Mitigating Negative Impacts of Read Disturb in SSDs

JUN LI, BOWEN HUANG, ZHIBING SHA, ZHIGANG CAI, and JIANWEI LIAO, Southwest University of China, China
BALAZS GEROFI and YUTAKA ISHIKAWA, RIKEN Center for Computational Science, RIKEN, Japan

Read disturb is a circuit-level noise in solid-state drives (SSDs), which may corrupt existing data in SSD blocks and then cause high read error rate and longer read latency. The approach of read refresh is commonly used to avoid read disturb errors by periodically migrating the hot read data to other free blocks, but it places considerable negative impacts on I/O (Input/Output) responsiveness. This article proposes scheduling approaches on write data and read refresh operations, to mitigate the negative effects caused by read disturb. To be specific, we first construct a model to classify SSD blocks into two categories according to the estimated read error rate by referring to the factors of block's P/E (Program/Erase) cycle and the accumulated read count to the block. Then, the data being intensively read will be redirected to the block having a small read error rate, as it is not sensitive to read disturb even though the data will be heavily requested. Moreover, we take advantage of reinforcement learning to predict the idle interval between two I/O requests for purposely conducting (partial) read refresh operations. As a result, it is able to minimize negative impacts toward subsequent incoming I/O requests and to ensure I/O responsiveness. Through a series of emulation tests on several realistic disk traces, we demonstrate that the proposed mechanisms can noticeably yield performance improvements on the metrics of read error rate and I/O latency.

CCS Concepts: • **Computer systems organization → Reliability**;

Additional Key Words and Phrases: Solid-state drive (SSD), read disturb, read refresh, scheduling, read errors

ACM Reference format:
Jun Li, Bowen Huang, Zhibing Sha, Zhigang Cai, Jianwei Liao, Balazs Gerofi, and Yutaka Ishikawa. 2020. Mitigating Negative Impacts of Read Disturb in SSDs. *ACM Trans. Des. Autom. Electron. Syst.* 26, 1, Article 3 (September 2020), 24 pages.
https://doi.org/10.1145/3410332

1 INTRODUCTION

NAND (Not AND) flash memory-based solid-state drives (SSDs) have a non-volatile nature and are then widely leveraged in digital devices. To be specific, SSDs are commonly featured with

This work was partially supported by National Natural Science Foundation of China (No. 61872299), Natural Science Foundation Project of CQ CSTC (No. CSTC2018jcyjAX0552), Hunan Provincial Natural Science Foundation of China (No. 2018JJ2309), and the Opening Project of State Key Laboratory for and Novel Software Technology (No. KFKT2019B06).
Authors' addresses: J. Li, B. Huang, Z. Sha, and Z. Cai, Southwest University of China, Chongqing, China, 400715; emails: lijun19991111@126.com, {minwan530755, shzb171318515}@163.com, czg@swu.edu.cn; J. Liao (corresponding author), Southwest University of China, Chongqing, China, 400715 and State Key Lab. for Novel Software Technology, Nanjing University, P.R. China; email: liaojianwei@il.is.s.u-tokyo.ac.jp; B. Gerofi and Y. Ishikawa, RIKEN Center for Computational Science, RIKEN, Wako, Japan, 351-0198; emails: bgerofi@gmail.com, ishikawa@is.s.u-tokyo.ac.jp.

1084-4309/2020/09-ART3 $15.00
https://doi.org/10.1145/3410332

small size, high performance, random-access performance and low energy consumption [1–3]. Due to many design methodologies to optimize the performances of SSDs, they will develop into the dominant secondary storage in the coming years [4, 5]. In order to better cut down the per-unit price of SSDs, the feature size of NAND flash memory cells reaches the limit of the 10nm level. Consequently, flash density increases are then driven by Trinary-level cell (TLC) or even Quad-level cell (QLC) (3 or 4 bits/cell) combined with vertical stacking of NAND memory planes. However, the decrease in endurance and the increase in bit error rates accompanying with the feature size shrinking are now becoming the issues to be reckoned with [6–8]. That is to say, as NAND flash memory cell capacity increases, it becomes more susceptible to diversified types of circuit-level noises [6], including program/erase (P/E) cycle noises [10, 11], retention noises [10, 12], cell-to-cell program interference noises [10, 13], and read disturb noises [14, 15]. Thus, dealing with such noises and then ensuring reliability of flash memory becomes a challenging task.

Specially, read disturb is an unexpected phenomenon in NAND flash memory, where reading data from a flash page can impact the threshold voltages of other (unread) pages in the same block. It is predicted to critically impact on the reliability of high-level-cell flash memory [27]. And it has become a growing source of flash errors, and the situation gets worse in a compact NAND memory [6, 9]. For example, read disturb errors appear after an average of one million reads onto a single flash block in SLC NAND memory (1 bits/cell) [15, 16], and the read limit becomes 100K in the first-generation of MLC NAND memory (2 bits/cell) [16]. But for TLC devices, this read threshold is greatly decreased to less than 40K [19]. This is because the geometry of a cell shrinks, and the cross-coupling voltage noise gets more intensive, which may consequently bring about more read disturb errors.

More importantly, it has been verified that even in the same type of SSD device, the blocks having varied program/erase (P/E) cycles and read counts may result in different levels of Raw Bit Error Rate (RBER) [23]. For example, if two TLC SSD blocks have P/E cycles of 1,001 and 6,001, respectively, reading 5,000 times onto both of them will correspondingly cause 0.00121 and 0.00301 of RBER [23].

Conventionally, error correction codes (ECCs) have been used in SSDs to cover the RBER issue caused by various factors, such as read disturb. For example, low-density parity-check (LDPC) codes, which is an advanced ECC, has a stronger error correction capability than traditional ECCs (e.g., BCH (Bose–Chaudhuri–Hocquenghem)) [20, 21]. Specifically, LDPC can cover the error through read retries at the cost of read latencies, while the RBER is in a correctable capacity of LDPC. But note that LDPC will exaggerate read disturb as they intend to get correct data through read retries [24, 25].

The accumulated read operations on SSD blocks may lead to more read disturb errors, which must affect the reliability of SSDs by suffering from uncorrectable bit errors beyond the capacity of ECCs. To overcome this problem, the mechanism of read refresh (RR) has been introduced to mitigate negative effects caused by read disturb [22]. There are two steps in the process of read refresh: ① the valid pages of the target block should be migrated to another available block, called as *page moves*; ② the target block is then erased for getting rid of the accumulated read count, called as *erase*. In other words, a read refresh operation is expected to be triggered once the read count of an SSD block reaches a pre-defined threshold. It is true that the read count limits of read disturb on SSD devices are different while the cell density varies, so that the read refresh threshold should be correspondingly adjusted. Then, after moving the hot read data to another free block in a read refresh process, the accumulated read count to the data is reset to 0. That is, the RBER can always be limited under the maximum ECC strength of LDPC, and the reliability and availability of SSDs can be guaranteed. Otherwise, if RR is not implemented as a background method, it is

inevitable to read the data having some uncorrectable bit errors when the RBER is beyond the error correction capacity of ECCs.

However, carrying out read refresh operations exclusively occupies the SSD resources (e.g., channel and chip), which must delay subsequent I/O requests. To reduce the number of RR operations, Liu et al. [29] proposed a novel mechanism by taking advantage of shadow blocks, which contain a number of invalid data pages or free pages, as both kind of pages are immune to read disturb. Then, the frequently read data can be moved from one page to another free page within the shadow blocks to avoid read refresh. Wu et al. [30] proposed an approach of adaptive cell bit-density with in-place reprogramming. This method reprograms the old Most Significant Bit (MSB) pages and transforms TLC blocks into MLC ones, as the read limit for triggering read refresh operations on MLC blocks is much greater than the read limit on the original TLC blocks. Consequently, the total number of read refresh can be greatly cut down. But, these proposals fail to take the factor of block attributes, such as the P/E cycle, into account, and they will definitely carry out RR if needed, regardless of the intensity of forthcoming I/O requests.

Besides, different from the process of garbage collection in SSDs, the number of *page moves* in the read refresh process is relatively large, which may block forthcoming I/O requests for a longer time interval, even compared with *erase*. For meeting the requirement of ensuring I/O responsiveness, we argue that two steps of read refresh are not necessarily dealt with together, and it can handle a partial RR operation (i.e., *page moves* or *erase*) in a time slot. To put it from another angle, we can first migrate the hot read data (or some of them) to other free blocks for resetting the accumulated read count on such data in an idle time interval between two I/O requests. Then, *erase* can be fulfilled if there is another idle time slot, or the free space of SSDs is not enough at late stages (i.e., garbage collection is triggered).

To further cut down the negative effects of read disturb, this article first discusses the driven factors of read disturb in SSDs and then builds a mathematical model for objectively assessing the read error rate caused by read disturb, with respect to a specific SSD block.[1] As a result, it proposes a scheduling approach on write requests to dispatch the future hot read data and the future cold read data onto SSD blocks having different read error rates. Moreover, we introduce a reinforcement learning-based read refresh scheduling method to properly dispatch (partial) read refresh operations in idle time intervals between I/O requests. In summary, it makes the following three contributions:

(1) We systematically analyze the impact factors of read disturb, including the number of block P/E cycles and accumulated block reads. Then, we build an empirical model to estimate the read error rate of the given block by referring to the aforementioned two SSD nature factors. After that, we propose a write scheduling approach to map future hot read data onto the SSD blocks having a small read error rate.
(2) We predict idle time intervals between two I/O requests by taking advantage of reinforcement learning, and then decide how and what partial read refresh operations should be carried out in the idle time period. Thus, it can minimize the side-effects of read refresh on the indicator of I/O responsiveness of subsequent I/O requests.
(3) We offer preliminary evaluation on several disk traces of real-world applications. As measurements indicate, our proposal can afford a better performance improvement on the metrics of average read latency, read error rate, and I/O long-tail latency.

The rest of the article is organized as follows. Section 2 introduces the background knowledge and related work on read disturb. Section 3 describes the comprehensive model for estimating read error rate, as well as the proposed write scheduling approach. The reinforcement

[1]We take the TLC SSDs as our target devices in building the model for assessing the read error rate of SSD blocks.

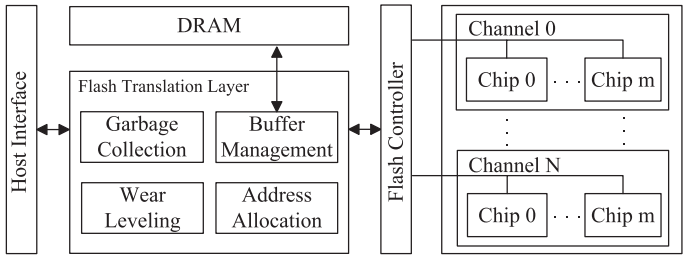

Fig. 1. The internal architectural overview in SSDs.

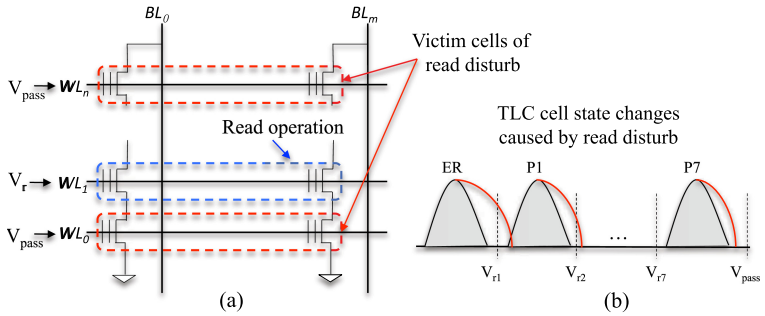

Fig. 2. (a) Voltage settings during a read operation and read disturb. (b) Reference voltage (V_{ri}) and read pass voltage (V_{pass}) of a TLC cell [19].

learning-based read refresh scheduling scheme is presented in Section 4. Section 5 shows the evaluation methodology and reports the experimental results. Finally, the article is concluded in Section 6.

2 BACKGROUND AND RELATED WORK

2.1 SSD Architecture and Read Disturb

Figure 1 shows an internal architectural overview of SSDs, including software and hardware components. As seen, the main software layer is Flash Translation Layer (FTL), which takes charge of translating logical address into physical address that flash memory can identify. Furthermore, it supports garbage collection, wear leveling, and buffer management. Garbage collection is carried out to reclaim the space occupied by the invalid data (i.e., outdated pages) due to the out-place updates, when the available capacity of SSD may be lower than the threshold. Since each SSD block affords a limited number of erases, it becomes critical to take advantage of the mechanism of wear-leveling, for extending the lifespan of flash memory by uniformly distributing erases across all blocks. The buffer management controls the dynamic random access memory (DRAM), which stores data structures of the address mapping table.

As discussed, read disturb is a circuit-level noise in NAND-based memory, which is induced by read operations [12, 19, 30]. Figure 2(a) shows the voltage settings in the case of dealing with a read operation. As seen, a reference voltage of V_r is applied to the target word line of WL_1; meanwhile a large read pass voltage of V_{pass} is imposed to other word lines. Therefore, because of the pass voltage, a read operation poses an impact to the cells of other word lines in the same block.

Nevertheless, applying a high read pass voltage to victim word lines shifts cell threshold voltages through electron injection to floating gates of cells [12]. Though a single read operation does not modify the neighboring cell data immediately, the relevant cell data will be eventually altered when

the side-effect is accumulated by repetitive read operations [19]. Figure 2(b) illustrates an example of cell data change resulted by read disturb. In the case, the multiple high read pass voltage results in the fact that the electrons consistently slowly inject in cells through floating gates; thus, part of states of *ER* and *P1* overlap. Therefore, the reference voltage of V_{r1} fails to clearly identify the states of (disturbed) *ER* and *P1*, which causes a read error and thus expects re-read tries for abstracting the correct data.

2.2 Related Work

There are a number of research efforts against read disturb, which can be generally classified as hardware-based and software-based mechanisms. Cai et al. [17] proposed learning the minimum pass-through voltage for each SSD block to dynamically tune the pass-through voltage on a per-block basis for minimizing read disturb errors. Zambelli et al. [18] characterized the different behaviors of TLC NAND Flash under uniform and concentrated read disturb, which can speculate the implications of the workload usage model on the reliability of enterprise Solid State Drives. Ha et al. [19] disclosed that read disturb is positively correlated with V_{pass} and the duration of imposing V_{pass}. Then, they proposed to write read-hot data using narrow threshold voltage levels, so that such data can be read by leveraging a low V_{pass} within a short interval. However, this approach limits write performance.

ECCs have been adopted by modern SSDs to correct the RBER. The advanced ECC scheme of LDPC can cover RBER through the LDPC soft decision at the cost of read retries. To improve the performance of LDPC-enabled SSDs, Li et al. [20] introduced a process variation aware read performance improvement for LDPC-based SSDs, by relocating the read-hot data to the block with the high reliability. Based on this work, Li et al. [21] further addressed the conflicts between the read performance and lifetime improvement, though limiting the amount of high reliable blocks occupied by read-hot data for balancing block wear evenness and read responsiveness. However, if the accumulated read count reaches the maximum read limit, errors may accumulate to the level beyond the capabilities of ECC.

Therefore, some software-based techniques have also been designed to mitigate the negative effects of read disturb, though preventively relocating the hot read data. In order to ward off corrupting existing data resulted by read disturb, Werner et al. [22] proposed to preventively relocate the data to other blocks if the original host blocks have reached a read disturb limit. That is to say, for avoiding data corruption by read disturbs, the process of RR is expected in partially read-disturbed blocks [19, 26].

Ha et al. [28] introduced a new read disturb-aware FTL, called *RedFTL*, which manages a portion of blocks as shadow blocks. It exclusively stores a few read-hot pages replica in each shadow block, for avoiding the hot data access in the same block and reducing the number of read refresh operations. But, this design introduces longer tail I/O latency because of data replication and synchronization.

Liu et al. [29] further advanced *Read Leveling*, which takes advantage of shadow blocks for managing the hot read data. Specifically, besides purposely keeping the hot read data, the shadow blocks also contain a number of invalid data pages or free pages, as both kinds of pages are immune to read disturb. As a result, it is able to decrease the frequency of read refresh on these shadow blocks, though they hold some hot read data. But note that *Read Leveling* does not perform well for utilizing the storage space in shadow blocks, which limits the improvements on read latency and read refresh cycles.

Wu et al. [30] proposed an adaptive cell bit-density with in-place reprogramming (IPR) for TLC devices, called *IPR*. To be specific, when the read-hot block reaches the read limits of TLC blocks, it migrates the data in *MSB* pages of block to other read-hot blocks, and then reprograms the old

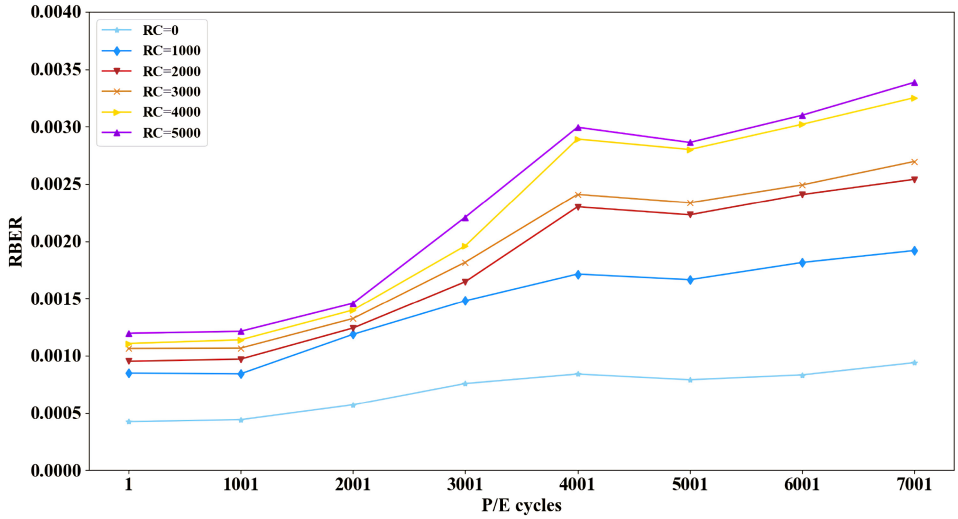

Fig. 3. The RBER of P/E cycles and read counts in TLC devices [23]. Note that RC in the legend indicates the read count.

MSB pages to make the original TLC block becoming an MLC block. As reported, the read limit of low-density block is thus from *10,000* (i.e., in TLC blocks) into *100,000* (i.e., in MLC blocks) after reprogramming. That is to say, the number of read refresh can be consequently cut down, for mitigating negative effects of read disturb. However, *IPR* wastes one third of space in the low-density block, and further increases the pressure of garbage collection.

In fact, read disturb does also exist other kinds of storage devices, such as STT-MRAM [31], challenging the reliability of these devices. To relieve side-effects of read disturb in STT-MRAM, Na et al. [46] proposed a half-pulse-width read disturbance scheme that is capable of significantly improving the read disturbance margin without sacrificing the sensing margin, speed, or energy efficiency by adopting the dual-stage sensing at the cost of additional bit line selection and source line multiplexers. Cheshmikhani et al. [31] proposed a Read Error Accumulation Preventer cache to prevent the accumulation of read disturbance in cache blocks, by performing ECC checking for the requested block and all other blocks within the cache set that have been read in a concurrent manner.

On the other side, a number of machine learning methods have been introduced to direct SSD optimization. Wu et al. [36] introduced a reinforcement learning based I/O merging approach for achieving better I/O performance in SSDs. It can adaptively perform I/O merging according to the different (learned) I/O patterns. Kang et al. [35] presented a reinforcement learning-assisted garbage collection (GC) scheme to reduce the negative effects caused by GC operations. Specifically, it analyzes the history information of I/O requests to define the current state. Besides, it can obtain a reward according to the I/O response time after a specific GC action to further offer a feedback for refining the pairs of state-action maintained by the model. According to the (nearly) fixed state-action pairs, the fit garbage collection action can be performed in the idle time interval between two I/O requests by referring to the current state.

2.3 Motivations

We have surveyed the factors impacting the read error rate in SSDs resulted by read disturb. Figure 3 illustrates that the blocks of TLC SSDs having different erase counts suffer from read

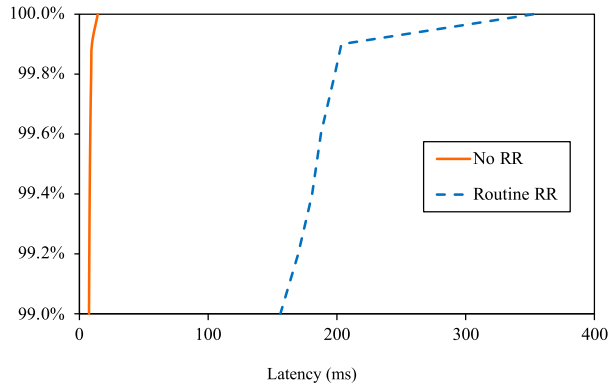

Fig. 4. The long tail latency of running *websearch_1* with or without routine read refresh (assuming no uncorrectable errors).

disturb at varied levels. In addition, the accumulated read count on the block is another thing to affect the read error rate.

Observation I: *Both factors of P/E cycle and read count affect the read error rate caused by read disturb. Therefore, putting the hot read data onto "strong" blocks, which are insusceptible to read disturb, may reduce the read data error.*

As discussed, routine read refresh operations are expected once the blocks are afforded a considerable number of reads to avoid read disturb errors. In order to disclose their negative influence on I/O responsiveness, we run the benchmark of *websearch_1*, which is abstracted from Umass Trace Repository [42] with or without the routine read refresh operations. Figure 4 shows the relevant results of the long tail latency, and note that we ignore read errors that are brought about by accumulated read operations in this test.

Clearly, the routine read refresh operations greatly impact the I/O long tail latency of I/O response time compared with the case without read refresh. This is because the expected routine read refresh operations are completed among I/O requests even though they do not have an idle interval, which must delay responding to forthcoming I/O requests.

Observation II: *Routine read refresh operations significantly impact the I/O long tail latency. Therefore, carrying out such operations in the idle time intervals among I/O requests may minimize the side-effects on I/O responsiveness.*

Such observations motivate us to mitigate negative effects of read disturb by the following: (1) we dispatch hot read data onto the blocks that are not susceptible to read disturb for reducing the read error rate. (2) We schedule (partial) read refresh operations in the idle time intervals between I/O requests for minimizing their side-effects on I/O responsiveness.

3 READ DISTURB-AWARE WRITE SCHEDULING

3.1 Overview

We first identify hot read data in the current time window, and then map their corresponding write requests (in the next window) to the block that is not susceptible to read disturb. To this end, we first build a mathematical model to classify the block into two categories, i.e., susceptible and insusceptible blocks to read disturb. After that, the write data will be mapped to the insusceptible blocks if they will be frequently read in the near future. Otherwise, the cold read data will be flushed to the block that is susceptible to read disturb.

Table 1. Values of ϕ_0 and ϕ_1 with Varied P/E Cycles (Unit: 10^{-3})

	<1K	<2K	<3K	<4K	<5K	<6K	<7K	<8K
ϕ_0	0.557	0.811	1.073	1.193	1.163	1.116	1.328	2.219
ϕ_1	0.129	0.175	0.252	0.339	0.415	0.459	0.451	0.370

Note: The input data of our model are reported in Ref. [23]. The largest block P/E cycle in the available dataset is 7K, our model predicts the values of ϕ_0, and ϕ_1 in the case of the block P/E cycle is 8K.

3.2 Modeling Read Error Rate

The block P/E cycle does directly impact the read error rate. On the other side, the factor of block read count affects the read error rate, but it also depends on the block P/E cycle. In other words, the read error rate resulted by read operations becomes bigger; in the case of the block P/E cycle, it is relative large. We thus build an empirical non-linear regression model to profile the impacts on the rate of read errors regarding a given block.

$$P_i = \phi_0(PE_i) + \phi_1(PE_i) \cdot R_i + \epsilon_i, \tag{1}$$

where P_i, PE_i, and R_i respectively denote the read error rate, the P/E cycle, and the read count (unit: k) regarding the ith block.

Moreover, ϕ_0 and ϕ_1 are two real-valued functions with the argument of PE, for weighting nonlinear effects of the P/E cycles and block read counts to the read error rate. Then, we use higher-order polynomials to estimate their values:

$$\phi_0(PE_i) = a_0 + a_1 \cdot PE_i + a_2 \cdot PE_i^2 + \cdots + a_p \cdot PE_i^p \tag{2}$$

$$\phi_1(PE_i) = b_0 + b_1 \cdot PE_i + b_2 \cdot PE_i^2 + \cdots + b_q \cdot PE_i^q \tag{3}$$

By taking advantage of the experimental data originally presented in Ref. [23], we employ the *Akaike* information criterion [32] to determine the orders of ϕ_0 and ϕ_1, and the outcomes are $p = 3$ and $q = 4$. Consequently, we can obtain the functions values of ϕ_0 and ϕ_1, when the erase number scales from 1K to 8K, as reported in Table 1. That is to say, we can figure out read error rates of given blocks, for classifying them into two categories, i.e., susceptible blocks and insusceptible blocks to read disturb.

Then, we make use of the coefficient of determination (i.e. ,R^2) and the Mean Absolute Percentage Error (MAPE) to objectively show the accuracy of the modeling on the read error rate. In fact, R^2 is the key output of regression analysis and normally interpreted as the proportion of the variance in the dependent variable that is predictable from the independent variable, and MAPE is another measure of prediction accuracy of a forecasting method in statistics [33]. The results show the coefficient of determination of our model is 0.939. That is, our model reflects that the factors of P/E cycles and read counts can account for 93.9% varieties of read error rate. Besides, the value of MAPE of our model is 13.1%, which indicates our predictions deviate from original data by only 13.1% on average.

3.3 Write Scheduling

Figure 5 illustrates the specifications on the proposed read disturb-aware write scheduling. As seen, the proposed scheme dispatches the received write requests by referring to the pre-identified hot dataset. We regard the data that will be requested multiple time (e.g., > 2 in our tests) in the historical time window as the hot data, and their addresses are collected in the hot dataset.

Then, the write requests whose logic sector numbers are in the hot read set will be mapped to an active block having a small read error rate. The purpose is trying to offset the side-effects of read

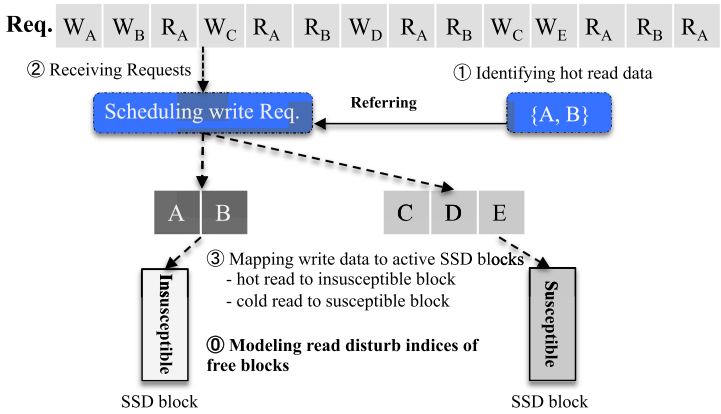

Fig. 5. High-level overview of read disturb-aware write scheduling. *W* and *R* denote write and read requests separately, and their logical sector numbers are represented as *A*, *B*..., and *E*.

disturb on the target block, though such data will be frequently read. Otherwise, the write data will be flushed to a block having a relatively large read error rate. Note that the historical frequent addresses are analyzed as the hot read address set for directing the write requests scheduling, which are considered to be frequently read in the future access.

To be specific, in the case from Figure 5, there are some requests in the I/O queue, labeled as *W* (Write) or *R* (Read), and their subscripts are the logical sector numbers of the requests. By referring to the hot dataset, the requests hit in the set should be flushed to an active block having a small read error rate. Assuming that we have mined the hot read set from the historical requests, and it has members of *A* and *B*, the requests W_A and W_B should be flushed to the insusceptible blocks, which have relatively small read error rate. On the other hand, since requests W_C, W_D, and W_E are not hit in the hot dataset, they should be flushed to the susceptible blocks, which have a relatively large read error rate.

4 REINFORCEMENT LEARNING-BASED READ REFRESH SCHEDULING

After flushing the data to the SSD blocks, read refresh operations are expected once the read count of the blocks reaches a threshold. This section discusses the details about read refresh scheduling by using reinforcement learning to make the best use of idle time intervals between I/O requests.

4.1 Analysis on Idle Time Intervals

In general, the operations of garbage collection or read refresh will be triggered once the SSD device reaches corresponding thresholds, to reclaim the capacity or eliminate negative effects of read disturb, even though the incoming I/O requests are very intensive. Similar to garbage collection, the read refresh operation needs to move valid pages onto other free SSD blocks, and then erase the candidate block for recycling the space.

Moreover, a read refresh operation commonly includes more page moves in contrast to a garbage collection process. Consequently, the average response time will increase and the long tail latency will be significantly worse, as previously illustrated in Figure 4. As mentioned in *Observation II*, the side-effects are expected to be reduced, if we can carry out such operations in the idle time slots between I/O requests. Then, we have further analyzed the inter-request interval distribution of selected traces of *websearch*_1, *hm*_1, *usr*_0, and *ts*_0 from the UMass Trace Repository [42]

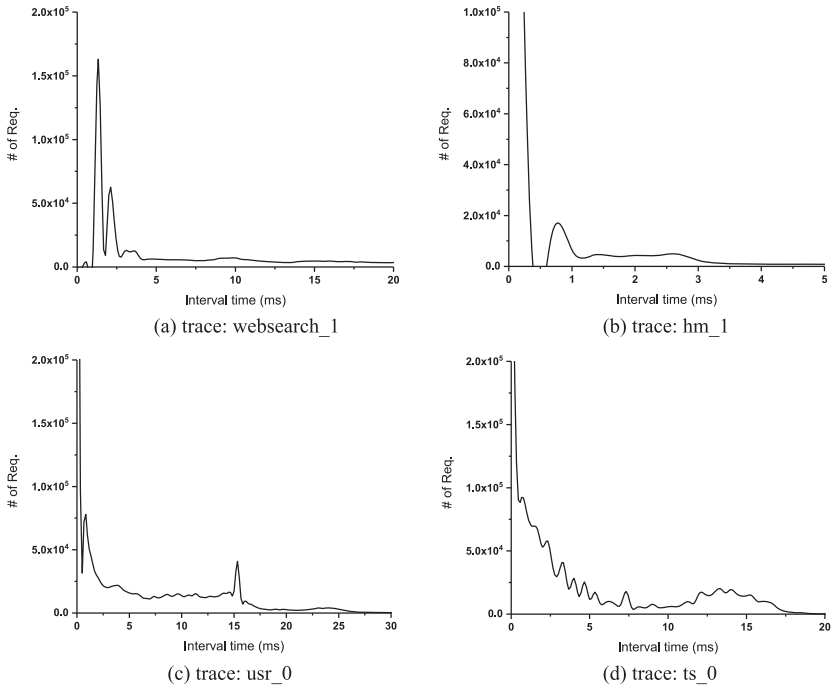

Fig. 6. The distribution of inter-request intervals.

and Microsoft Research Cambridge [45]. In the Figure 6, the *X*-axis represents the length of idle time intervals, and the *Y*-axis shows the number of requests.

We can summarize from the figure, a variety of applications do have some relative long idle time slots, although a major part of interval time may have a small value. Thus, we can dispatch some read refresh operations in some large idle periods to ward off carrying them out in the periods having intensive I/O requests. More importantly, a full read refresh operation consisting of many page moves and an erase may take time to complete, which is larger than the most of idle time intervals of applications. Then, in order to make the utmost use of idle time intervals between I/O requests, certain partial tasks of read refresh, i.e., one or more *page moves* and/or *erase*, are supposed to be performed in such time intervals.

4.2 Reinforcement Learning-Based Read Refresh Scheduling

Different from most machine learning and deep learning approaches, reinforcement learning is a lightweight method and has low space and computation overhead. As a result, it has been applied in the resource-limited SSDs to guide garbage collection scheduling [35] and I/O merging optimization [36]. In other words, the online training feature of reinforcement learning makes it possible to approximate optimal policies with not much overhead. It puts more effort into learning to make good decisions for frequently encountered states, at the expense of less effort for infrequently encountered states [37]. Then, we introduce reinforcement learning in our application context, to identify the best fit (partial) RR operations in the idle time intervals between forthcoming I/O requests.

Figure 7 shows the basic reinforcement model about the interaction between *Agent* (the RR scheduler in our context) and *Environment* (the storage system in our context). *Agent* maintains

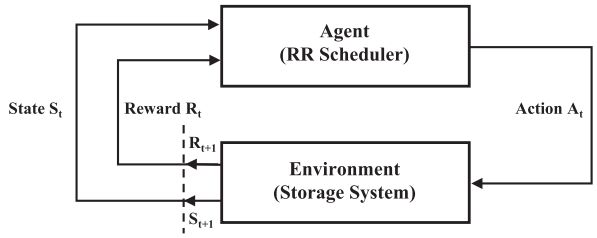

Fig. 7. The basic reinforcement Learning model. Note that S_{t-1}, S_t, S_{t+1} are the states of *Environment* at different time points.

a data structure (e.g., a table) to record the values of corresponding pairs of state and action, for directing the best fit action according to the future state. When *Environment* becomes idle, *Agent* repeatedly works with the following steps for directing (partial) read refresh operations:

(1) *Agent* monitors *Environment* to obtain its state of S_t and the reward of R_t. In general, R_t is related to the I/O response time of the previous I/O request, after completing the last action of A_{t-1}.
(2) *Agent* updates the value in the data structure related to the pair of state and action (i.e., the pair of S_{t-1} and A_{t-1}), as R_t is the outcome of influence to the previous request caused by A_{t-1} in the case of S_{t-1}. The rule of updating the value of the pair of state and action will be specifically described in Equation (4).
(3) *Agent* can provide a fit action of A_{t+1} by searching the maximum value from the table data structure regarding the given state of S_{t+1} in the future.

Note that the reward from the current state acts on *Agent*, for the purpose of giving a feedback associating with the last action. After that, the next state S_{t+1} and the reward R_{t+1} can be observed by *Agent* for carrying out continuous iterations and updating the rewards. Thus, it is possible to obtain certain fit pairs of state and action in a specific situation after aggressive exploring. Because of the limited quantities of states and actions in our context, we employ the basic Q-learning scheme, which holds a data structure of *q-table* and their values are referred as *q-value*s, to fulfill reinforcement learning [34].

In fact, dispatching read refresh operations is to make a decision about doing an action of certain partial read refresh operations (i.e., one or more *page moves* and/or one *erase*), by making use of reinforcement learning. We define many states in the storage system by considering factors of the current request interval, the history request intervals, the previous actions, and the average interval. The reward reflects whether the last action is beneficial to I/O response time or not. If the action has an advantageous effect, the reward is a positive number and to be a feedback to *q-table* for strengthening the selection of action when the same state appears at the next time. Otherwise, the reward is a negative number feedback, which will weaken the selection of action.

To better carry out scheduling on read refresh, we implement the soft threshold to pre-generate read refresh tasks and organize them in a waiting queue. Therefore, the (partial) read refresh operations can be fulfilled in idle time intervals between I/O requests to ward off conducting them in busy time. That is, when the accumulated read count of a specific block is higher than soft threshold T_S, we put the read refresh task on the target block into the waiting queue. Note that when the accumulate read count of a block reaches the hard RR threshold of T_H, it must conduct the normal read refresh operations immediately.

In our reinforcement learning model for guiding read refresh scheduling in SSDs, we have the following definitions:

States: which are comprised of the information on the current inter-request interval, the previous inter-request interval, the previous action, and the erase state. To be specific, the current inter-request interval reflects the intensity of I/O requests from the host system, intuitively directing how much partial RR tasks should be carried out. The previous inter-request interval unveils the most recent history about I/O requests intensity. The previous action represents the decision under the previous situation, and the erase state implies that there is an erasable block or not. The erasable block does not hold any valid pages, whose valid pages have been migrated by the previous partial RR operations.

In our design, we divide `10` groups of current intervals as the main sub-states, 2 types of previous intervals, 2 categories of previous action, and 2 kinds of erase state as other sub-states (that is, we have $10 \times 2 \times 2 \times 2 = 80$ states in total). Specifically, the first sub-state of the current interval is set as $< 0.2ms$, and the step is $0.2ms$. Therefore, the final sub-state of the current interval is $\geq 1.8ms$. The previous action is divided as $< \frac{1}{2}$ maximum page moves and $\geq \frac{1}{2}$ maximum page moves, and the maximum page move is set to eight since a major part of idle time intervals is not enough to complete eight *page moves*. If a partial read refresh action contains an *erase* operation, the previous action will be identified as $\geq \frac{1}{2}$ maximum page move, though the number of page moves in the previous action is $< \frac{1}{2}$ maximum page move. Then, the previous interval time is divided as $< 0.2ms$ and $\geq 0.2ms$. Finally, the erase state is divided as 1 and 0, respectively, representing whether there has been an erasable block or not.

Actions: which mean the varied numbers of *page moves* and/or one *erase*. After an action is decided by selecting the maximum value with the given state by referring to *q-table*, a corresponding action can be triggered. In other words, the instances of actions consist of different scales of *page moves* (1, 2, 4, and 8 page copies in our experiment), one *erase*, and one *erase* with one or more *page moves*. As a result, there are totally nine actions in our model. Note that each action is regarded as a partial RR operation in this article. In addition, the actions in *q-table* having the *erase* operation should not be taken into consideration if there is no erasable block induced by previous partial RR operations.

Especially, we argue that the data (pages) having a large number of read counts are most likely to be accessed again in the near future. Therefore, the basic principle of selecting the data pages to be moved in the target block is to preferentially migrate the most frequent read page, as resetting read counts on them can best mitigate the side-effects of read disturb.

Rewards: they provide feedback to refine the critical data structure in our model, i.e., *q-table* after a specific action. If the response time of the previous completed I/O request is relatively small, the reward value is set as a large positive number. Otherwise, the reward value will be assigned as a negative number, in order to penalize the corresponding action. Therefore, we update *q-table* by following Equation (4).

$$Q(S_{t-1}, A_{t-1}) = (1 - \alpha)Q(S_{t-1}, A_{t-1}) + \alpha[r + \gamma Q(S_t, A_t)], \tag{4}$$

where $Q(S_{t-1}, A_{t-1})$ and $Q(S_t, A_t)$ are the maximum *q-value* with the previous and the current pairs of state and action in *q-table*. The parameter of r means a short-term reward, which is related to the response time of the previous completed request. $Q(S_t, A_t)$ indicates a long-term reward, which is the further decision, compared with the previous decision. In other words, the reward of R_t depends on the short-term reward r and the long-term reward $Q(S_t, A_t)$, so that it is employed for refining *q-table*. The parameters of α and γ are the step size and the discount factor, which are set as typical values, i.e., `0.3` and `0.8` [35, 37]. In our tests, while the response time of the completed I/O request is lower than the 70^{th}, 90^{th}, and 99^{th} percentiles of I/O response time, the reward of r will be respectively set as `1`, `0.5`, and `0`, to express varied levels of positive feedback. Otherwise, the reward of r will be assigned as `-1`.

Table 2. Q-Table Definition Comparison with Existing Reinforcement Learning-Based Methods in SSDs

	State (quantity)	Action (quantity)	Reward
RL-gc [35]	① Current interval (17) ② Previous interval (2) ③ Previous action (2)	① Page move (2) ② Erase (1)	① Response time
RL-merge [36]	① Request type (2) ② Request size (6) ③ Request number (5,103)	① I/O merge (128) ② I/O sort (2)	① Response time ② IOPS
RL-rr (our work)	① Current interval (10) ② Previous interval (2) ③ Previous action (2) ④ Erase state (2)	① Page move (4) ② Erase (1) ③ Page move+Erase (4)	① Response time

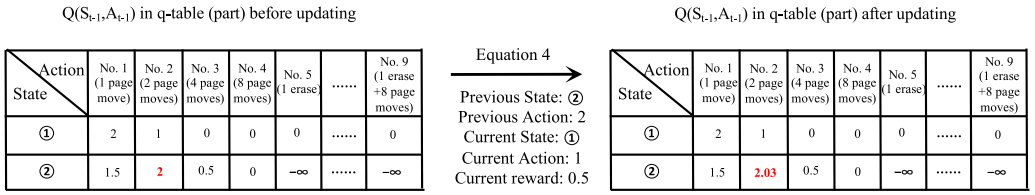

Fig. 8. An illustration of an updating of q-table (left: before updating, right: after updating).

Table 2 compares the definitions of state, action, and reward in our proposal with those in existing reinforcement learning-based methods, including garbage collection scheduling [35] and I/O merging [36]. Obviously, because of applying reinforcement learning in varied background tasks in SSDs, these methods present a distinctive *q-table* definition. Besides, Figure 8 illustrates an example about determining which action is expected with the given state, as well as updating *q-table* by referring to the reward value. In the figure, each row of *q-table* represents a state that is dependent on the selected four determinable factors. As discussed, our model supports `80` states in total, though only 2 states are shown in Figure 8. Each column indicates an action, i.e., nine kinds of partial RR operations. Note that the value of $-\infty$ in *q-table* denotes the corresponding action is never selected for the given state.

Assuming the current reward r is `0.5`, the current state is *State* ①, and the action is *Action No. 1*, i.e., *1 page move*. Considering the previous state was *State* ② and the chosen action was *Action No. 2*, then we can obtain the new *q-value* for the pair of *State* ② and the action of *Action No. 2* by following: $Q(S_{t-1}, A_{t-1}) = (1 - 0.3) * 2 + 0.3 * (0.5 + 0.8 * 2) = 2.03$.

In order to form a fit *q-table* that becomes stable, we randomly select the actions by utilizing ε-greedy initialization in the initial period. That is, the first `1,000` actions are randomly selected in our experiments when running the benchmarks [37]. More exactly, we perform a large ε (`80`%) in the initial period and a small ε (1%) during the rest of the period.

5 EXPERIMENTS AND EVALUATION

5.1 Experiment Setup

The experimental platform was constructed by utilizing a widely used SSD simulator of *SSDsim* (`ver 2.1`) to conduct trace-driven tests [38, 39]. Note that *SSDsim* can only model the performance of SSDs; it cannot store and restore real data for each request. Table 3 presents our TLC settings of *SSDsim* in experiments. In the table, the latencies of write and erase are referred to [40]. Since the

Table 3. Experimental Settings of *SSDsim* (TLC Cell)

Parameters	Values	Parameters	Values (ms)
Page size	8KB	*Read time*	0.085
Page per block	384	*Extra read-retry time*	0.024
RR threshold	25K	*Maximum LDPC level*	7
Soft RR threshold	24.5K	*LSB write time*	0.5
GC threshold	0.3	*CSB write time*	2
Overprovide	0.25	*MSB write time*	5.5
FTL scheme	Page level mapping	*Erase time*	15

Table 4. Specifications on Selected Disk Traces

Trace	Req. #	Read R	Read SZ	Hot R	Concn R	Avg./Max. INVL
websearch_1	1,055,448	99.9%	15.1 KB	90.8%	100.0%	0.03/1 ms
hm_1	609,311	95.3%	14.9 KB	37.9%	54.7%	1.9/1,440 ms
usr_0	2,237,889	40.4%	40.9 KB	46.2%	56.3%	2.7/268 ms
ads	1,532,120	90.5%	31.5 KB	13.7%	0.0%	5.7/224 ms
lun1-1	1,764,623	76.0%	28.9 KB	0.02%	0.0%	0.13/3,770 ms
lun1-2	1,570,278	82.4%	17.5 KB	0.03%	3.0%	0.16/7,022 ms

Note: *Hot R* indicates the ratio of the frequently requested addresses (i.e., the accessed time is not less than 4) to all read address space. *Concn R* represents the percentage of the RR target blocks having concentrated read accesses, in which a major part of read accesses (i.e., 66.7%) target at a small part of pages in the block (i.e., less than 20%).

read latencies depend on the level of LDPC soft decision, we set the basic read time as 0.085 ms, and increase the read time by 0.024 ms per read retry after increasing an LDPC level [24, 41]. That is to say, the read time varies from 0.085 ms to 1.099 ms, according to the LDPC soft decision levels.

To avoid occurrences of uncorrectable read errors, we expect triggering read refresh operations before the block read count reaches the maximum threshold. In our tests, this maximum read limit is 26.3K, which is the outcome of our model by analyzing the collected data. Then, we set the read count threshold of read refresh as 25K, which is slightly less than the (computed) hard threshold. That is to say, when the accumulated read count to a block reaches 25K, a process of read refresh must be activated to reset the read count of the block data.

Because read disturb rarely happens in write-intensive workloads, we employed six commonly used disk traces, including five read-intensive workloads and one read-write balanced workloads. Among these workloads, *websearch_1* is collected from UMass Trace Repository [42], *ads* from Microsoft Production Server [43], *lun1-1* and *lun1-2* are collected from a part of an enterprise Virtual Desktop Infrastructure (VDI) [44], and the other two traces are from Microsoft Research Cambridge [45]. Specifically, *lun1-1* and *lun1-2* are short for *additional-03-2016021719-LUN3* and *additional-03-2016021720-LUN4*. In order to trigger more read refresh operations in the small-scale block traces, we amplify the read requests of two LUN traces by 15 times, and *hm_1* and *usr_0* by 3 times, by referring to Ref. [19]. The detailed specifications on the traces are shown in Table 4.

Besides, we used the following comparison counterparts for measuring the performance of our proposed mechanism:

— *Baseline*: which indicates the default dynamic mapping scheme adopted by *SSDsim*, and the functionality of routine read refresh is supported to fight against read disturb.

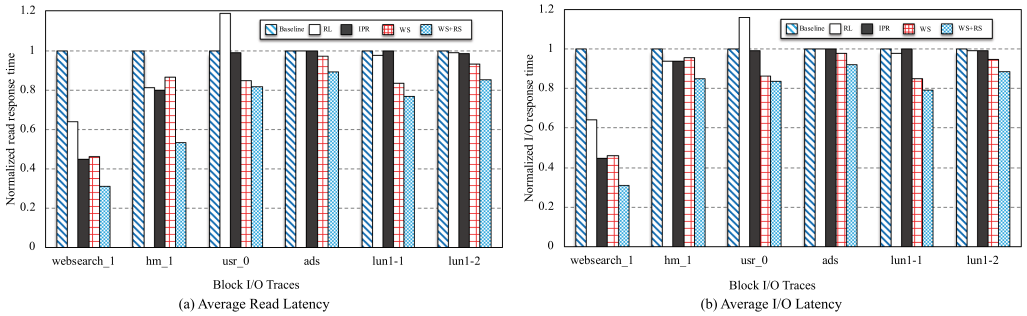

Fig. 9. Average read latency of selected block traces.

— *Read leveling*: which is a software-based mechanism to weaken the side-effects of read disturb [29]. It has been employed as another counterpart in our experiments and labeled as *RL*. We argue that *RL* is one of the most related work to ours. Similarly, it works at Flash Translation Layer of SSDs to distribute hot read data, and makes use of read refresh to relieve the negative effects of read disturb. Specially, *RL* classifies the blocks into three categories, including the normal blocks, the monitor blocks, and the shadow blocks. The latter two kinds of blocks respectively occupy 10% and 15% of total blocks by referring to Ref. [30].
— *In-place reprogramming*: which is also a software-based mechanism for mitigating the negative effects of read disturb [30]. Thus, we take advantage of it as another comparison counterpart, and label it as *IPR* in the article. Similar to *RL*, it manages the blocks as three groups, including the normal blocks, monitor blocks, and the hot-read blocks. In the tests, the latter two groups, respectively, take 10% and 15% of total blocks.
— *Write Scheduling and Read Refresh Scheduling*: which is the newly proposed scheduling schemes on write requests and read refresh operations, labeled as *WS+RS*. It supports both *write scheduling* and *reinforcement learning-based RR scheduling*, for further minimizing the negative effects of read disturb. When using *WS+RS*, we set a soft read refresh threshold as 24.5K, for triggering read refresh in idle time intervals in advance.

  Furthermore, in order to disclose how write scheduling and read refresh scheduling contribute to the performance gain, respectively; we additionally configured our scheme about write scheduling as one counterpart and labeled it as *WS*.

In addition, the maximum number of I/O requests processed in each time window is configured as 8,192 in the evaluation. The logical sector numbers of historical requests in the last time window are leveraged for mining the hot read set. Then, the hot read set is used in direct mapping the write requests in the next time window.

5.2 Tests and Benefit Illustration

To measure validity of the proposed mechanism that aims to mitigate the negative effects resulted by read disturb in SSDs, we use the following four metrics in our tests: (a) *average latency*, (b) *read error rate*, (c) *long tail latency*, and (d) *read latency distribution*.

5.2.1 Average Latency. Read disturb may partially corrupt data in SSD blocks, which will increase the time required for correctly reading data from such blocks. Because the average read latency and I/O latency greatly vary from case to case, we count the normalized read latency and I/O latency of all selected traces, and Figure 9 presents the results.

As seen in the figure, the proposed approach of *WS+RS* outperforms four other comparison counterparts on the measurements of read response time and total I/O response time in all cases. Regarding the related work of *RL* and *IPR*, we see that *IPR* does outperform *RL* in half of the selected traces, i.e., *websearch_1*, *hm_1*, and *usr_1*, which have a high ratio of hot read. In addition, the proposed write scheduling approach, i.e., *WS* generally performs well in all the traces, but does not outperform *IPR* while processing the benchmarks of *websearch_1* and *hm_1*. This is because these two workloads are read-intensive and have a large value of *Hot R*; *IPR* focuses on the read-dominant applications and has attractive performance improvements on them.

More interestingly, *WS+RS* does perform the best, and it respectively cuts down the read response time and the I/O response time by 21.1% and 15.5% on average, compared with the state-of-the-art related work. Even compared with *WS*, *WS+RS* decreases the read response time and the I/O response time by 16.4% and 11.0% on average. This fact verifies the proposed scheduling scheme on read refresh operations can contribute to the reduction of I/O latency to a great extent.

It is worth mentioning the cases of running the traces of *ads*, which has a small ratio of *Hot R*. For example, the ratio of *Hot R* of *ads* is only 13.7%, though it is a read-intensive workload. That is to say, a major part of data pages of such traces are requested less than four times, so that the related work of *RL* and *IPR* could not yield attractive performance improvements. On the other hand, *WS+RS* tries to dispatch the read refresh tasks in the idle time intervals during processing I/O requests, which can minimize the delay on I/O requests. For instance, it can reduce the read response time by up to 11.0% in contrast to others when running the benchmark of *ads*.

Regarding the traces of *lun1-1* and *lun1-2*, we amplify the read requests to trigger more read refresh processes. Although these original two traces have small *Hot R*, they have the uniform frequent read access after read requests amplification [19]. We can see our proposed *WS* and *WS+RS* can yield an attractive improvement from Figure 9, compared with the other methods.

Another general indication revealed in Figure 9(a) is that *WS+RS* can cut down more average read latency when running the traces that have a large portion of concentrated read accesses (e.g., *websearch_1* and *hm_1*). This is because *WS+RS* preferably moves the hot read pages to other blocks in a partial RR operation, which can effectively relieve side-effects of read disturb and then boost I/O responsiveness.

In brief, we argue that *WS+RS* can effectively mitigate negative impacts of read disturb and then boost I/O responsiveness. This is because it maps hot read to the blocks that are insusceptible to read disturb and dispatches (partial) read refresh tasks in the idle time intervals between I/O requests.

5.2.2 Read Error Rate. Raw bit error rate (*RBER*) is a measure of the number of bit errors that occur in a given number of bit transmissions before Error Correction Code corrects the error. We specifically record the bit error rate on read operations (which is defined as read error rate in this article), when processing the read requests of the selected traces. Figure 10 shows the results of read error rate after running the selected benchmarks.

Obviously, the proposed *WS* scheme can greatly decrease the read error rate by up to 15.8%, compared with *Baseline*. Besides, it is able to achieve reduction on read errors by 13.2% on average, compared with the related work of *RL* and *IPR*. This is because the frequently requested data are kept by insusceptible blocks, which can confine read errors caused by intensive read operations on the blocks. In brief, the results verify the proposed scheme of scheduling on write requests can effectively reduce the read error rate.

Another interesting clue shown in the figure is that *WS+RS* achieves almost the same read error rate as *WS*. This is due to read refresh scheduling only working on when to carry out partial RR

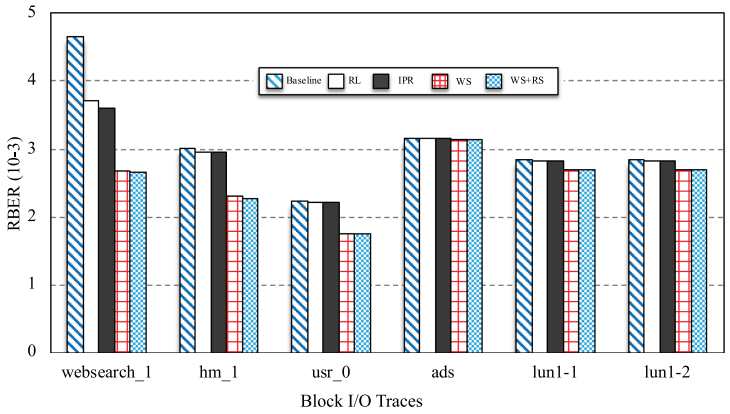

Fig. 10. Read error rate of selected block traces.

operations and what partial operations are expected; it cannot benefit to the reduction of read error rate.

5.2.3 Long Tail Latency. As discussed in our motivations, read refresh operations will postpone some I/O requests and then increase the I/O latency. Figure 11 shows the comparison of long-tail latency (in Cumulative Distribution Function) for read requests. The lines of *Baseline* are almost the lowest ones since it does not adopt any optimization strategies to decrease the long tail latency.

Our proposed *WS+RS* approach exhibits better long-tail latency than that using other selected schemes. As seen, *WS+RS* significantly reduces the long-tail latency by 17.1% and 19.9% at the 99.99th percentile, compared with *RL* and *IPR*. Another noticeable clue is that *WS+RS* decreases the long-tail latency by 20.1% at the 99.99th percentile, compared with *WS*. This fact proves that scheduling on read refresh in idle time slots can efficiently minimize the negative effects caused by performing RR tasks.

5.2.4 Read Latency Distribution. In addition to the metric of the long-tail latency, the latency distribution could show more information on I/O latency. Figure 12 shows the distribution of the read latency after running all benchmarks. On the one side, we can unveil that the most requests can be responded with the latency of 50 ms in the case of *WS+RS*. On the other side, *WS+RS* brings about the least number of requests that are responded with more than 350 ms. Then, we can conclude that the proposed scheme of *WS+RS* can remarkably reduce the read latency for a major part of I/O requests.

5.3 State and Action Illustration in Reinforcement Learning

We have introduced the Q-learning method, which is a basic reinforcement learning mechanism, to assist read refresh scheduling in SSDs. Figure 13 demonstrates some information on the states and actions after running the benchmarks. The left-column figures present the average reward in each episode (i.e., 1,000 RL steps), as a metric to reflect the convergence situation. The middle column figures unveil the occurred actions in 40 randomly selected partial RR operations closing to the end of traces. The right-column figures disclose the most frequently appeared two states in the selected 40 partial RR operations, as well as their corresponding actions.

Let us take *websearch_0* as an example; Figure 13(a)–(c) shows the relevant results. As shown in Figure 13(a), the average reward in each episode has similar value, except for the first episode (i.e., the initial period). We think RL is a relative convergence after the initial period when running

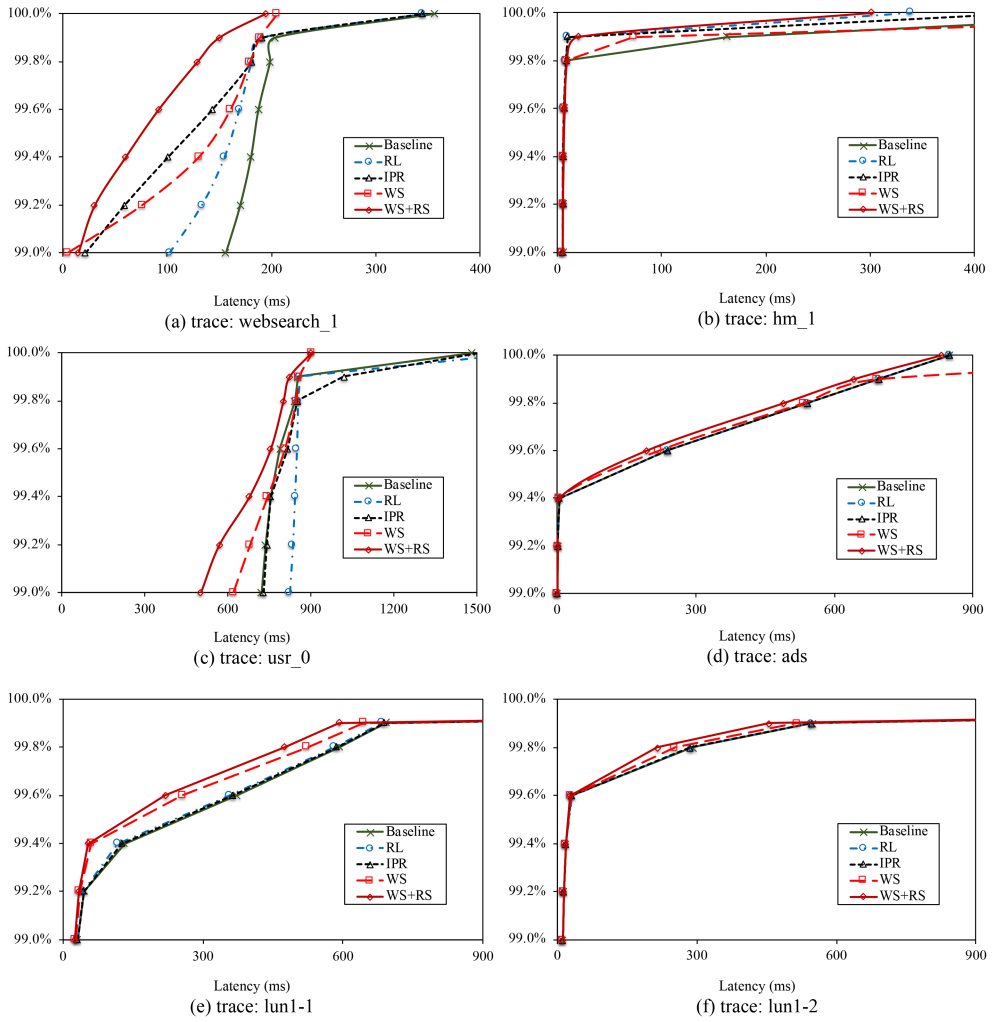

Fig. 11. Comparison of read long-tail latency.

the traces of *websearch_0*. As seen in Figure 13(b), there are *22* times of *Action No. 1* (*1 page move*), *4* times of *Action No. 2* (*2 page moves*), *2* times of *Action No. 5* (*1 erase*), and the others are *Action No. 4* (*8 page moves*). Figure 13(c) indicates the action distribution of *State* ⑥⓪ and *State* ⑦⓪. As seen, *State* ⑦⓪ occurs *8* times, whose corresponding decision is stably *Action No. 4*. This means,*Action No. 4* prefers to be conducted, in the case of *State* ⑦⓪ at this stage when running *websearch_0*. Similarly, a major part of right figures disclose identical decisions of action, which implies *q-table* becomes stable. An irregular fluctuation happens in Figure 13(l), since there has a small ε (1%) to randomly select the action during the rest of the period. But, the decision of *state* ②⓪ is consistently preferred to *Action No. 1*, except for that random selection.

With respect to the convergence of reinforcement learning, we can understand that our method yields an attractive convergence tendency in the case of replaying most traces. But, it is noticeable to mention that the average rewards in Figure 13(m) and (p) that are related to *lun1-1* and *lun1-2* reveal obvious fluctuations. This is because different episodes of these two traces have distinct

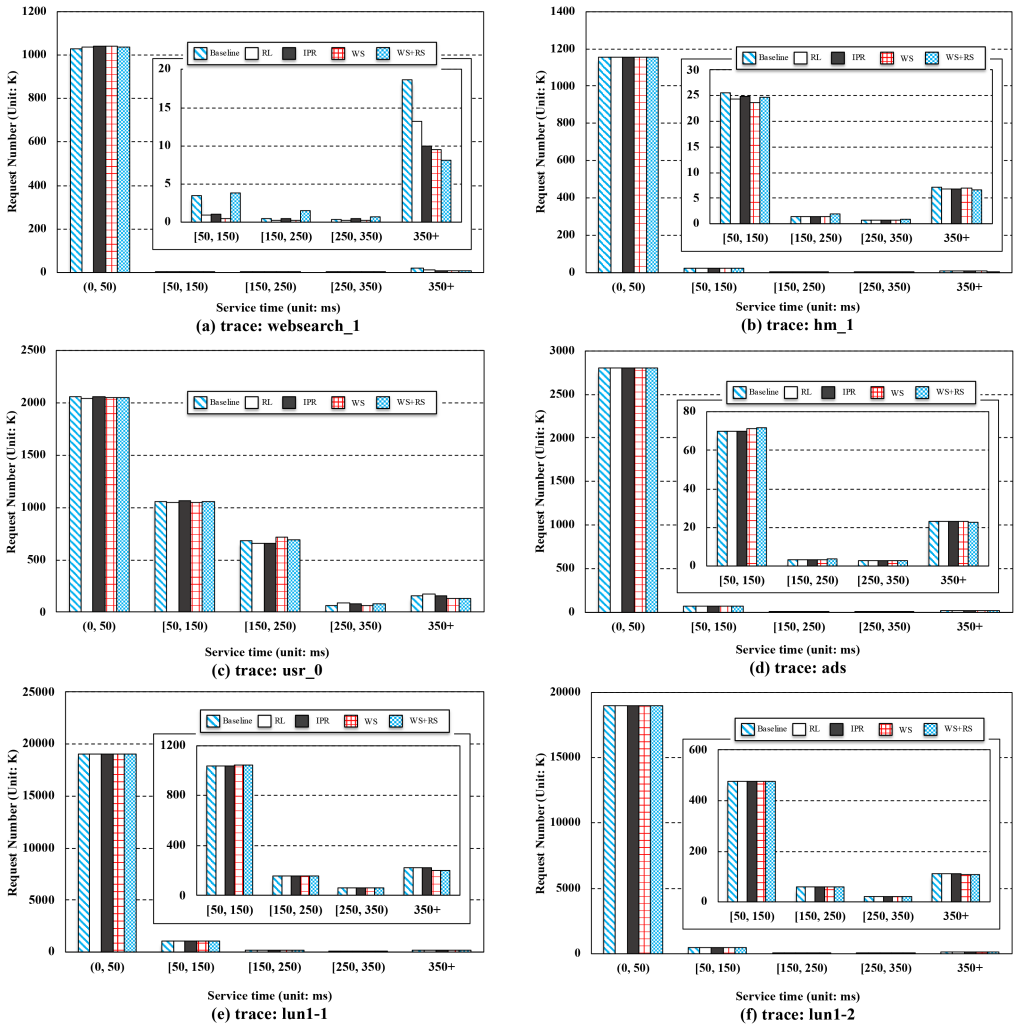

Fig. 12. The distribution of read latency of selected traces.

features of time intervals between I/O requests including the maximum interval and the average interval, which greatly affect I/O responsiveness and the convergence of the algorithm. To be specific, as seen in Figure 13(m), the average intervals in *Episodes* `2` and `6` are `1.16 ms` and `1.37 ms`; both of them are smaller than the average interval of `1.66 ms` in all episodes. On the contrary, the peak points of *Episodes* `3` and `7` have the average intervals of `1.84 ms` and `2.0 ms`. Similarly, Figure 13(p) shows *lun1-2* has the same characteristic to *lun1-1*. As seen, *Episodes* `2, 4,` and `8` respectively have the average time interval of `0.99 ms`, `0.60 ms`, and `1.04 ms`; all of them are lower than the average interval of `1.40 ms` in all episodes.

5.4 Overhead Analysis

This section reports the overhead caused by our proposed mechanism. It first analyzes the erases caused by read refresh and normal garbage collection. After that, the overhead of mining hot read set and mapping write requests to different blocks will be presented.

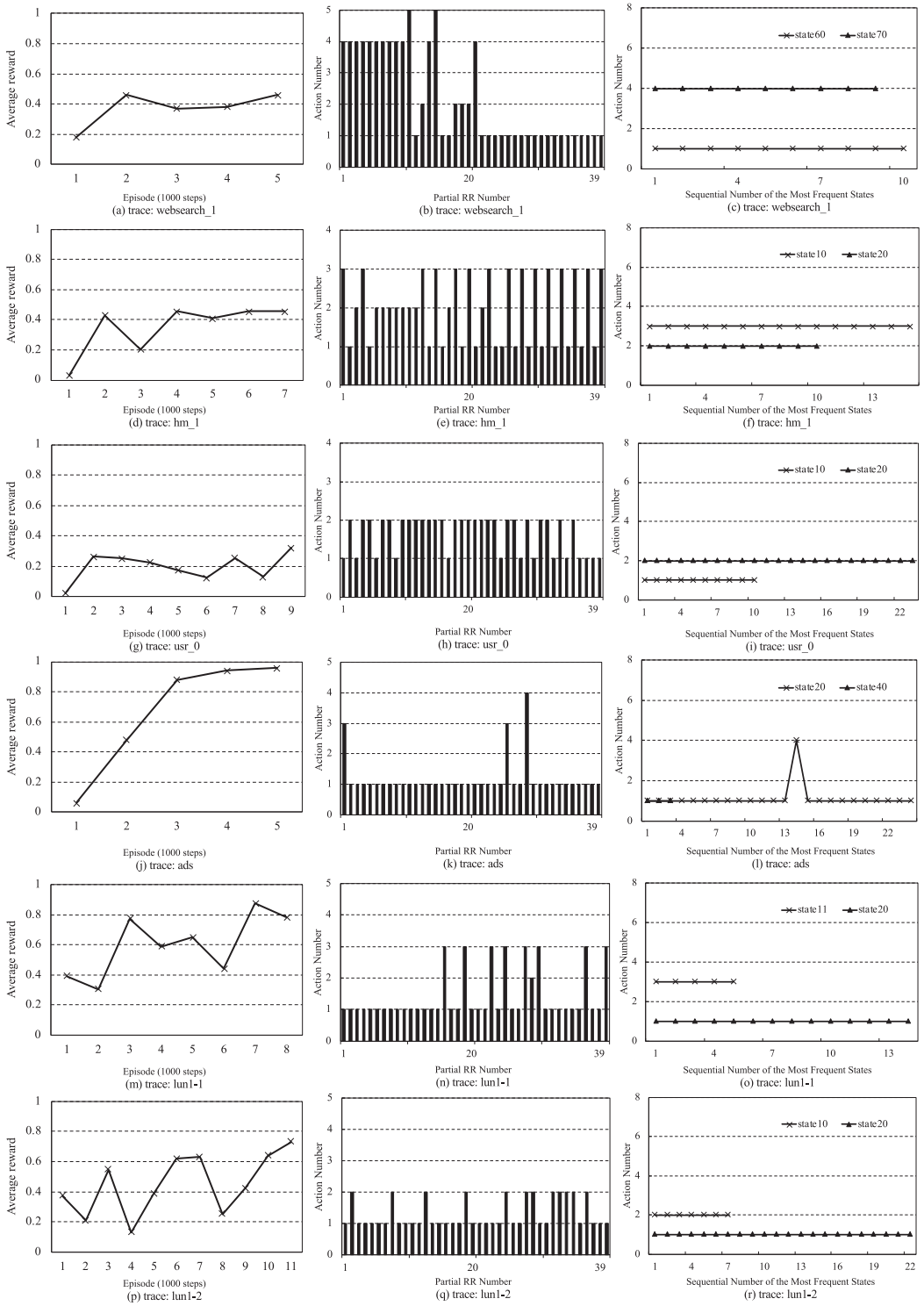

Fig. 13. State and action illustration.

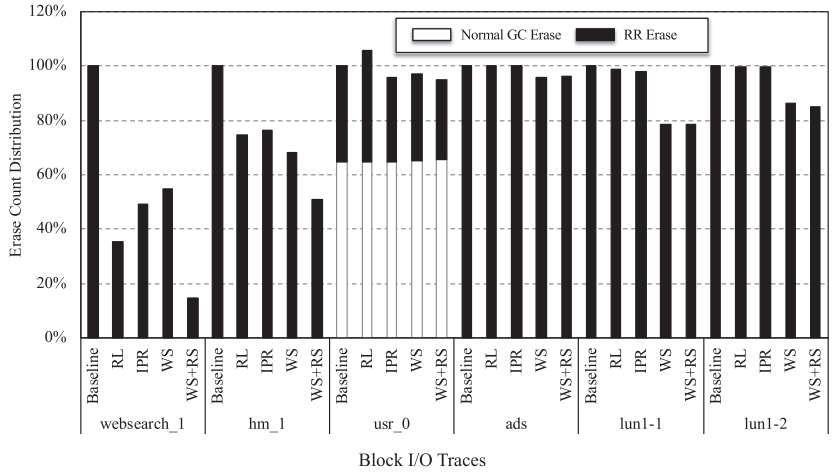

Fig. 14. Breakdown of erase counts contributed by garbage collection and read refresh.

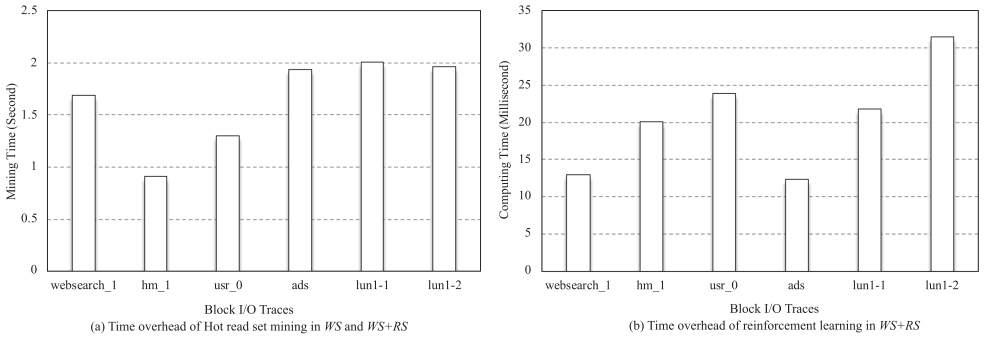

Fig. 15. Time overhead: (a) Hot read set mining in *WS* and *WS+RS*. (b) Reinforcement learning in *WS+RS*.

5.4.1 Erase Overhead. Figure 14 shows the breakdown of the erase counts induced by normal garbage collection and read refresh after running all workloads. The results are normalized to those of *Baseline*. The five read-dominant workloads do not have normal GC erases because the available space after running them is not less than the pre-defined GC threshold. But, the other three workloads have different rates of normal GC erases.

As illustrated in Figure 14, *WS+RS* has the least RR erases among all comparison counterparts, even though it has a few larger numbers of normal GC erases in the trace of *usr_0*. This implies that a part of RR blocks will not be erased until the normal GC operations are triggered, in case that available space is not enough.

5.4.2 Mining and Mapping Overhead. The comparison counterparts of *Baseline*, *RL*, and *IPR* do not carry out mining hot read dataset and mapping relevant write data to specific blocks, so that they do not bring about any mining and mapping overhead. But our proposal results in the mining and mapping overhead. In general, the mining and mapping overhead is related to the number of total requests and the number of frequently read addresses in the trace.

Figure 15(a) shows the time required by mining hot read set and mapping write requests when using *WS* and *WS+RS*. As shown in the Figure 15(a), the mining and mapping overhead is less than `2.01` seconds for all traces, which is acceptable.

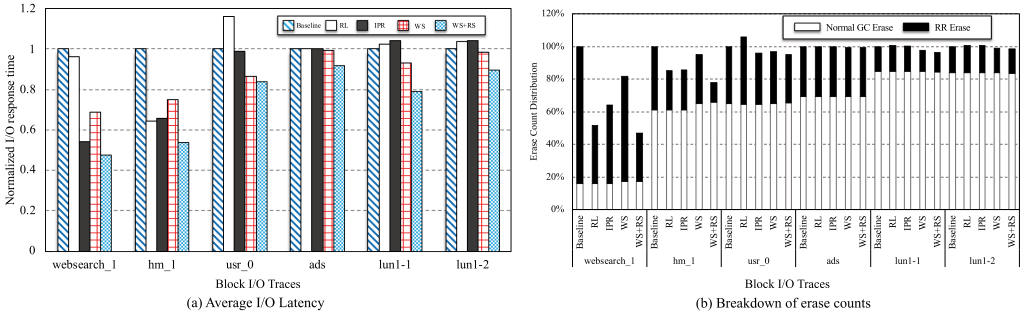

Fig. 16. I/O latency and the breakdown of erase counts with deliberately triggered garbage collection.

5.4.3 Computation Overhead of Reinforcement Learning. The reinforcement learning method is a lightweight model, which has low space and time overheads. In our case, it holds a q-table to direct read refresh scheduling, which only consumes `2.81KB` (= `80` (states)*`9` (actions)*`4B`) taking a negligible amount of memory space in SSDs. Besides, matching the state and the action, as well as updating the relevant values in the q-table according to the rewards in system running, will result in certain computation overhead. According to our measurements, it only takes between `12.3` and `31.5` milliseconds computation time after replaying the selected block I/O traces, as shown in Figure 15(b). Then, we argue that the time overhead caused by the proposed reinforcement learning method is acceptable, even though our tests are conducted on a resource-limited platform that has an ARM Cortex A7 Dual-Core CPU with 800MHz and 128MB of memory.

5.5 GC Impacts on the Effectiveness of RL-Based RR

In Sections 5.2 to 5.4, most of selected block I/O traces do not have erase operations induced by normal garbage collection because most of them are read-intensive. Considering garbage collection is time-consuming, and it competes with read refresh, this section intends to disclose whether garbage collection affects the effectiveness of our proposal or not in contrast to other existing methods.

In order to deliberately trigger garbage collection when replaying the selected read-intensive traces, the simulated SSD is aged to `70%` of its capacity.[2] Figure 16(a) and (b) illustrates the results of I/O latency and Erase breakdown. Compared with the results of having no deliberately triggered garbage collection, which are shown in Figures 9 and 14, we see the proposed RR scheduling approach does have a very similar improvement tendency in the metrics of I/O latency and Erase if certain GC operations are carried out. In brief, we can conclude that garbage collection does not noticeably place negative impacts on the effectiveness of our proposed scheme.

5.6 Summary

With respect to comparing existing schemes and the newly proposed mechanism, we emphasize the following two key observations. *First*, the read disturb level-based data mapping scheme can confine negative impact of the read bit error rate. *Second*, the read refresh scheduling scheme can dispatch partial RR operations in the idle time intervals between two I/O requests to cut down the long tail latency caused by read refresh operations. In brief, we conclude that the proposed scheduling mechanisms are able to significantly reduce the negative effects introduced by read disturb in SSDs.

[2]Running *usr_0* is an exception because more than a half of erase operations are induced by its original garbage collection operations.

6 CONCLUSIONS

We have proposed and evaluated the newly proposed schemes of write scheduling and read refresh scheduling in SSDs by considering the factor of read disturb. To this end, we have built a mathematical model for assessing the read disturb level of block. Then, the frequently read data can be flushed to the blocks that are insusceptible to read disturb, since such blocks are not sensitive to heavy read operations on them.

Furthermore, we have proposed a method of read refresh scheduling by using reinforcement learning to minimize the average read latency. In other words, our proposal intends to take advantage of idle time intervals between I/O requests for carrying out the best fit (partial) read refresh operations. Consequently, the side-effects of routine read refresh operations can be greatly confined.

The evaluation tests show the newly proposed scheme outperforms other comparison counterparts regarding the measurements of read error rate, I/O response time, long tail latency, and erase distribution. In conclusion, our proposed approaches of write scheduling and read refresh scheduling can effectively mitigate the negative impacts of read disturb in modern SSDs.

ACKNOWLEDGMENTS

The authors would like to thank Weihua Liu at Huazhong University of Science and Technology (liuweihua@hust.edu.cn) for his statistical data on the RBER of P/E cycles and read counts in TLC devices.

REFERENCES

[1] C. Matsui, C. Sun, and K. Takeuchi. 2017. Design of hybrid SSDs with storage class memory and NAND flash memory. In *IEEE*, 2017.
[2] R. Chen, C. Zhang, Y. Wang, et al. 2019. DCR: Deterministic crash recovery for NAND flash storage systems. In *IEEE TCAD*, 2019.
[3] R. Micheloni. 2017. Solid-state drive (SSD): A nonvolatile storage system. In *IEEE*, 2017.
[4] M. Bjorling, J. Gonzalez, and P. Bonnet. 2017. LightNVM: The Linux open-channel SSD subsystem. In *FAST*, 2017.
[5] L. Zuolo, C. Zambelli, R. Micheloni, et al. 2017. Solid-state drives: Memory driven design methodologies for optimal performance. In *IEEE*, 2017.
[6] Y. Cai, Y. Luo, S. Ghose, et al. 2018. Read disturb errors in MLC NAND flash memory. *arXiv preprint arXiv:1805.03283*, 2018.
[7] X. Shi, F. Wu, S. Wang S, et al. 2018. Program error rate-based wear leveling for NAND flash memory. In *DATE*, 2018.
[8] W. Lee, M. Kang, S. Hong, et al. 2019. Interpage-based endurance-enhancing lower state encoding for MLC and TLC flash memory storages. In *IEEE TVLSI*, 2019.
[9] I. Narayanan, D. Wang, M. Jeon, et al. 2016. SSD failures in datacenters: What? When? And why? In *SYSTOR*, 2016.
[10] Y. Cai, E. Haratsch, O. Mutlu, et al. 2012. Error patterns in MLC NAND flash memory: Measurement, characterization, and analysis. In *DATE*, 2012.
[11] Y. Cai, E. Haratsch, O. Mutlu, et al. 2013. Threshold voltage distribution in NAND flash memory: Characterization, analysis, and modeling. In *DATE*, 2013.
[12] Y. Cai, Y. Luo, E. Haratsch, et al. 2015. Data retention in MLC NAND flash memory: Characterization, optimization, and recovery. In *HPCA*, 2015.
[13] Y. Cai, O. Mutlu, E. Haratsch, et al. 2013. Program interference in MLC NAND flash memory: Characterization, modeling, and mitigation. In *ICCD*, 2013.
[14] N. Mielke, T. Marquart, N. Wu, et al. 2013. Bit error rate in NAND flash memories. In *IRPS*, 2013.
[15] L. Grupp, A. Caulfield, J. Coburn, et al. 2009. Characterizing flash memory: Anomalies, observations, and applications. In *MICRO*, 2009.
[16] C. Manning. 2012. Yaffs NAND flash failure mitigation. Retrieved from https://yaffs.net/sites/default/files/downloads/YaffsNandFailure Mitigation.pdf.
[17] Y. Cai, Y. Luo, S. Ghose, et al. 2015. Read disturb errors in MLC NAND flash memory: Characterization, mitigation, and recovery. In *DSN*, 2015.
[18] C. Zambelli, P. Olivo, L. Crippa, et al. 2017. Uniform and concentrated read disturb effects in mid-1X TLC NAND flash memories for enterprise solid state drives. In *IRPS*, 2017.

[19] K. Ha, J. Jeong and J. Kim. 2016. An integrated approach for managing read disturbs in high-density NAND flash memory. In *IEEE TCAD*, 2016.
[20] Q. Li, L. Shi, Y. Di, et al. 2017. Exploiting process variation for read performance improvement on LDPC based flash memory storage systems. In *ICCD*, 2017.
[21] Q. Li, L. Shi, Y. Di, et al. 2020. Process variation aware read performance improvement for LDPC-based NAND flash memory. In *IEEE Trans. Reliability*, 2020.
[22] J. Werner, E. Cohen, and T. Canepa. 2014. Read disturb handling for non-volatile solid state media. U.S. Patent Application 13/729,966, 2014.
[23] W. Liu, F. Wu, M. Zhang, et al. 2019. Characterizing the reliability and threshold voltage shifting of 3D charge trap NAND flash. In *DATE*, 2019.
[24] K. Zhao, W. Zhao, H. Sun, et al. 2013. LDPC-in-SSD: Making advanced error correction codes work effectively in solid state drives. In *FAST*, 2013.
[25] B. Kim, J. Choi, and S. Min. 2019. Design tradeoffs for SSD reliability. In *FAST*, 2019.
[26] Y. Seo, J. Yun, W. Lee, and D. Jung. 2013. Memory controller, method of operating the same and memory system including the same, U.S. Patent 14/081 371, 2013.
[27] L. Grupp, J. Davis, and S. Swanson. 2012. The bleak future of NAND flash memory. In *FAST*, 2012.
[28] K. Ha, J. Jeong, and J. Kim. 2013. A read-disturb management technique for high-density NAND flash memory. In *APSys*, 2013.
[29] C. Liu, Y. Chang, and Y. Chang. 2015. Read leveling for flash storage systems. In *SYSTOR*, 2015.
[30] T. Wu, Y. Ma, and L. Chang. 2018. Flash read disturb management using adaptive cell bit-density with in-place reprogramming. In *DATE*, 2018.
[31] E. Cheshmikhani, H. Farbeh, and H. Asadi. 2019. Enhancing reliability of STT-MRAM caches by eliminating read disturbance accumulation. In *DATE*, 2019.
[32] Y. Sakamoto, M. Ishiguro, and G. Kitagawa. 1986. *Akaike Information Criterion Statistics*. Dordrecht, The Netherlands: D. Reidel, 81.
[33] R. Johnson and D. Wichern. 2014. *Applied Multivariate Statistical Analysis*. 6th ed., Pearson Press.
[34] C. Watkins and P. Dayan. 1992. Q-learning. *Machine Learning* (1992).
[35] W. Kang, D. Shin, and S. Yoo. 2017. Reinforcement learning-assisted garbage collection to mitigate long-tail latency in SSD. In *ACM TECS*, 2017.
[36] C. Wu, C. Ji, Q. Li, et al. 2020. Maximizing I/O throughput and minimizing performance variation via reinforcement learning based I/O merging for SSDs. In *IEEE Trans. Computers*, 2020.
[37] R. Sutton and A. Barto. 1998. *Introduction to Reinforcement Learning (1st ed.)*. MIT Press.
[38] J. Li, X. Xu, X. Peng, et al. 2019. Pattern-based write scheduling and read balance-oriented wear-leveling for solid state drivers. In *MSST*, 2019.
[39] C. Gao, L. Shi, Y. Di, et al. 2018. Exploiting chip idleness for minimizing garbage collection-induced chip access conflict on SSDs. In *ACM TODAES*, 2018.
[40] W. Zhang, Q. Cao, H. Jiang, et al. 2018. PA-SSD: A page-type aware TLC SSD for improved write/read performance and storage efficiency. In *ICS*, 2018.
[41] Y. Du, Y. Zhou, M. Zhang, et al. 2019. Adapting layer RBERs variations of 3D flash memories via multi-granularity progressive LDPC reading. In *DAC*, 2019.
[42] Search Engine I/O. Retrieved from http://traces.cs.umass.edu/index.php/Storage/Storage.
[43] Microsoft Production Server Traces. Retrieved from http://iotta.snia.org/traces/158.
[44] C. Lee, T. Kumano, T. Matsuki, et al. 2017. Understanding storage traffic characteristics on enterprise virtual desktop infrastructure. In *SYSTOR*, 2017.
[45] D. Narayanan, A. Donnelly, and A. Rowstron. 2008. Write off-loading: Practical power management for enterprise storage. In *ACM TOS*, 2008.
[46] T. Na, J. Kim, S. Kang, et al. 2016. Read disturbance reduction technique for offset-canceling dual-stage sensing circuits in deep submicrometer STT-RAM. In *IEEE Trans. on Circuits and Systems*, 2016.

Received January 2020; revised July 2020; accepted July 2020