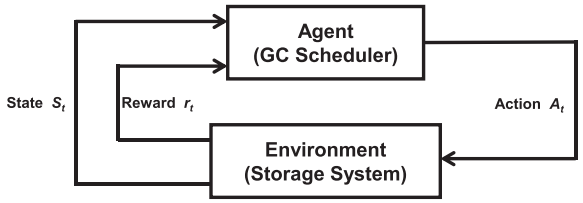

Fig. 5. Environment - agent interaction.

e.g., reduction in the long-tail latency, in selecting an action. After executing an action, the environment can enter a new state.

Note that the reinforcement learning is an online method. Thus, it applies exploitation (e.g., taking an action suggested by the current policy) and exploration (e.g., trying an action different from the one the current policy suggests and updating the policy with the reward) during the runtime [10].

The basic model of the reinforcement learning is given as follows.

State (S): a set of environment and agent states
Action (A): a set of actions of agent
Reward (r): reward associated with last action
Policy (π): agent's way of action selection at a given time

As shown in Figure 5, the agent interacts with the environment in discrete time steps. At time t, the agent receives observation o_t, which includes the state S_t and reward r_t. The agent's policy selects an action A_t from a set of actions and sends it to the environment. The environment changes its state to a next state S_{t+1} and gives the agent reward r_{t+1} via the state transition (S_t, A_t, S_{t+1})[1]. The goal of the agent's policy is to maximize the reward.

To apply the reinforcement learning to our GC problem, we need to define four components: states, actions, reward, and policy, with respect to the storage system. In terms of the actions, we exploit a fine-grained partial GC method wherein an action involves performing a number of valid page copy operations or a single erase operation [17]. Thus, the only objective of the policy is to determine how many valid copies to perform or whether to perform an erase operation.

For policy learning, we employ Q-learning [10], which manages the value functions of the state-action pair and updates them based on the recent state-action pairs and rewards. The value function of the state-action pair is defined as follows.

$$Q(s, a) = E\{r_t | s_t = s, a_t = a\} \tag{1}$$

where s (s_t) and a (a_t) represent the state and action at time t, respectively, and r_t is the reward at time t. $Q(s,a)$ is the expectation of the reward when action a is taken at state s and time t.

The policy is defined as follows.

$$\pi(s) = argmax_a Q(s, a) \tag{2}$$

As expressed in Equation (2), the policy chooses an action to maximize the Q value at state s.[2] As mentioned earlier, the reinforcement learning is an online method. Thus, the policy is modified for the dynamic storage access behavior. Hence, we use ε-greedy technique [10]. In this method,

[1] (S_t, A_t, S_{t+1}) represents a state transition from S_t to S_{t+1} initiated by action A_t.
[2] There are two types of policy, deterministic and stochastic ones. The policy in Equation (2) is deterministic since the action of the maximum Q value is chosen. A stochastic policy chooses an action in a probabilistic manner that the probability of choosing action a is proportional to $Q(s,a)$. In our experiments, we applied the stochastic policy.

the agent, i.e., the GC scheduler, performs exploitation and exploration. In most cases, the GC scheduler selects an action, e.g., two valid page copies, by exploiting the learned policy. At a low probability, e.g., 1% (corresponding to ε), the agent explores a new possibility by taking a random action, which is different from the one selected by the policy. The policy is updated with the reward of this random choice in exploration or the one chosen by the policy in exploitation as follows.

$$Q(s, a) = Q(s, a) + \alpha\{r + \gamma Q(s', a') - Q(s, a)\} \quad (3)$$

where r is the reward (i.e., given from the response time of the storage access), s' and a' are the subsequent state and action (for the next storage access), respectively, α is the step size, and γ is the discount factor [10][3]. The basic concept of Equation (3) is that the Q value update is proportional to the difference between the target reward, i.e., $r + \gamma Q(s', a')$, and the old estimate of the reward, $Q(s, a)$.

When implementing the reinforcement learning, the key data structure is q-table, which stores $Q(s, a)$. The size (# of entries) of q-table is # states x # actions. The size needs to be small to reduce the memory overhead of the reinforcement learning-assisted solution.

In Equation (3), Q values are reused from the existing ones (in the q-table), which is called bootstrapping, enabling a fast calculation of Q value updates. Thus, only reward r, which is measured by the GC scheduler, is newly needed to update the Q value in the q-table, which finally updates the policy because the policy is determined by the Q values. The exploration finally improves the policy by adapting to the characteristics of the given storage accesses.

Given that a fine-grained GC method and Q-learning with ε-greedy technique are applied, our key contribution is to define the states and rewards for the reinforcement learning-assisted GC. In Section 5, we describe how the states and rewards are defined and when to trigger the RL-assisted GC scheduler.

5 PROPOSED METHODS

5.1 Solution Overview

We aim to reduce the long-tail latency by (1) hiding the GC latency by exploiting the idle time, and (2) minimizing the GC-induced blocking. In this section, we present an RL-assisted GC scheduler to hide the GC latency (Section 5.2) and an aggressive fine-grained partial GC scheme to reduce the blocking time (Section 5.3).

Our proposed RL-assisted GC scheduler is triggered in a lazy manner. Thus, only when an access request arrives at the storage and the number of free pages goes below a threshold [1], it is triggered. When triggered, it chooses an action. Because our GC method is based on the partial GC, the action is to perform a number of partial GC operations, e.g., five page copies from a victim block to a free block. Thus, the GC scheduler chooses an action, i.e., determines how many partial GC operations will be performed after serving the current request. An erase operation is performed when an action is chosen by the scheduler and a block is ready to be erased. In such a case, instead of executing the action, the block is erased.

After serving the request, the GC scheduler calculates the response time. Because our goal is to reduce the long-tail latency, we need to reflect the response time in our reward. We explain the details of how the reward is calculated using the response time in Section 5.2. Note that the response time of the k^{th} request gives the reward for the $k\text{-}1^{\text{th}}$ request. Thus, in the aforementioned Q-learning (Equation (3)), we update the Q value for the current state s and action a only after the next request is served and the corresponding reward is calculated.

[3]Step size α and the discount factor γ are set to typical values, 0.3 and 0.8, respectively [10].

ALGORITHM 1: RL-Assisted GC Scheduling

Inputs: request, $state_{t-1}$ (S_{t-1}), $state_t$ (S_t), $action_{t-1}$ (A_{t-1})
Output: $action_t$ (A_t)

```
1: if T_GC >= N_free then
2:     A_t = e_greedy(interval_{t-1}, interval_t, action_{t-1})
3:     if interval_t == 0 then
4:         go to line 1
5:     end if
6:     serve the request and obtain response time
7:     run partial_gc(A_t)
8:     r = reward(response_time)
9:     Q(S_{t-1}, A_{t-1}) = (1 − α)Q(S_{t-1}, A_{t-1}) + α[r + γQ(S_t,A_t)]
10: end if
```

In Section 5.2, we explain the baseline RL-assisted GC scheduling. In Section 5.3, we present a more aggressive method of GC to further reduce the long-tail latency.

5.2 RL-Assisted Garbage Collection Scheduling

States: In the reinforcement learning, the states need to represent the history, which helps in maximizing the reward. We propose using the following information as the states.

- Previous inter-request interval
- Current inter-request interval
- Previous action

The inter-request interval is an important information of history because it reflects the intensity (i.e., the idleness) of storage traffics. Thus, if the interval is large, the RL-assisted GC scheduler tends to take a more aggressive action, i.e., more number of partial GC operations. The previous action plays a role of a summary of both recent history and the decision of the GC scheduler.

From the viewpoint of the agent, both the host and the SSD subsystem constitute the environment. The inter-request intervals represent the state of the host. Note that the previous action can represent that of the SSD subsystem as well as that of the host. It is because the previous action does not only plays a role of a summary of both recent history and the decision of GC scheduler, but also affects the state of the SSD subsystem, i.e., being busy in page copy or idle. For instance, if the previous action is to copy a large number of pages, then the current state of SSD subsystem tends to be busy.

We divide each of the three components into multiple bins, 2 bins for previous inter-request interval, 17 bins for current inter-request interval, and 2 bins for previous action, which gives a total 68 ($=2 \times 17 \times 2$) states. The details of binning are given in Section 6.1.

Reward: Regarding the reward, we need to assign a larger reward for a smaller response time. We also need to penalize an action giving a long response time. Figure 6 shows our reward function. The reward ranges between −0.5 and 1. For instance, if the response time is large (larger than the threshold t_3), a negative reward is assigned to penalize the action.

The thresholds in the reward function in Figure 6 need to be adjusted to the characteristics of the storage accesses. A fixed set of thresholds will not cover diverse scenarios in the storage accesses. Thus, we set the thresholds based on the characteristics of the storage accesses. In particular, we set three thresholds, t_1, t_2, and t_3 to the 70^{th}, 90^{th}, and 99^{th} percentiles of the response time,

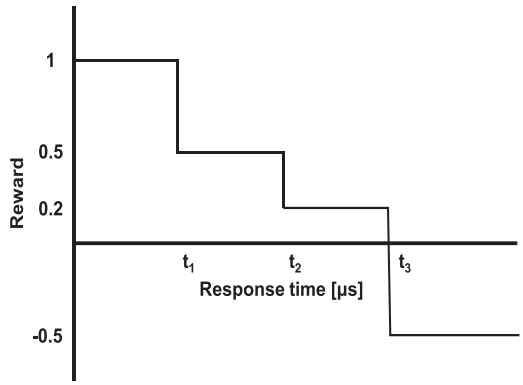

Fig. 6. Reward function.

respectively. Hence, even if the storage-access behavior changes, the thresholds can be adjusted based on the new distribution of the response time.

Exploitation and Exploration Balance: The exploration aims at filling in all the entries of the q-table, and subsequently, improving them toward the optimal policy. To do that, we employ the Ɛ-greedy technique [10]. In the initial period of RL execution (the first 1000 GC operations in our experiments), we utilize a large Ɛ value (80%) to perform aggressive explorations. Then, we utilize a small Ɛ value (1%) for a balance between exploitation and exploration during the rest of period.

GC Scheduling: Algorithm 1 shows the pseudo code of the proposed RL-assisted GC scheduler. For each request to the storage, the GC scheduler compares the number of free blocks N_{free} with threshold T_{GC} (=10 blocks in our experiments). If $T_{GC} >= N_{free}$, we call function e_greedy() (line 2), which performs either exploration or exploitation based on the probability of Ɛ, i.e., a random action is selected at a probability of Ɛ or an action is selected using the policy at a probability of 1 - Ɛ [10]. Note that we do not trigger the GC scheduler in case of consecutive requests wherein the inter-request interval is zero (line 3–5). After serving the request and obtaining the response time for the current request (line 6), we perform the selected action, i.e., partial GC operation (line 7). We then call the reward function with the response time of the current request (line 8). Finally, we update the q-table entry of the previous request (line 9). Note that, as mentioned previously, we update the entry of the q-table associated with the previous request.

Intensive Garbage Collection: The baseline method in Algorithm 1 is not free from a blocked situation wherein the flash storage is out of the free block. To avoid such a situation, we employ an intensive garbage collection (GC) method from LazyRTGC [1] and modify it for further improvement. The objective of the intensive GC is to perform more (5 or 7 valid page copies in our experiments) partial GC operations than that in the normal partial GC operations (typically, 1 or 2 page copies), thus enabling faster reclamation of free blocks. The number (5 or 7) of partial GC operations is determined by considering the number of pages in a block and other parameters of the flash memory, e.g., erase time [1].

In [1], the intensive GC is triggered when there is only one free block left. Under the intensive GC, the action chosen by the RL policy is ignored and a fixed number of partial GC operations is performed after serving a write request. In [1], after the number of free blocks becomes greater than one, the intensive GC is no longer applied. In our work, we propose to utilize a larger threshold (termed the threshold of the stopping intensive GC, T_{IGC}) than the one required to stop

applying the intensive GC. We use a larger one (3), which is obtained via a sensitivity analysis in our experiments.

5.3 Aggressive RL-Assisted Garbage Collection Scheduling

In this subsection, we propose two methods of aggressively triggering the GC to further reduce the long-tail latency. To reduce the long-tail latency, it is effective to limit the maximum number of partial GC operations per action. In our experiments, we found that when the number of partial GC operations is limited to two, the best result is obtained. Thus, when the policy chooses an action, and if the action has more than two partial GC operations, we set the number of GC operations to two. When applying this method, we need to consider the blocking situation where the flash storage is out of the free block) because we limit the maximum number of partial GC operations. To avoid the blocking situation, we trigger the GC collection more aggressively by introducing a new threshold for number of free blocks T^A_{GC}. T^A_{GC} is set higher than T_{GC} (10). We call this method early GC triggering with the maximum limit of partial GC operation, in short, *max-limited early GC triggering*. Note that, the maximum number of partial GC operations is limited only when the number of free blocks N_{free} is between T^A_{GC} and T_{GC}. When $N_{free} <= T_{GC}$, the maximum limit is not applied to the action chosen by the RL-assisted GC scheduler.

The aggressive GC operation can increase the erase count. To avoid this, we carefully select the victim blocks. When N_{free} is within the two thresholds T^A_{GC} and T_{GC}, we select a victim block only when it has a larger number of invalid pages than the threshold (60% of the block size in our experiments).

In conventional GC methods, a write request triggers GC when the number of free blocks is less than a certain threshold. In case of the read request, the GC is not triggered to avoid the increase in the read latency. We propose triggering a partial GC operation even for a read request when the triggering condition is met. Note that the latency of the read request does not increase because the GC operation is performed after serving the read request. We call this method *read-initiated GC triggering*.

Note that, in our aggressive method, the RL-assisted GC scheduler is triggered using the two methods: max-limited early GC triggering and read-initiated GC triggering. Based on our experiments, they prove useful in obtaining free blocks during the idle time, thereby reducing the long-tail latency.

6 EXPERIMENTS

6.1 Experimental Setup

We compare our proposed RL-assisted GC methods (**baseline** in Section 5.2 and **aggressive** in Section 5.3) with a typical GC method based on page-level mapping (**page-level**) [11] and **LazyRTGC** [1]. We implemented our proposed methods, page-level and LazyRTGC on a FlashSim simulator [3]. We use the mtrics of long-tail latency at the 99^{th}, 99.99^{th}, and 99.9999^{th} percentiles and erase count. We use eight real-world workloads (six workloads from FIU [19] and two workloads from Microsoft [19]) and a synthetic one (from filebench [20]) as listed in Table 1. The goal of our work is to reduce long tail latency. In read-intensive workloads, the problem of long tail latency is not severe since GC is rarely invoked. Thus, we used write-intensive workloads in our experiments.

We started simulations with empty contents in the flash-memory model and measured the latency of all the requests for each workload. We use two types of 3D flash-memory systems as listed in Table 2.

Table 1. Workload Characteristics

	Write ratio	Avg. interval [μs]	Avg. request size [KB]
home1	99%	85565	8.08
home2	91%	320548	9.40
home3	99%	1882329	8.26
home4	94%	693651	7.56
webmail	74%	303762	8.00
webmail + online	78%	127184	8.00
RBESQL	82%	11664	57.85
MSNSFS	67%	739	21.67
oltp	99%	84	4.46

Table 2. NAND Flash Memory

	3D 128 Gb [18]	3D 512 Gb [8]
Page size	8KB	16KB
Number of pages/block	384	768
Number of blocks/plane	2731	2874
Number of planes	2	2
Page read time	49 μs	60 μs
Page program time	600 μs	700 μs
Block erase time	4000 μs	3500 μs
Data transfer rate	533 Mbps	1 Gbps

Table 3. States

Previous inter-request interval [μs]	Previous action	Current inter-request interval [μs]
<100	< max action/2	<100
		<500
		■■■
		>100000
	> max action/2	■■■
>100	■■■	■■■
		■■■
	> max action/2	>100000

Table 3 shows the binning for the components of the state. The binning was obtained by a sensitivity analysis on binning choices by varying the numbers of bins, 1~3 and 15~20 for previous and current inter-request intervals, and 1~3 for previous actions, respectively, with an aim to reduce the q-table size, i.e., the number of states while improving the long tail latency.

Considering that the accesses to NAND flash memory take 10~1000μs, e.g., 49μs for read and 600μs for write [18], even though the agent is triggered at every storage access, the runtime overhead of the agent is negligibly small. It is because the agent accesses the q-table (in a small SRAM) at maximum twice and executes a few instructions on the controller chip. Thus, the runtime of the agent is much smaller than the read latency of NAND flash memory.

Table 4. Threshold

Threshold	Value	Remark
T_{GC}	10	Triggering GC
T_{IGC}	3	Stopping intensive GC
T^{A}_{GC}	100	Triggering aggressive GC

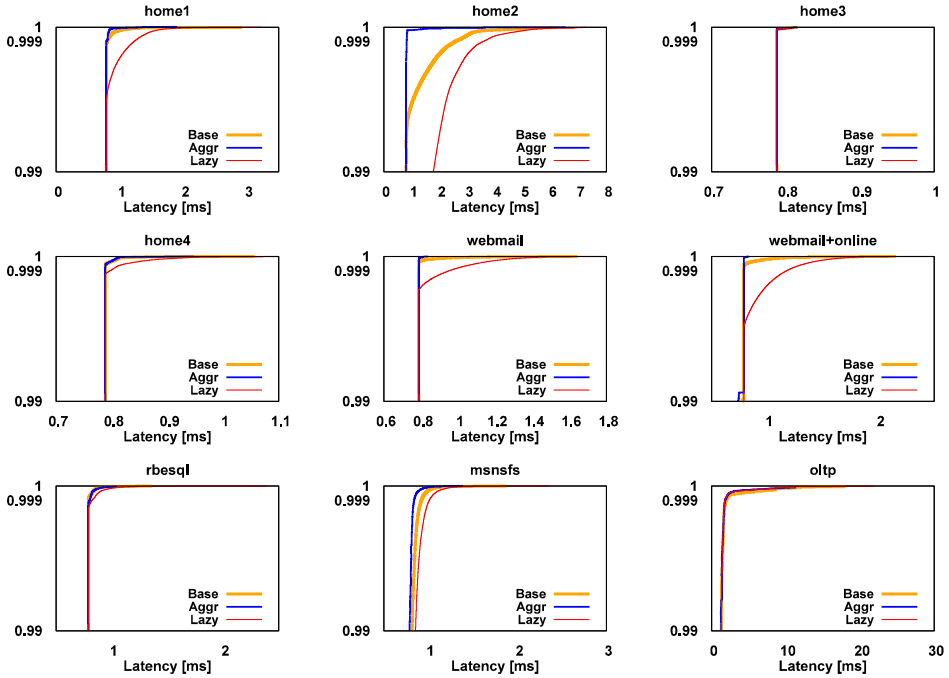

Fig. 7. Comparison of write long-tail latency (512Gb 3D NAND Flash memory).

Table 4 summarizes the thresholds used in our method. We obtained them by conducting a sensitivity analysis with all the storage traces. To improve the generality of our proposed methods, in our future work, we will investigate the feasibility of reducing the number of thresholds by enhancing the RL model, e.g., by introducing the number of free blocks into the states of the agent.

6.2 Results and Discussion

Figure 7 compares the long-tail latency (in CDF) for writes. The figure shows that our proposed methods exhibit better long-tail latency than that using page-level and LazyRTGC. Page-level is not shown in the figure due to too large latency since it does not adopt any optimization to reduce long tail latency. LazyRTGC lies partial GC operations in a lazy manner and shows better latency than page-level.

Latency: Table 5 compares the latency normalized to LazyRTGC on a 512 Gb 3D NAND flash memory. Our baseline method (Base in the table) gives better (smaller) average latency: 0.86x at 99.9999^{th}, 0.94x at 99.99^{th}, and 0.92x at 99^{th} percentile. The gain is a result of the reinforcement learning-assisted action selection. LazyRTGC utilizes a fixed number of partial GC operations. In contrast, our proposed RL-assisted method can adapt to the characteristics of storage behavior,

Table 5. Latency Comparison on 512Gb 3D NAND

Percentile		home1	home2	home3	home4	web+online	webmail	MSNSFS	RBESQL	oltp	AVG
99.9999th	Page	465	239	N/A	962	514	697	9353	2246	33.1	1813
	Lazy	1.00	1.00	N/A	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Base	0.75	0.86	N/A	0.87	0.82	0.89	0.82	0.81	1.10	0.86
	Aggr	0.58	0.90	N/A	0.83	0.35	0.48	0.88	0.92	1.14	0.76
99.99th	Page	769	292	127	1105	679	848	6396	3435	62.9	1823
	Lazy	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Base	0.65	0.90	1.00	0.88	0.66	0.69	0.86	0.84	1.00	0.94
	Aggr	0.50	0.27	1.00	0.88	0.47	0.58	0.84	0.85	1.06	0.71
99th	Page	6.67	3.95	1.00	5.93	6.06	5.79	5077	10.5	2.91	568
	Lazy	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Base	1.00	0.44	1.00	1.00	0.98	1.00	0.95	1.00	0.99	0.92
	Aggr	1.00	0.44	1.00	1.00	0.93	1.00	0.94	1.00	0.98	0.92

thereby providing variable number of partial GC operations to better exploit the idle time, which contributes to reducing the long-tail latency. Our aggressive method (Aggr in the table) gives much smaller latency: 0.76x at 99.9999th, 0.71x at 99.99th, and 0.92x at 99th percentile. This proves that the two aggressive solutions, max-limited early GC triggering and read-initiated GC triggering, are effective in further reducing the long-tail latency.

In particular, the aggressive method gives much better latency in the four workloads: home1, home2, webmail, and webmail + online. These workloads have heavy overwrite traffics distributed across a wide range of addresses. Figure 8 exemplifies the distribution of the write traffics for home1 and home3. As the figure shows, in the case of home1, the overwrites are much stronger than that in home3 (see y axis). In addition, such strong overwrites are more distributed across a wider address range than that in home3.

Such a write behavior in home1 increases the ratio of invalid pages across a large number of blocks, which makes the GC cheaper, i.e., a free block can be obtained for fewer valid page copies. Thus, our aggressive method is effective in home1. However, as shown in Figure 8(b), home3 has weaker overwrite behavior than home1, which makes it difficult for the aggressive method to reclaim the free blocks using fine-grained partial GC.

In Table 5, both the LazyRTGC and our methods give similar latencies in home3 and oltp. In case of home3, the inter-request interval is large as listed in Table 1. In such a case, the GC (and its optimization) does not help in reducing the latency. On the other hand, oltp has very short idle time, i.e., small inter-request interval as listed in Table 1.

Thus, there is little o pportunity to improve the GC. Table 6 compares the latencies in the case of a 128 Gb 3D NAND flash memory. Compared to the results in Table 5, our proposed methods give further reductions, e.g., 0.66x (in Table 6) vs 0.76x (Table 5), compared to the aggressive method at the 99.9999th percentile. This is largely because of the low capacity of the 128 Gb flash memory. The low capacity triggers GC more frequently, which increases the overhead of the GC in the conventional GC method (page-level). In Table 6, our proposed methods are more effective than the LazyRTGC in reducing the GC overhead in such a difficult condition.

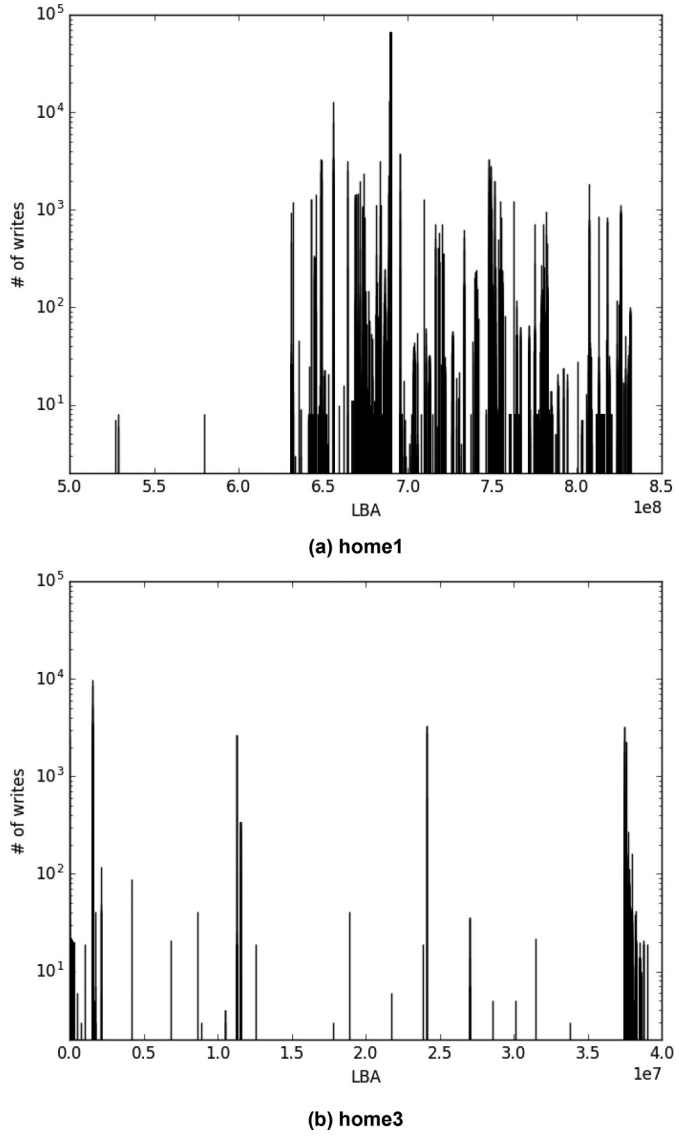

Fig. 8. Distribution of write traffics.

Free block: Figure 9 shows the variation in the number of free blocks over time in the workload home1 under LazyRTGC, and under our baseline and aggressive methods. As shown in the figure, after an initial period, LazyRTGC continues to retain 3 or 4 free blocks, which can lead to frequent GC operations because the number of free blocks is less. Our baseline method manages slightly more number (3–6) of free blocks. Our aggressive method manages significantly more number of free blocks, which helps in reducing the GC operations, thereby contributing to reducing the long-tail latency.

Note that, as mentioned in Section 5.3, our aggressive method increases the number of free blocks only when there are victim blocks having a large ratio of invalid pages. Thus, although the

Table 6. Latency Comparison on 128Gb 3D NAND

Percentile		home1	home2	home3	home4	web+online	webmail	MSNSFS	RBESQL	oltp	AVG
99.9999th	Page	127	99	N/A	190	121	137	1007	717	1677	509
	Lazy	1.00	1.00	N/A	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Base	0.82	0.99	N/A	0.88	0.66	0.81	0.77	0.69	1.28	0.86
	Aggr	0.64	0.54	N/A	0.59	0.26	0.29	0.74	0.80	1.51	0.66
99.99th	Page	181	109	16.3	237	185	198	748	454	16.7	238
	Lazy	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Base	0.62	0.93	1.00	0.80	0.77	0.74	0.69	0.37	1.21	0.79
	Aggr	0.62	0.36	1.00	0.65	0.39	0.40	0.55	0.48	1.29	0.64
99th	Page	3.07	1.68	1.00	3.73	2.20	2.94	354	371	2.19	82.4
	Lazy	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Base	0.74	0.54	1.00	1.00	0.56	0.76	0.39	0.33	1.90	0.80
	Aggr	0.74	0.37	1.00	1.00	0.53	0.76	0.38	0.33	0.92	0.67

Table 7. Erase Count Comparison on 512Gb 3D NAND

	home1	home2	home3	home4	web+online	webmail	MSNSFS	RBESQL	oltp	AVG
Page	1.92	1.33	1.50	1.83	1.59	1.69	10.9	2.07	1.67	2.72
Lazy	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Base	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Aggr	0.91	1.00	1.02	1.01	1.02	1.03	1.03	0.95	1.01	1.00

aggressive method manages a significantly more number of free blocks than LazyRTGC, it does not have a negative impact on the erase count, as demonstrated later in this section.

Erase Count: Tables 7 and 8 compare the erase counts (normalized to LazyRTGC) on 512 Gb and 128 Gb 3D NAND flash-memory systems, respectively. From Tables 7 and 8, it is clear that our proposed aggressive method and LazyRTGC give similar erase counts while the page-level gives a higher erase count because of the block-level GC.

RL related Analysis: In order to evaluate the robustness of our method, we measured the latency of ten executions of each trace. Tables 9 and 10 show that the results of proposed method are consistent having a very small standard deviation of latency, 3.8% of the average normalized latency.

We evaluated the utilization of q-table entries for each workload. In the analysis, we found that the average utilization is 79% and there is possibility of further improvement by adjusting the q-table size to each workload, which is left for our future work.

Average application performance: It is important to evaluate the impact of our proposed method on average application performance. Since we did trace-based experiments, the

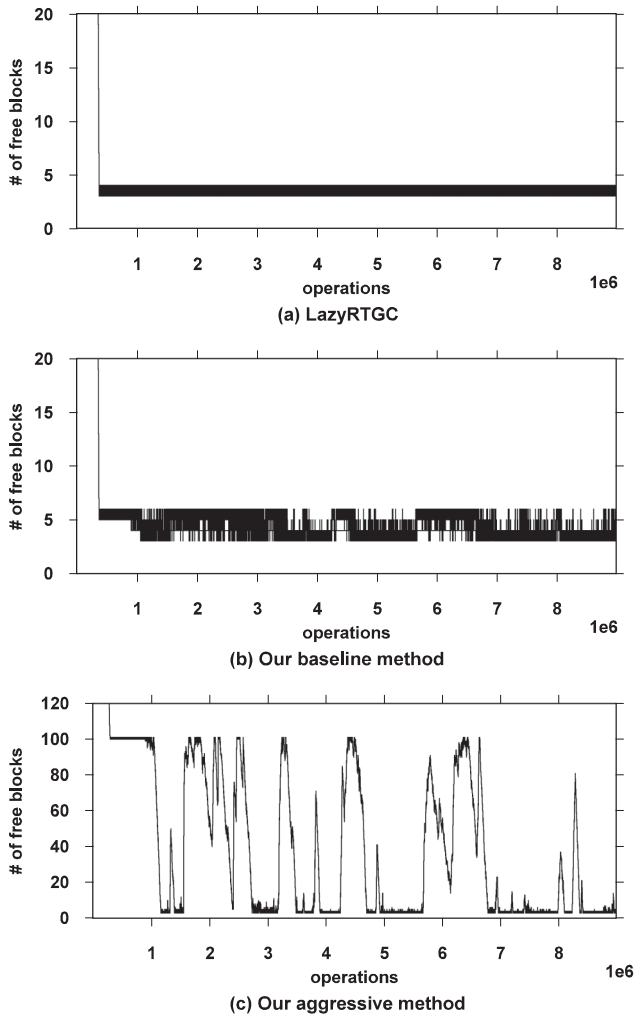

Fig. 9. Comparison of number of free blocks.

Table 8. Erase Count Comparison on 128Gb 3D NAND

	home1	home2	home3	home4	web+online	webmail	MSNSFS	RBESQL	oltp	AVG
Page	1.26	1.16	1.26	1.56	1.19	1.41	0.56	0.63	1.81	1.20
Lazy	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Base	0.92	0.97	1.00	1.01	0.98	1.01	0.93	0.97	3.15	1.21
Aggr	0.93	1.00	1.06	1.12	1.01	1.1	0.94	0.98	1.14	1.03

Table 9. Standard Deviation of Normalized Latency on 512Gb 3D NAND

Percentile		home1	home2	home3	home4	web+online	webmail	MSNSFS	RBESQL	oltp	AVERAGE
99.9999th	Base	0.055	0.051	0.000	0.031	0.030	0.043	0.038	0.042	0.055	0.038
	Aggr	0.026	0.049	0.000	0.027	0.002	0.002	0.026	0.029	0.008	0.019
99.99th	Base	0.017	0.072	0.000	0.006	0.015	0.022	0.003	0.002	0.003	0.016
	Aggr	0.014	0.052	0.000	0.000	0.000	0.000	0.003	0.002	0.005	0.008
99th	Base	0.000	0.000	0.000	0.000	0.004	0.000	0.000	0.000	0.003	0.001
	Aggr	0.005	0.000	0.000	0.000	0.004	0.000	0.000	0.000	0.001	0.001

Table 10. Standard Deviation of Normalized Latency on 128Gb 3D NAND

Percentile		home1	home2	home3	home4	web+online	webmail	MSNSFS	RBESQL	oltp	AVERAGE
99.9999th	Base	0.047	0.047	0.000	0.036	0.033	0.053	0.027	0.046	0.044	0.037
	Aggr	0.025	0.111	0.000	0.048	0.018	0.000	0.025	0.029	0.000	0.028
99.99th	Base	0.011	0.035	0.000	0.027	0.006	0.013	0.005	0.004	0.065	0.018
	Aggr	0.006	0.070	0.000	0.016	0.000	0.000	0.004	0.005	0.000	0.011
99th	Base	0.000	0.009	0.000	0.000	0.000	0.000	0.000	0.000	0.009	0.002
	Aggr	0.000	0.000	0.000	0.000	0.000	0.000	0.001	0.000	0.000	0.000

average latency of request is considered to be correlated with average application performance. Our
experiments (the corresponding results of which are omitted due to page limit) show that, the average latency of our proposed baseline and aggressive methods is slightly better than that of the existing method, Lazy RTGC. Thus, we can state that our proposed methods improve the long tail latency without degrading the average application performance.

Long trace experiment: We also evaluated our proposed method with longer traces by stitching the original traces. Our experiments show that, in the long trace cases, our proposed method outperforms the existing one, lazy as in the case of short ones.

Simple prediction method: Our problem of reducing long tail latency could be addressed by existing, possibly simpler, alternatives such as those based on time series prediction. In our experiments, we did a quantitative comparison with GC methods based on two typical methods of predicting the inter-request interval with moving average and exponential smoothing, respectively. Tables 11 and 12 show that our proposed method constantly outperforms them. It is because our method manages history and learns appropriate actions in a more fine-grained manner using the q-table.

In summary, the experimental results show that the LazyRTGC does not fully utilize the idle time available in the storage workload. In contrast, our baseline method can better exploit the idle time because of the reinforcement learning-based GC. In addition, our aggressive method helps in

Table 11. Latency Comparison of Simple Prediction Method on 512Gb 3D NAND

Percentile		home1	home2	home3	home4	web+online	webmail	MSNSFS	RBESQL	oltp	AVERAGE
99.9999th	Aggr	0.58	0.90	N/A	0.83	0.35	0.48	0.88	0.92	1.14	0.76
	Mov10	0.97	1.01	N/A	1.01	0.67	0.60	0.97	0.70	1.56	0.94
	Mov100	1.11	1.02	N/A	1.01	0.90	0.95	0.99	0.76	1.36	1.01
	Exp0.1	0.87	1.02	N/A	0.93	0.84	0.90	0.97	0.63	1.56	0.97
	Exp0.3	0.97	1.01	N/A	1.00	0.65	0.69	0.99	0.59	1.62	0.94
99.99th	Aggr	0.50	0.27	1.00	0.88	0.47	0.58	0.84	0.85	1.06	0.71
	Mov10	0.61	0.98	1.00	0.95	0.58	0.71	0.98	0.84	1.12	0.86
	Mov100	1.00	1.05	1.00	0.95	0.97	0.99	0.97	0.82	1.12	0.99
	Exp0.1	0.84	1.04	1.00	0.94	0.84	0.92	0.99	0.82	1.12	0.95
	Exp0.3	0.66	0.99	1.00	0.95	0.64	0.78	0.98	0.82	1.08	0.88
99th	Aggr	1.00	0.44	1.00	1.00	0.93	1.00	0.94	1.00	0.98	0.92
	Mov10	1.00	0.75	1.00	1.00	1.00	1.00	1.00	1.00	0.99	0.97
	Mov100	1.00	0.75	1.00	1.00	1.00	1.00	0.99	1.00	0.99	0.97
	Exp0.1	1.00	0.71	1.00	1.00	1.00	1.00	1.00	1.00	0.99	0.97
	Exp0.3	1.00	0.72	1.00	1.00	1.00	1.00	1.00	1.00	0.98	0.97

Table 12. Latency Comparison of Simple Prediction Method on 128Gb 3D NAND

Percentile		home1	home2	home3	home4	web+online	webmail	MSNSFS	RBESQL	oltp	AVERAGE
99.9999th	Aggr	0.64	0.54	N/A	0.59	0.26	0.29	0.74	0.80	1.51	0.66
	Mov10	0.93	0.99	N/A	0.94	0.76	0.84	0.79	0.64	1.50	0.92
	Mov100	1.53	3.51	N/A	1.41	1.05	1.09	0.72	0.68	1.41	1.43
	Exp0.1	1.16	1.08	N/A	1.21	1.01	1.07	0.72	0.61	1.59	1.06
	Exp0.3	0.96	0.99	N/A	1.04	0.74	1.00	0.84	0.54	1.66	0.97
99.99th	Aggr	0.62	0.36	1.00	0.65	0.39	0.40	0.55	0.48	1.29	0.64
	Mov10	0.77	0.95	1.00	0.98	0.81	0.85	0.86	0.63	1.15	0.89
	Mov100	0.99	1.68	1.00	1.03	1.08	1.08	0.85	0.57	1.32	1.07
	Exp0.1	0.94	1.01	1.00	1.03	1.07	1.06	0.86	0.56	1.13	0.96
	Exp0.3	0.85	0.96	1.00	0.98	0.98	1.02	0.86	0.56	1.13	0.93
99th	Aggr	0.74	0.37	1.00	1.00	0.53	0.76	0.38	0.33	0.92	0.67
	Mov10	0.87	0.83	1.00	1.00	0.84	0.94	0.93	1.00	0.92	0.93
	Mov100	1.07	0.86	1.00	1.00	1.09	1.11	0.93	1.00	0.92	1.00
	Exp0.1	1.00	0.86	1.00	1.00	0.97	0.99	0.93	1.00	0.93	0.96
	Exp0.3	0.89	0.84	1.00	1.00	0.83	0.89	0.93	1.00	0.92	0.92

further reducing the long-tail latency by (1) preparing free blocks with frequent small fine-grained partial GCs, which helps in reducing the frequency of triggering the GC operations and stalling the subsequent requests, and (2) hiding the GC operation by exploiting the idle time based on the reinforcement learning. Consequently, as presented in Tables 5 and 6, our proposed aggressive method helps in reducing the long-tail latency by 29–36% at the 99.99^{th} percentile for the two flash-storage devices.

7 CONCLUSION

In this paper, we addressed the problem of long-tail latency in NAND flash memory-based storage systems and proposed a reinforcement learning-assisted garbage collection technique, which learns the storage access behavior online and determines the number of GC operations to exploit the idle time. We also presented aggressive methods, which helps in further reducing the long-tail latency by aggressively performing fine-grained GC operations. We evaluated our proposed methods with eight real-world workloads on two 3D NAND flash memory storages. We managed to reduce the long-tail latency by 29–36%. We expect that such a reduction is beneficial for real-time embedded systems and quality-critical server systems.

REFERENCES

[1] 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

[2] Zhiwei Qin, Yi Wang, Duo Liu, and Zili Shao. 2012. Real-time flash translation layer for NAND flash memory storage systems. In *Symp. IEEE 18th Real Time and Embedded Technology and Application.* 35–44. DOI:10.1109/RTAS.2012.27

[3] Youngjae Kim, Brendan Tauras, Aayush Gupta, and Bhuvan Urgaonkar. 2009. FlashSim: A simulator for NAND flash-based solid-state drives. In *Proceeding. 1st International Conference in Advances in System Simulation (SIMUL).* 125–131. DOI:10.1109/SIMUL.2009.17

[4] Yi Wang, Zhiwei Qin, Renhai Chen, Zili Shao, Qixin Wang, Shuai Li, and Laurence T. Yang. 2016. A real-time flash translation layer for NAND flash memory storage systems. *IEEE Trans. on Multi-scale Computer Systems.* 2. 1. 17–29. DOI: 10.1109/TMSCS.2016.2516015

[5] 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, New York, NY, USA, 19–24. DOI: http://dx.doi.org/10.1145/1450135.1450141

[6] Hsin-Yu Chang, Chien-Chung Ho, Yuan-Hao Chang, Yu-Ming Chang, and Tei-Wei Kuo. 2016. How to enable software isolation and boost system performance with sub-block erase over 3D flash memory. In *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES'16).* ACM, New York, NY, USA, Article 6, 10 pages. DOI:https://doi.org/10.1145/2968456.2968475

[7] Tseng-Yi Chen, Yuan-Hao Chang, Chien-Chung Ho, and Shuo-Han Chen. 2016. Enabling sub-blocks erase management to boost the performance of 3D NAND flash memory. In *Proceedings of the 53rd Annual Design Automation Conference (DAC'16).* ACM, New York, NY, USA, Article 92, 6 pages. DOI:10.1145/2897937.2898018

[8] Chulbum Kim, Ji-Ho Cho, Woopyo Jeong, Il-han Park, Hyun-Wook Park, Doo-Hyun Kim, Daewoon Kang, Sunghoon Lee, Ji-Sang Lee, Wontae Kim, Jiyoon Park, Yang-lo Ahn, Jiyoung Lee, Jong-hoon Lee, Seungbum Kim, Hyun-Jun Yoon, Jaedoeg Yu, Nayoung Choi, Yelim Kwon, Nahyun Kim, Hwajun Jang, Jonghoon Park, Seunghwan Song, Yongha Park, Jinbae Bang, Sangki Hong, Byunghoon Jeong, Hyun-Jin Kim, Chunan Lee, Young-Sun Min, Inryul Lee, In-Mo Kim, Sung-Hoon Kim, Dongkyu Yoon, Ki-Sung Kim, Youngdon Choi, Moosung Kim, Hyunggon Kim, Pansuk Kwak, Jeong-Don Ihm, Dae-Seok Byeon, Jin-yub Lee, Ki-Tae Park, and Kye-hyun Kyung. 2017. A 512Gb 3b/cell 64-stacked WL 3D V-NAND flash memory. In *Proceeding. 2017 IEEE International Solid-State Circuits Conference (ISSCC).* 202–204. DOI:10.1109/ISSCC.2017.7870331

[9] Sungdae Choi, Duckju Kim, Sungwook Choi, Byungryul Kim, Sunghyun Jung, Kichang Chun, Namkyeong Kim, Wanseob Lee, Taisik Shin, Hyunjong Jin, Hyunchul Cho, Sunghoon Ahn, Yonghwan Hong, Ingon Yang, Byoungyoung Kim, Pilseon Yoo, Youngdon Jung, Jinwoo Lee, Jaehyeon Shin, Taeyun Kim, Kunwoo Park, and Jinwoong Kim. 2014. A 93.4mm2 64Gb MLC NAND-flash memory with 16nm CMOS technology. In *Proceeding. 2014 IEEE International Solid-State Circuits Conference (ISSCC).* 328–330. DOI:10.1109/ISSCC.2014.6757455

[10] Richard S. Sutton and Andrew G. Barto. 1998. *Introduction to Reinforcement Learning (1st ed.)*. MIT Press, Cambridge, MA, USA.

[11] 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, New York, NY, USA, 229–240. DOI: http://dx.doi.org/10.1145/1508244.1508271

[12] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. 2015. Operating Systems: Three easy pieces. *Arpaci-Dusseau Books*. LLC.

[13] Qingsong Wei, Bozhao Gong, Suraj Pathak, Bharadwaj Veeravalli, LingFang Zeng, and Kanzo Okada. 2011. WAFTL: A workload adaptive flash translation layer with data partition. In *Proceedings of the IEEE 217th Symposium on Mass Storage Systems and Technologies (MSST)*. DOI:10.1109/MSST.2011.5937217

[14] Kee-Hoon Jang and Tae Hee Han. 2010. Efficient garbage collection policy and block management method for NAND flash memory. In *Proceeding. 2010 2nd International Conference on Mechanical and Electronics Engineering (ICMEE)*. 327–331. DOI:10.1109/ICMEE.2010.5558538

[15] Longzhe Han, Yeonseung Ryu, and Keunsoo Yim. 2006. CATA: A garbage collection scheme for flash memory file systems. In *Proceeding. International Conference on Ubiquitous Intelligence and Computing (UIC)*. 103–112. DOI:10.1007/11833529_11

[16] Mingwei Lin and Shuyu Chen. 2013. Efficient and intelligent garbage collection policy for NAND flash-based consumer electronics. *IEEE Trans. On Consumer Electronics*. 2, 3. 538–543. DOI:10.1109/TCE.2013.6626235

[17] 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 (November 2004), 837–863. DOI:10.1145/1027794.1027801

[18] Samsung Electronics. 2014. Samsung V-NAND technology white paper. http://www.samsung.com/us/business/oem-solutions/pdfs/V-NAND_technology_WP.pdf.

[19] SNIA IOTTA: Storage Networking Industry Association's Input/Output Traces, Tools and Analysis. http://iotta.snia.org.

[20] Filebench. https://github.com/filebench/filebench/wiki.

[21] Top considerations for Enterprise SSDs a primer. 2016. http://ingrammicrosystemarchitechs.com/wp-content/uploads/2015/10/White-Paper-Top-Considerations-for-Enterprise-SSDs-WP30.pdf.

[22] Jeffrey Dean and Luiz Andre Barroso. 2013. The tail at scale. *Communication of ACM* 56. 74–80. DOI: http://dx.doi.org/10.1145/2408776.2408794

[23] Christian Monzio Compagnoni, Akira Goda, Alessandro S. Spinelli, Peter Feeley, Andrea L. Lacaita, and Angelo Visconti. 2017. Reviewing the evolution of the NAND flash technology. In *Proceedings of the IEEE*. 99, 1–25. DOI:10.1109/JPROC.2017.2665781

[24] Rino Micheloni, Seiichi Aritome, and Luca Crippa. 2017. Array architectures for 3-D NAND flash memories. In *Proceedings of the IEEE*. 99, 1–16. DOI:10.1109/JPROC.2017.2697000

[25] Engin Ipek, Onur Mutlu, Jose F. Martinez, and Rich Caruana. 2008. Self-optimizing memory controllers: A reinforcement learning approach. In *Proceeding. 2008 IEEE International Symposium on Computer Architecture (ISCA'08)*. DOI:10.1109/ISCA.2008.21

[26] Yanzhi Wang, Qing Xie, Ahmed Ammari, and Massoud Pedram. 2011. Deriving a near-optimal power management policy using model-free reinforcement learning and bayesian classification. In *Proceedings of the 48th Design Automation Conference (DAC'11)*. ACM, New York, NY, USA, 41–46. DOI: http://dx.doi.org/10.1145/2024724.2024735

[27] Leeor Peled, Shie Mannor, Uri Weiser, and Yoav Etsion. 2015. Semantic locality and context-based prefetching using reinforcement learning. In *Proceeding. ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, Portland. 285–297. DOI:10.1145/2749469.2749473

Received April 2017; revised June 2017; accepted June 2017